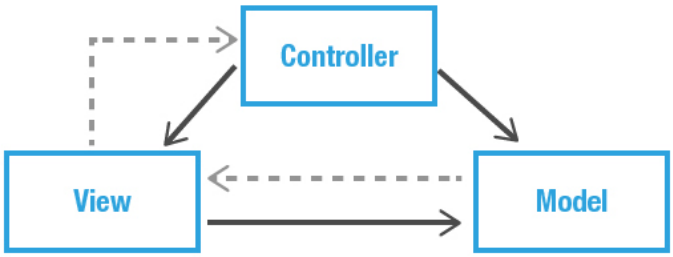
UNIDADE 4 – TÓPICOS AVANÇADOS EM PROGRAMAÇÃO ORIENTADA A OBJETOS
MÓDULO 1 – USANDO MVC – PARTE I

01

1- ENTENDENDO O CONCEITO MVC (MODEL, VIEW, CONTROLLER)

MVC é um padrão de projeto de software. Ele organiza um sistema de modo a oferecer uma separação entre os dados, as regras de negócio, e a interface.

Basicamente, o MVC divide todo um sistema em 3 partes:



Model

Está relacionado aos dados e seus estados. Num sistema que utilize banco de dados, por exemplo, a implementação do modelo dos dados, persistência e acesso, estariam nesta parte.

Controller

Está relacionado aos comandos (ordens) que um sistema realiza para alterar o seu estado. Por exemplo: num sistema de cadastro de pessoas, esta parte poderia oferecer um conjunto de comandos para gerenciar pessoas, tais como: editar pessoas, listar pessoas etc.

View

Está relacionada à visualização (interface). A parte **view** pode solicitar através de um comando da parte **controller**, uma informação da parte**model**. Por exemplo: uma tela (view), através de um comando "Salvar" (controller), acionada por um botão, envia os dados de uma pessoa (model), para serem salvos. Num sistema web, esta parte pode ser implementada por uma página html.

O padrão MVC é amplamente adotado para o desenvolvimento Web. Existem várias bibliotecas neste padrão, tais como: ASP, ASP NET, JSP, PHP, etc. Neste curso veremos o JSP (Java Server Pages). Para entendermos o MVC aplicado a desenvolvimento Web, veremos os principais conceitos envolvidos neste tipo de sistema.

02

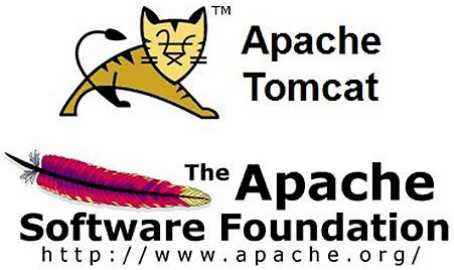
1.1 Servidor de aplicações

Os sistemas webs atuais, usualmente, são executados em um servidor de aplicações.

Um servidor de aplicação é um processo, que é executado em um computador, que gerencia o compartilhamento de processamento, conexão e recursos entre um ou mais sistemas webs.

Entre os tipos de servidores de aplicações disponíveis no mercado, podemos citar:

- Apache Tomcat
- JBoss
- IIS



Por ser de livre instalação, iremos utilizar nesta disciplina o Apache Tomcat.

03

1.2 Sistema Web

Um sistema Web, por definição, é um sistema que provê uma interface de acesso aberto para internet/intranet, para ser utilizada por usuários conectados a rede. Para isso, utiliza protocolos de comunicação (tal como o http) para transmitir dados em formato padronizado (tal como html).

No caso do protocolo http, o sistema segue um modelo cliente/servidor, onde as principais requisições trocadas entre o cliente (browser) e o servidor (servidor de aplicação) são **GET** e **POST**.

GET é utilizado para carregar, por exemplo, uma página, imagens, arquivos etc.	**POST** é utilizado para enviar dados do browser ao servidor, tal como campos de um formulário, arquivos etc.

O endereçamento de uma tela ou operação de um sistema web é feito através de URLs (*Uniform Resource Locators*). URL corresponde ao endereço que digitamos em um browser para acessar determinada página da internet, por exemplo.

Quando um servidor de aplicações recebe uma URL para ser atendida, ele verifica qual sistema Web instalado atende, e repassa o processamento a este sistema para que ele possa atendê-lo, respondendo

com o conteúdo solicitado. Se não houver sistema Web próprio para atender, o servidor de aplicações retorna um erro (404 – not found).

Sistemas Web podem ser utilizados não apenas para exibir páginas, mas também para executar serviços, tais como WebServices. Esta tecnologia permite que se enviem comandos em forma de URLs (ou outros formatos).

Existem várias formas de desenvolver sistemas Web, com diferentes tecnologias e linguagens. Um sistema web pode ser implementado tanto por um executável (exe), como por uma biblioteca java do tipo jar ou bibliotecas .NET aspx, que são encapsuladas em arquivos dll.

Neste curso veremos como implementar um sistema web utilizando o Java e JSP.

URL
Para saber mais sobre URL, acesse http://pt.wikipedia.org/wiki/URL

1.3 Servlets

Quando a Web foi criada, a intenção era disponibilizar páginas de hipertexto estáticas (html), com links, para exibir textos e imagens. Com a popularização da Web surgiu a necessidade de se criar páginas dinâmicas, que oferecessem respostas conforme a requisição do usuário. Para atender esta demanda foram criados **geradores dinâmicos** de páginas html.

Os primeiros geradores dinâmicos eram programas executáveis rodando no servidor web, conhecidos como CGI. Posteriormente, surgiu na plataforma Java os geradores dinâmicos de páginas chamados Servlets.

No Java, em sua forma mais simples, implementamos Servlets fazendo uma classe “filha” de **HttpServlet**. Esta classe tem um método **service**que é chamado quando o servidor de aplicações recebe uma url. Este método basicamente deve montar uma resposta (em formato html) para devolver ao cliente (browser), conforme exemplo abaixo:

```
import java.io.IOException;
import java.io.PrintWriter;

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

@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet {
       protected void service (HttpServletRequest request,HttpServletResponse
response)
```

```
            throws ServletException, IOException
      {
            PrintWriter out = response.getWriter();
            // escreve o texto
            out.println("<html>");
            out.println("<body>");
            out.println("Hello World!!!!");
            out.println("</body>");
            out.println("</html>");
      }
}
```

Note que os parâmetros **request** e **response** contem respectivamente a requisição (que contém dados oriundos da própria URL) e a resposta (que deve ser devolvida ao browser cliente). **Response.getWriter()** retorna um writer que permite ao programador gerar a página dinâmica a ser devolvida ao cliente.

Relacionando com o tópico principal desta unidade, que é o MVC, podemos considerar que o html montado é a parte "View", o Servlet é a parte "Control" e não há uma parte "Model" implementada (seria o "Hello World"???).

Para demonstrar isto, teremos que avançar na complexidade. Mas antes, vamos criar o ambiente para nós trabalharmos posteriormente com um desenvolvimento de uma pequena aplicação web.

05

2 - MONTANDO O AMBIENTE DE DESENVOLVIMENTO

Desenvolver um sistema web completo não é a mesma coisa que desenvolver um sistema desktop, que basicamente pode ser feito com uma classe java. Um sistema web requer:

- servidor de aplicações,
- configuração do servidor,
- ambiente para desenvolvimento (eclipse),
- bibliotecas.

A seguir veremos como montar um ambiente de desenvolvimento totalmente "free".

Primeiro, vamos realizar o download da ferramenta de desenvolvimento que iremos utilizar. No nosso caso será o Eclipse para desenvolvedores Java EE. Para isso acesse o link: http://www.eclipse.org/downloads/ e escolha a opção **Eclipse IDE for Java EE developers**.

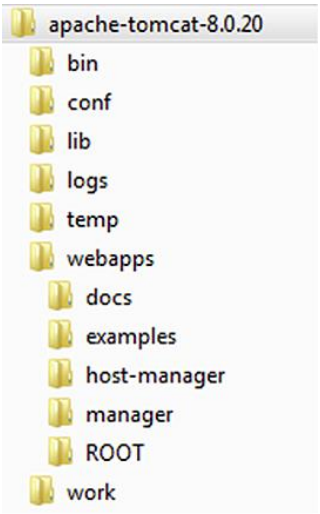
Após baixar, descompacte o eclipse em uma pasta. Em seguida, vamos realizar download do servidor de aplicação que iremos utilizar, que é o Apache Tomcat.

06

2.1 Instalando o Tomcat

Para isto, acesse a página do Apache Tomcat (versão 8 - http://tomcat.apache.org/download-80.cgi), clique na versão de distribuição binária (zip) que seja compatível com seu sistema operacional.

Após o download, descompacte o arquivo zip em uma pasta onde o Apache Tomcat ficará rodando. Se você fez download e descompactou a pasta corretamente, ela deverá ter a seguinte estrutura:



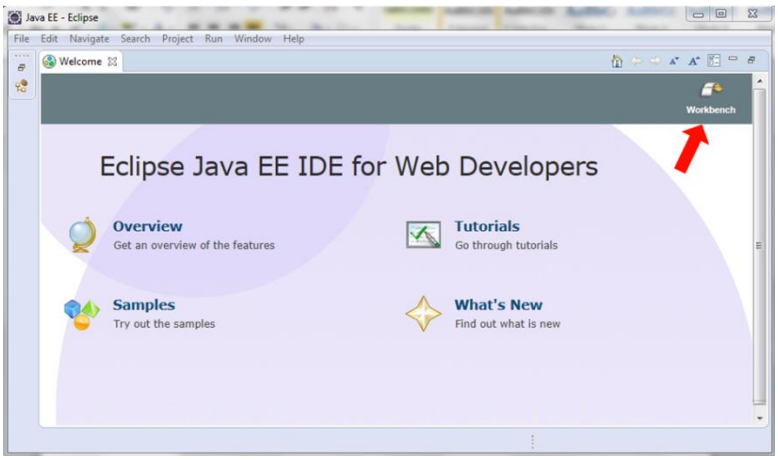
A seguir precisamos instruir o Eclipse para executar os programas web nesta instalação do Apache Tomcat.

07

2.2 Instruindo o Eclipse para utilizar o Tomcat

Ao abrir o eclipse, um caminho de workspace é solicitado. Crie um novo para este módulo, a fim de criar nossos projetos deste curso. Para isto basta colocar o caminho de uma pasta vazia a seu critério.

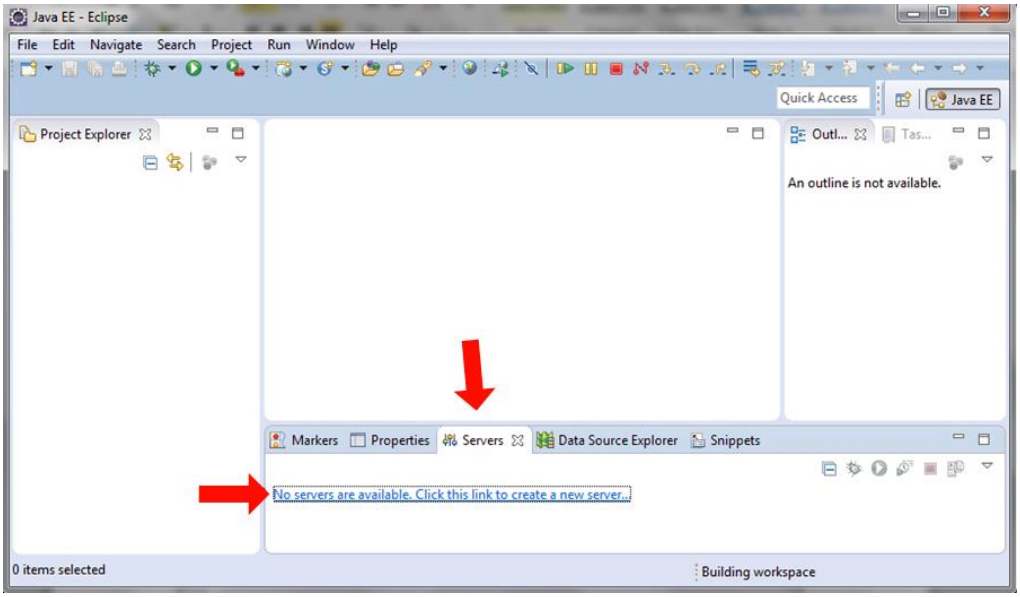
A seguir, a tela do Eclipse é aberta:



08

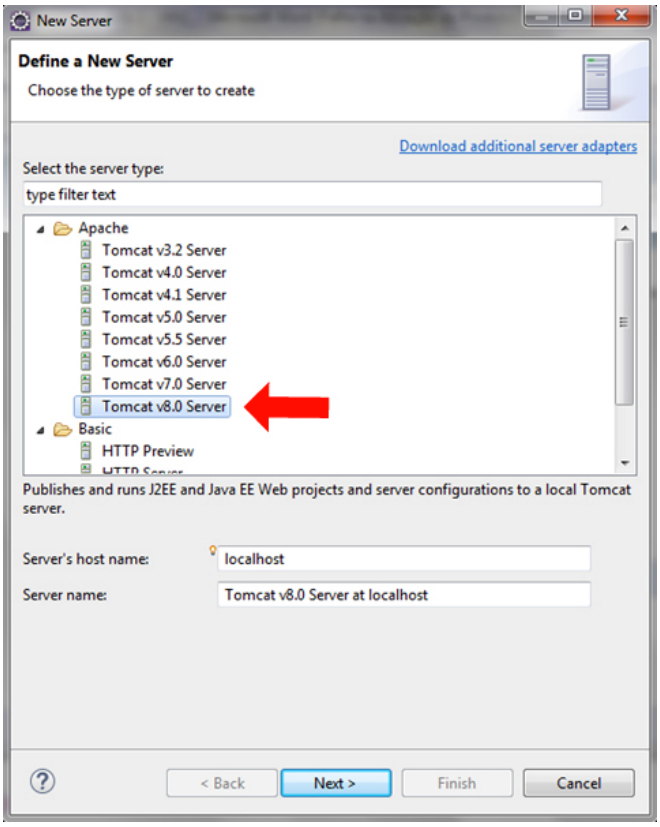
Agora iremos ligar o eclipse ao servidor Tomcat. Para isto siga as instruções abaixo:

- Clique em **Workbench**
- Na aba **Servers**, clique no local indicado:



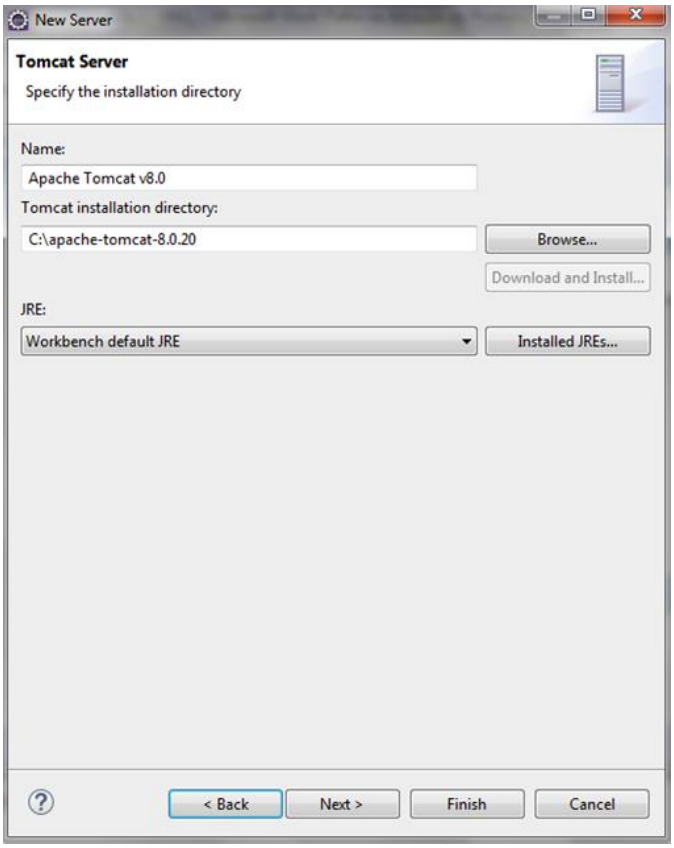
09

- Escolha o servidor **Tomcat v8.0 Server** e aperte **Next**



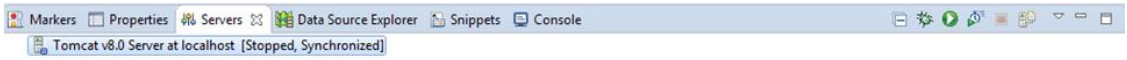
10

- Forneça em **Tomcat installation directory** o caminho da pasta onde você descompactou os arquivos do Tomcat e aperte **Finish**.



11

- Se você fez tudo certo, aparecerá na aba “Servers” uma linha para você iniciar e parar o servidor Tomcat, usando os botões de controle desta aba.



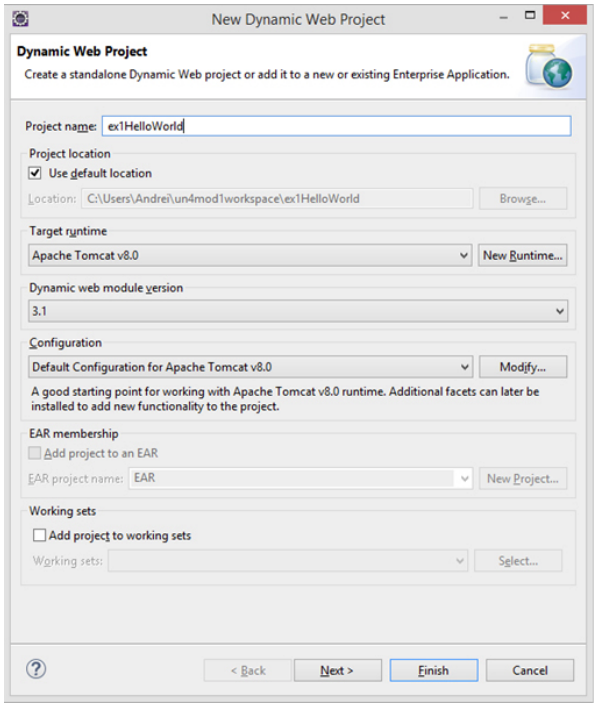
- Inicie o servidor apertando o triangulo verde (*Start the server*), depois, acesse o browser e digite: http://localhost:8080. Se estiver ok, aparecerá um erro 404, porque ainda não há um projeto/aplicação instalado.

12

2.3 Criando um novo projeto de uma aplicação Web

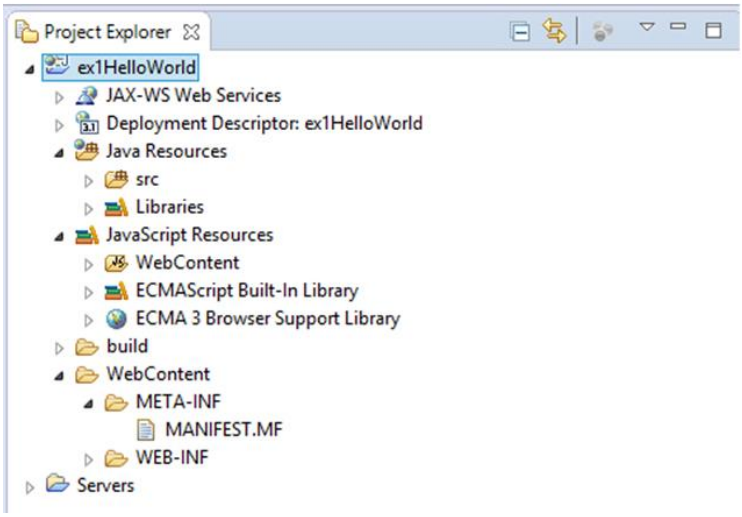
Para demonstrar o padrão MVC, iremos criar um novo projeto web. Para isto, vá em File → New → Dynamic Web Project.

Na tela abaixo, de o nome do projeto de “ex1HelloWorld”:



13

Mantenha as opções default tal qual está na tela acima, e aperte **Finish**. Um projeto será criado com toda uma estrutura pré-montada, conforme figura abaixo:



14

2.4 Minha primeira página html

Para testar o projeto, iremos criar uma página html. As páginas devem ser criadas dentro da pasta "WebContent". Para criar a página, selecione a pasta WebContent e aperte o botão direito do mouse, e escolha a opção **New → HTML File**. Coloque o nome de **hello.html**.

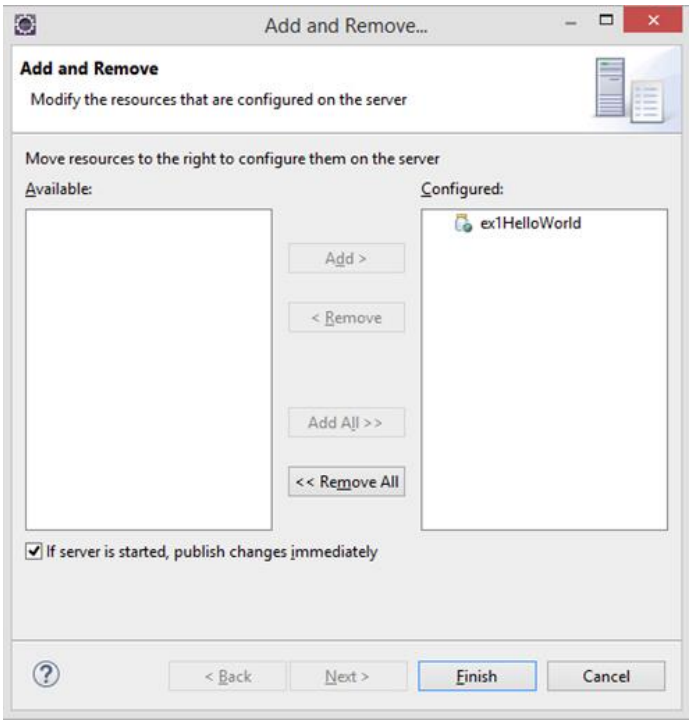
O Eclipse gera um código automático da seguinte maneira:

```
<!DOCTYPE html>
<html>
<head>
<meta charset  ="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

Digite qualquer texto entre as tags **<body>** e **</body>**.

15

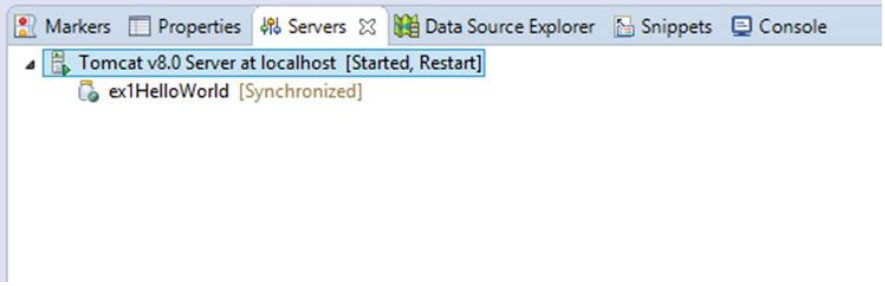
Agora, para testar, adicione seu projeto no servidor criado no item 4.2.2.2 desta aula. Para isto, vá à aba **Servers** do Eclipse, aperte o botão direito do mouse, e escolha a opção **Add and remove**... A seguinte tela será mostrada:



Selecione o **ex1HelloWorld** e aperte **Add** para colocar na caixa **Configured**. E aperte **Finish**.

16

A seguir, se o servidor não estiver rodando, escolha a opção **Start**. O servidor estará rodando se sua aba **Servers** estiver como mostra abaixo:



Então, se tudo estiver certo, você pode ir agora num browser e digitar a url http://localhost:8080/ex1HelloWorld/hello.html

Se apareceu o texto que você digitou em **body** dentro do browser, parabéns, você fez o "V" (View!) mais simples possível do padrão MVC.

17

3 - DESENVOLVENDO PÁGINAS DINÂMICAS

Como já vimos, servlet é um gerador de páginas dinâmicas. Ao invés de ficar preso a um html estático, você pode criar um código java que gera o html que você deseja apresentar, a partir de um banco de dados, ou uma resposta a uma consulta a um sistema, ou ainda imagens e gráficos, conforme escolha do usuário.

Há duas formas simples de fazer servlets:

- Colocando código java embutido em páginas do tipo JSP (Java Server Pages)
- Criando classes filhas de HttpServlet

A partir do projeto que já criamos na parte 4.2.2 (**ex1HelloWorld**), vamos criar exemplos para demonstrar a utilização destes servlets. Veja a seguir.

18

3.1 Criando um JSP simples

Para criar um JSP no projeto Ex1HelloWorld, vá na pasta WebContent deste projeto, clique o botão direito do mouse, e escolha a opção **New → JSP File**. Atribua o nome hello.jsp ao arquivo.

O conteúdo do arquivo é autogerado pelo eclipse, e possui o seguinte conteúdo:

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

Como podemos ver, o formato é igual o html. Mas o JSP pode possuir algumas tags que permitem que seja embutido um código java (scriptlets). A principal tag para isto é a " <%".

Para exemplificar, entre as tags <body> e </body> vamos colocar o seguinte código:

```
<%for (int t=0;t<10;t++) { %>
    OI!
<%} %>
```

Tente executar. Para isso, execute o servidor conforme foi aprendido anteriormente, abra o browser e digite o endereço da página JSP:http://localhost:8080/Ex1HelloWorld/hello.jsp

O que aconteceu? Veja aqui.

Que tal praticar?

Crie um novo arquivo jsp, que construa uma tabela com 100 linhas e 5 colunas (Atenção: a repetição das linhas deverá ser feita com java, e não com copy e paste!
Ah, para fazer esse desafio você terá que aprender como criar uma tabela no html - dica: http://www.w3schools.com/html/html_tables.asp)

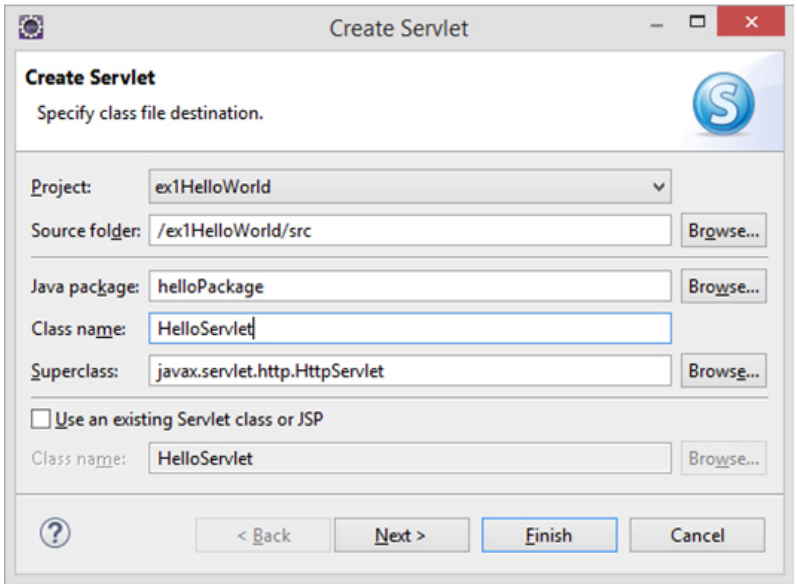
19

3.2 Criando uma classe servlet simples

Neste caso, uma página html será gerada completamente a partir de um código java, e não haverá um arquivo jsp ou html correspondente à página.

Para criar um servlet java, vá na página **Java Resources/src** de nosso projeto, clique o botão direito sobre ela, e escolha a opção **New → Package**. Digite o nome do package como **helloPackage**. Não é recomendado criar "default package" para sistemas Web.

Dentro deste package crie seu servlet. Para isso, clique no package com o botão direito, e escolha a opção **New → Servlet**. Escreva o nome HelloServlet em **Class name**, e aperte **Finish**.



20

O seguinte código será gerado no arquivo **HelloServlet.java**:

```
package helloPackage;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class HelloServlet
 */
@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public HelloServlet() {
        super();
        //TODO Auto-generated constructor stub
    }

    /**
   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
response)
   */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        //
            TODO Auto-generated method stub
    }

    /**
   * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
response)
   */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

}
```

Apesar de parecer complicado, foque em dois detalhes:

- A anotação **@WebServlet** contém o descritor do nome da servlet que deverá ser usado na URL para acessar este servlet com o browser (isto é um recurso desta versão Java EE 6, antes isso era feito no arquivo web.xml).

- No método **doGet** implementamos a resposta a ser dada pelo servlet quando algum usuário acessá-lo com um browser. Basicamente, devemos escrever em “response” o conteúdo html que o browser cliente receberá.

21

Para ilustrar isso, coloque o seguinte código dentro de **doGet**:

```
PrintWriter out = response.getWriter();
out.println("Oi<br>Mundo!");
```

Se por algum acaso a IDE Eclipse não importar automaticamente, no momento em que você salvar o código fonte, a classe "PrintWriter", o código fonte possuirá um erro de sintaxe. Para resolver esse erro, digite manualmente a instrução abaixo junto das demais instruções 'import' já escritas no início do código fonte da classe "HelloServlet".

import java.io.PrintWriter;

Para rodar, reinicie o servidor, execute um browser e acesse a url:

http://localhost:8080/Ex1HelloWorld/HelloServlet

Importante: alterações no código do servlet requerem que as classes java sejam recompiladas e recarregadas pelo servidor web. Para isto, você precisa reiniciar o servidor tomcat toda vez que for testá-lo. A ferramenta Eclipse já faz isso, mas, por precaução, verifique na aba Servers se o código do servidor de aplicação está sincronizado (Synchronized) com o código sendo editado no Eclipse.

Muito trabalho para fazer um simples “hello world”? As principais vantagens de um sistema web são duas:

- o usuário não precisa instalar nenhum aplicativo adicional no computador para acessar o sistema (exceto o browser);

  e

- o usuário pode acessar o sistema de qualquer lugar, bastando estar conectado na internet.

22

3.3 Servlets e frameworks MVC

Como veremos a seguir, os servlets serão utilizados como *controller* do padrão MVC.

O *Controller* nada mais é do que um "despachante" de solicitações: ele é responsável por encaminhar ao código que realiza o atendimento de determinada solicitação (url).

Para organizar a parte *controller*, contamos com várias bibliotecas prontas. A principal delas é a **struts 1**, que é gratuita e também é oferecida pela Apache Foundation.

Estas bibliotecas facilitam muito o trabalho de se organizar o código-fonte em um padrão. Implementar formulários e telas de requisição e resposta usando servlet e jsp é muito trabalhoso. O uso de bibliotecas diminui o nosso código MVC.

Entretanto, pode haver aplicações simples em que seja mais interessante implementar usando servlets. Cabe avaliar cada caso.

23

4 - STRUTS

O struts é um framework para implementação da camada *controller* de um sistema web. Ela organiza as requisições em *actions* que podem ser usadas no âmbito de uma servlet. Tem recursos que permitem mapear padrões de urls e redirecioná-las.

4.1 Instalando o Struts no projeto

Antes de tudo, iremos baixar as bibliotecas necessárias para rodar o struts em nosso projeto. Ele está disponível em:http://struts.apache.org/download.cgi

Baixe exatamente a versão (struts-2.3.20-all.zip).

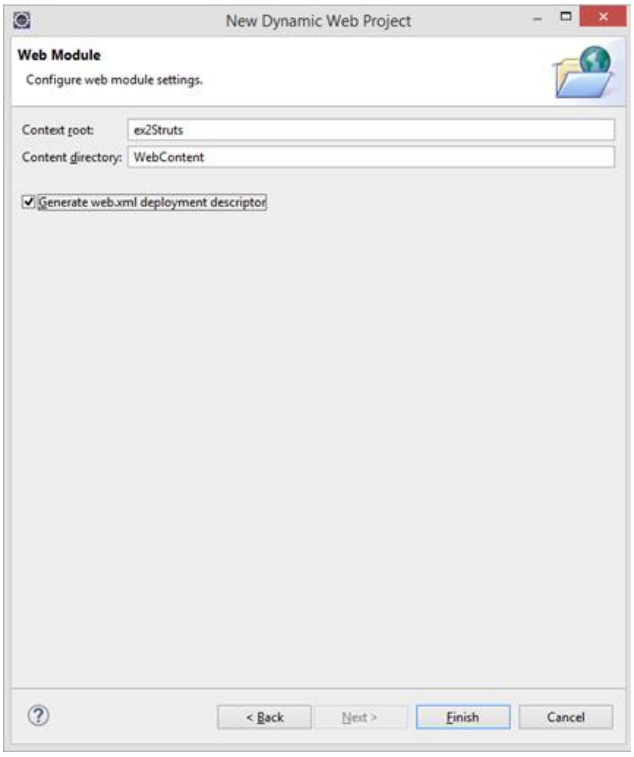
Descompacte o zip em uma pasta. Dos arquivos descompactados, os arquivos de biblioteca do struts que serão utilizados são:

- commons-fileupload-x.y.z.jar
- commons-io-x.y.z.jar
- commons-lang3-x.y.jar
- commons-lang-x.y.jar
- commons-logging-x.y.z.jar
- commons-logging-api-x.y.jar
- freemarker-x.y.z.jar
- javassist-.xy.z.GA
- ognl-x.y.z.jar
- struts2-core-x.y.z.jar
- xwork-core.x.y.z.jar

24

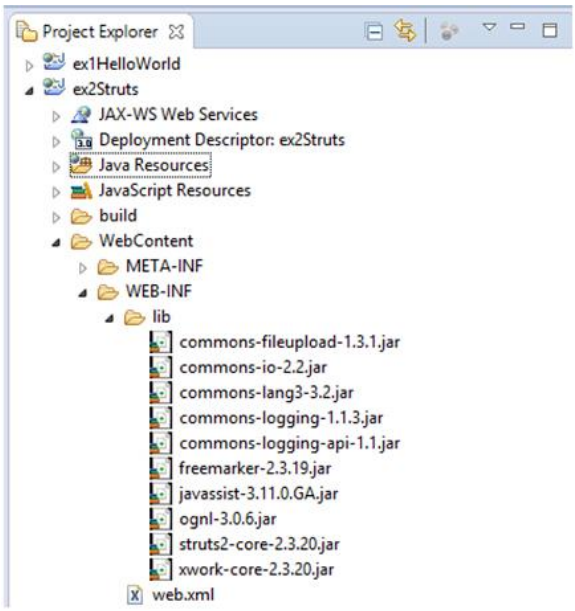
Vamos criar agora o projeto **ex2Struts** que irá ilustrar a utilização do struts para montar nosso sistema MVC. Para isso, escolha a opção **File → New → Dynamic Web Development**

Coloque o nome ex2Struts e aperte **Next** até o fim. Na última tela (ver abaixo), selecione o checkbox **Generate Web .xml Deployment Descriptor** para o eclipse criar o arquivo web.xml no projeto.



25

Copie os arquivos do struts da lista anterior na pasta **WebContent/WEB-INF/lib**.



Pronto, agora iremos criar a parte funcional do sistema.

26

4.2 Criando a classe Action

As classe Actions do Struts são as principais classes a serem implementadas pelo desenvolvedor para criar as regras de negócio do sistema. Elas podem estar associadas as opções de menu a serem executadas, ou aos casos de uso do sistema a ser criado, por exemplo.

Para criar uma classe Action, vá na pasta **Java Resources/src** e crie um package **helloPack**. A seguir, dentro deste package crie uma classe**HelloAction** com o código abaixo:

```
package helloPack;

public class HelloAction {
   private String mensagem;

   public String execute() throws Exception {
      return "success";
   }

   public String getMensagem() {
      return mensagem;
   }

   public void setMensagem(String mensagem) {
      this.mensagem = mensagem;
   }
}
```

O framework Struts irá criar um objeto desta classe quando um usuário solicitar, via browser, uma url que esta action estiver relacionada. Mais em frente veremos como são mapeadas as actions e as urls que as chamam.

27

4.3 Criando a parte VIEW do MVC

Diferentemente de um servlet, a action não monta a view, ou seja, o html que visualizará a resposta. No struts montamos a view com uma página JSP que será interconectada com a action.

Então, para criar o JSP, vá na pasta **WebContent** e escolha a opção **New → JSP File.**

O JSP deverá ter o seguinte código:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
      A mensagem da action é <s:property value="mensagem"/>
</body>
</html>
```

Chame-o de **HelloOK.jsp**.

A linha `<%@ taglib prefix="s" uri="/struts-tags" %>` instrui ao servlet container (servlet que executa o struts framework) que os tags iniciados por “s” são do struts.

A linha `<s:property value="mensagem"/>` instrui que o servlet container irá colocar aqui a mensagem retornada por HelloAction.getMensagem().

28

Agora, iremos criar a tela JSP que faz a requisição da Action. Crie da mesma forma que a anterior e chame-a de **HelloReq.jsp**. Ela terá o seguinte código:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

<form action="hello">
<label>Digite sua mensagem ao mundo:</label><br/>
<input type="text" name="mensagem">
<input type="submit" value="Diga a mensagem!">
</form>
</body>
</html>
```

Há três detalhes importantes neste código:

- A linha `<form action="helloAction">` contém o nome da action que irá atender este formulário. **HelloAction** é a action que foi implementada no item 4.2.4.2
- A linha `<input type="text" name="mensagem"/>` contém um item de formulário com o mesmo nome da propriedade**HelloAction.mensagem**. A ideia é justamente esta: o item de formulário está associado à propriedade da action.
- O botão “Diga a mensagem” é do tipo **submit** e o efeito do botão é executar o código **HelloAction.execute**.

Desta forma, ilustramos como a tela se comunica com o código java da action. Note que há mais facilidade de se implementar formulários com esta técnica do que com servlets puros (sem struts).

29

4.4 Mapeando a action e os JSPs na configuração (struts.xml e web.xml)

Para instruir o struts que o helloAction.java, helloOk.jsp e helloReq.jsp estão relacionados, é necessário criar este mapa no arquivo**struts.xml** e alterar o arquivo web.xml com a declaração para o Struts.

Para criar o arquivo struts.xml, vá na pasta **src**, clique na opção **New → File→Other... →XML→XML File** e crie o arquivo **struts.xml**.

Este arquivo deverá ter o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<constant name="struts.devMode" value="true" />
   <package name="helloPack" extends="struts-default">

      <action name="hello"
            class="helloPack.HelloAction"
            method="execute">
            <result name="success">/HelloOK.jsp</result>
      </action>
   </package>
</struts>
```

Nos últimos arquivos gerados, repare os seguintes mapeamentos:

helloReq.jsp	Struts.xml
`<form action="hello">`	`<action name="hello" class="helloPack.HelloAction" method="execute"> <result name="success">/HelloOK.jsp</result> </action>`

Struts.xml	HelloAction.java
`<action name="hello" class="helloPack.HelloAction" method="execute"> <result name="success">/HelloOK.jsp</result> </action>`	`package helloPack; public class HelloAction {`

Atenção quando estiver definindo os nomes dos arquivos, pois a não coincidência dos nomes nestes arquivos leva a erros de difícil detecção. Geralmente, erros nestes locais são exibidos na página como “HTTP Status 404 - There is no Action mapped for namespace [/] and action name [hello] associated with context path”, por exemplo.

Já o arquivo web.xml que foi criado lá no começo tem a finalidade de informar qual página será aberta quando abrirmos a url do projeto e o nome da aba, dentre outras informações. Esse arquivo deverá ter o seguinte conteúdo:

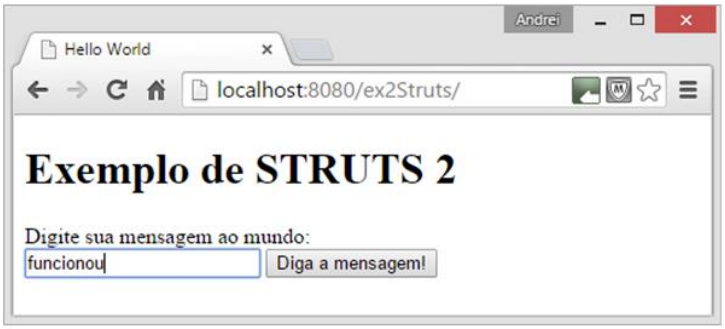
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   id="WebApp_ID" version="3.0">

   <display-name>Struts 2</display-name>
   <welcome-file-list>
      <welcome-file>HelloReq.jsp</welcome-file>
   </welcome-file-list>
   <filter>
      <filter-name>struts2</filter-name>
      <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter
-class>
   </filter>

   <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

Podemos ver, por exemplo, que o descritor <welcome-file> define que o conteúdo HelloReq.jsp será aberto no momento que a URL for acessada.

31

Se tudo estiver correto, você pode adicionar o projeto ex2Struts no servidor Tomcat, reiniciá-lo, abrir um browser e digitar a url:http://localhost:8080/ex2Struts

A seguinte tela aparecerá:



Digite um texto na caixa e aperte o botão. Se tudo foi feito corretamente, aparecerá o HelloOK.jsp:

Acabamos a nossa aplicação MVC simples por aqui, mas na próxima oportunidade iremos ver um exemplo mais completo usando, inclusive, recursos de bases de dados.

Vamos em frente!

32

RESUMO

Nesse módulo apresentamos o padrão de projetos chamado de MVC – Model – View – Controller. Ele organiza um sistema de modo a oferecer uma separação entre os dados, as regras de negócio, e a interface. A camada **Model** está relacionado aos dados e seus estados. Num sistema que utilize banco de dados, por exemplo, a implementação do modelo dos dados, persistência e acesso, estariam nesta parte. O **Controller**está relacionado aos comandos (ordens) que um sistema realiza para alterar o seu estado. Por exemplo: num sistema de cadastro de pessoas, esta parte poderia oferecer um conjunto de comandos para gerenciar pessoas, tais como: editar pessoa, listar pessoas, etc. E a camada **View**está relacionada a visualização (interface). A parte **view** pode solicitar através de um comando da parte controller, uma informação da parte**model**. Por exemplo: uma tela (view), através de um comando "Salvar" (controller), acionada por um botão, envia os dados de uma pessoa (model), para serem salvos. Num sistema web, esta parte pode ser implementada por uma página html.

Para ilustrar os conceitos básicos do MVC, desenvolvemos uma aplicação web. Para isso preparamos um sistema web com um servidor de aplicações (tomcat), configuramos o servidor, o ambiente para desenvolvimento (eclipse) e bibliotecas necessárias. Após a criação de uma página básica em html, vimos o conceito de servlets e o uso do framework struts para a criação da camada controller.

UNIDADE 4 – TÓPICOS AVANÇADOS EM PROGRAMAÇÃO ORIENTADA A OBJETOS
MÓDULO 2 – USANDO MVC – PARTE II

01

1 - CRIANDO UMA APLICAÇÃO MVC DE CADASTRO DE CLIENTES

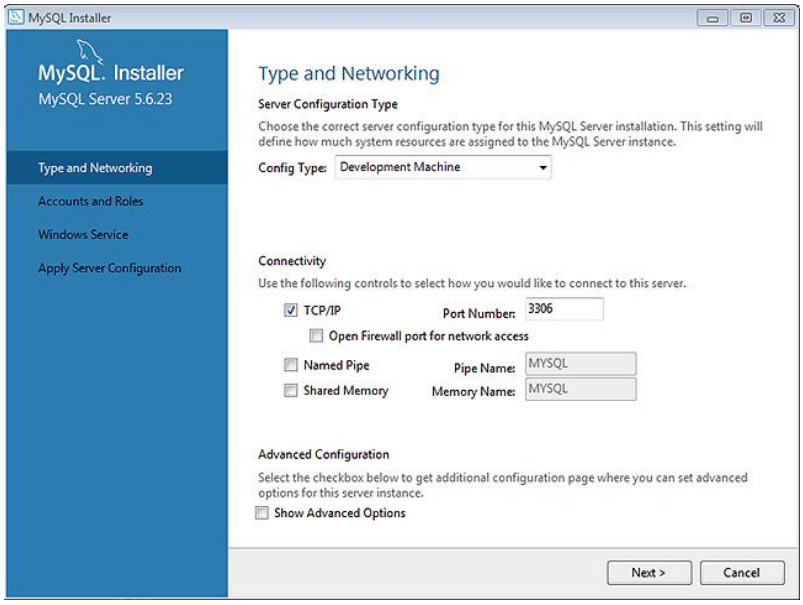
1.1 - A camada Model do MVC

Para implementar um cadastro de clientes precisamos de um recurso que nos permita tornar os clientes "persistentes", ou seja, arquivados, armazenados, memorizados. Para isto, utilizaremos um banco de dados. Por ser livre de direitos autorais (*free*), selecionamos o mySql, que é compatível com vários sistemas operacionais. Para fazer *download* deste banco de dados acesse o link:http://dev.mysql.com/downloads/mysql/

- ***Instalando o MySql***

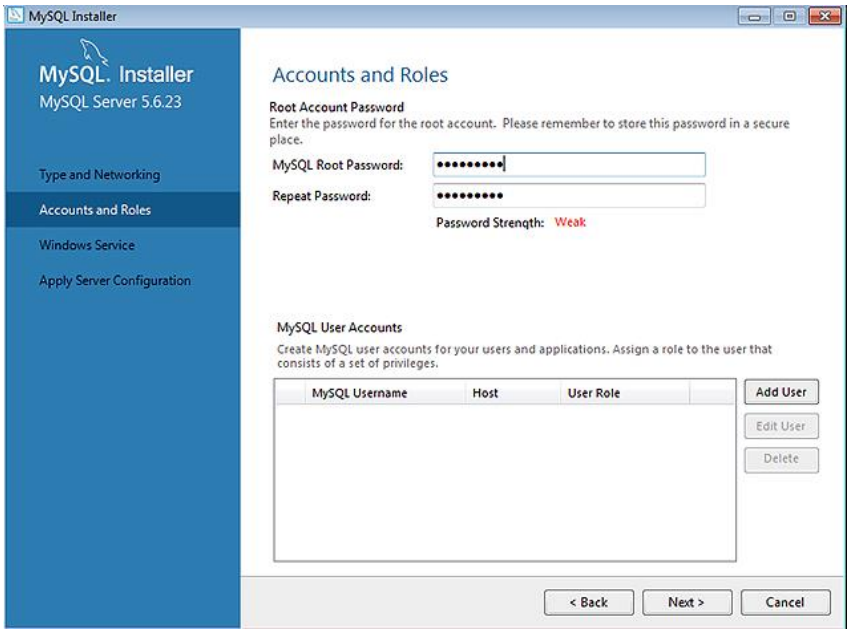
Na instalação do MySql, selecione o modo Development.

Durante a instalação do MySql, fique atento à seguinte tela, você precisará lembrar das configurações dela, tal como a porta TCP/IP configurada:



02

Durante a instalação, você definirá a senha de root; atribua a senha: “admin”. Você pode criar outros usuários se quiser aumentar sua segurança:



Depois de instalado o MySql, para o java comunicar-se com este banco de dados utilizamos uma biblioteca chamada Jdbc. Ela permite acessar as tabelas, campos, e realizar consultas SQL no banco de dados.

03

- ***Criando o modelo da base de cadastro de clientes no MySql***

Não basta acessar tabelas e registros para começar a implementar em java a camada de persistência (Chamamos de camada de persistência a parte do sistema responsável por manter os dados, mesmo quando a aplicação não estiver sendo executada.). É necessário criar um modelo de dados. No nosso cadastro de clientes, iremos implementar um modelo simples, para facilitar uma primeira compreensão do que é a camada*Model* do MVC.

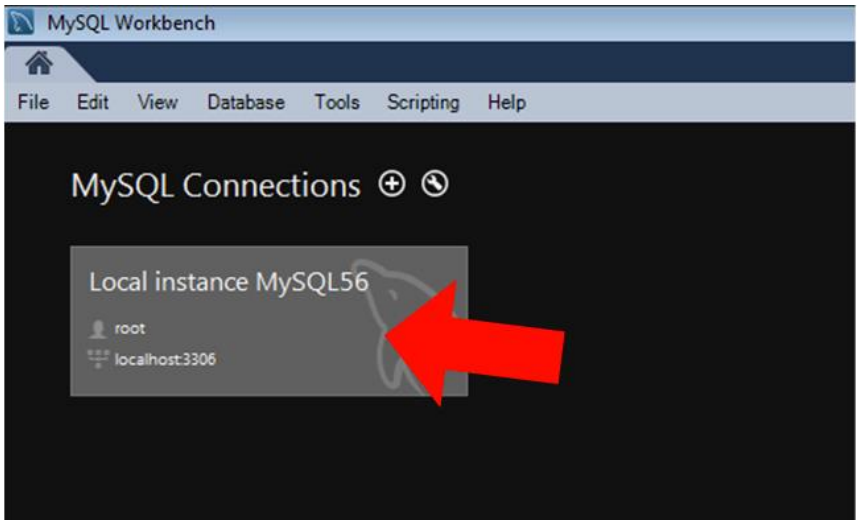
O nosso modelo basicamente será constituído por uma tabela de clientes, com os seguintes campos:

ID do cliente Nome do cliente Data de nascimento Endereço Complemento Cep Telefone

Para começarmos a trabalhar com o MySql, execute o Workbench, que é uma ferramenta que vem com a instalação do MySql. Esta ferramenta permite executarmos scripts SQL para criação de bases e tabelas.

04

No Workbench, clique no local indicado para se conectar com o servidor MySql.



Na aba “Query 1” você pode digitar o script que você deseja executar no servidor. Primeiramente, temos que criar uma base de dados com o seguinte script:

```
create database MyDB
```

Para executar o script acima clique no botão executar (). A seguir, criaremos a tabela de clientes, com o seguinte script SQL. Para isso apague o script anterior e escreva o seguinte código:

```
create table MyDB.Cliente (
  idCliente     int not null,
      nome      varchar(64) not null,
      dataNascimento    int,
      endereco      varchar(256),
      complemento   varchar(256),
      cep     varchar(10),
      telefone      varchar(32)
)
```

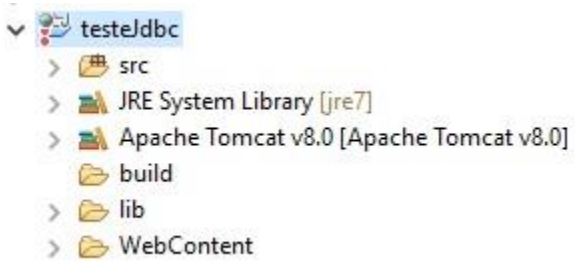
Infelizmente não iremos explorar a linguagem SQL em profundidade neste curso. O SQL é assunto suficiente para um curso à parte. Veremos aqui somente o que for necessário para definirmos a camada modelo de nosso sistema de cadastro de clientes.

05

- ***Testando o JDBC***

Para testarmos se o java consegue acessar a base de dados de clientes (MyDB), vamos criar um projeto java (Java Project) **testeJdbc**.

Clique com o botão direito no projeto e crie uma pasta lib (New-> Folder):

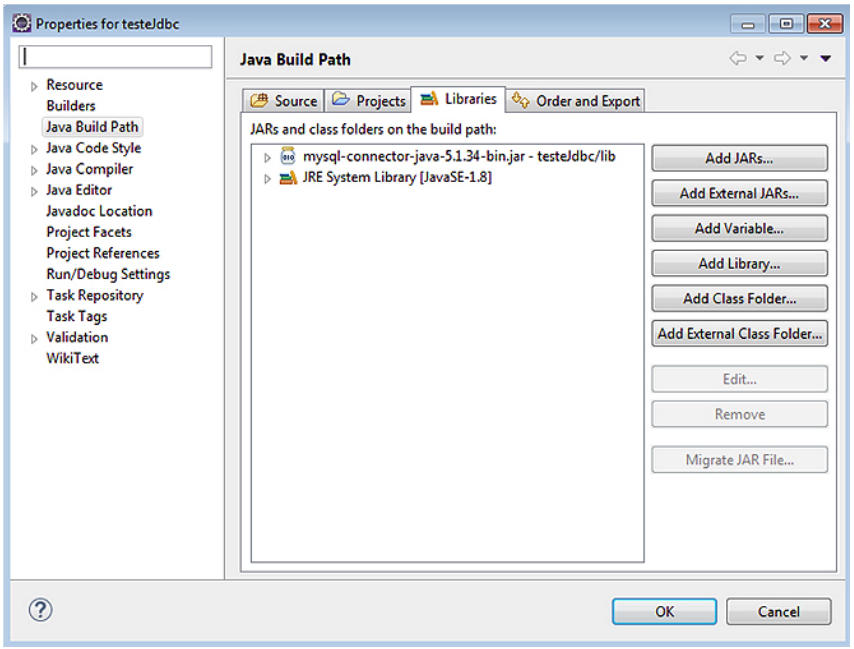


Para acessarmos o driver jdbc que conecta o java ao mySql, precisamos copiar a biblioteca jar **mysql-connector-java-5.1.34-bin.jar** da pasta **MySQL\Connector.J 5.1** (ver onde você instalou seu MySql) na pasta **lib**. Caso não exista essa pasta, você poderá fazer o download do conector no site (http://dev.mysql.com/downloads/file/?id=459313)

A seguir, deve-se instruir o Eclipse/java a utilizar esta biblioteca. Para isso:

1. clique o botão esquerdo do mouse no projeto e escolha a opção **Properties**.
2. Selecione no menu lateral **Java Build Path**
3. Clique na aba **Libraries** e aperte Add **JARs**....
4. Adicione então o arquivo jar.

A tela ficará conforme abaixo:



06

Então, você cria uma classe **TesteJdbc** que deverá ter o seguinte código:

```
package testeJdbc;

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

public class TesteJdbc {

  public static void main(String[] args) {
    try {
     String url = "jdbc:mysql://localhost/MyDB";
     String usuario = "root";
     String senha = "admin";
      Connection conexao =
                 DriverManager.getConnection(url, usuario, senha);
     System.out.println("Conectado!");
```

```
      conexao.close();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }

}
```

O parâmetro *password* (senha) no método **getConnection** deve ser a senha que você configurou em sua instalação MySql.

Execute o **TesteJdbc**. Se aparecer "Conectado!" na aba **Console** de seu eclipse, parabéns! Tudo foi feito certinho.

07

2 - Criando o projeto do cadastro de clientes

Agora, vamos criar o projeto **ex3Clientes** que implementará a nossa aplicação Web de cadastro de clientes.

Para facilitar a criação deste projeto, utilizaremos a mesma configuração usada no projeto **ex2Struts** que foi feito no módulo anterior. Para isso, crie um novo Workspace chamado uni4Mod2POO e crie um projeto do tipo Dynamic Web Project chamado **ex3Clientes**. Não se esqueça dos seguintes passos:

1. Selecione a opção de criação do arquivo Web.xml no momento da criação do projeto.
2. Inclua o servidor Tomcat, conforme especificado no módulo anterior.
3. Inclua as bibliotecas do Struts necessárias no diretório WebContent\WEB-INF. Não se esqueça de incluí-las nas Libraries da opção Java Build Path.

Iremos implementar três telas:

ListaClientes

Que mostrará a lista de clientes e ao lado de cada cliente terá um link Editar e Excluir. E também terá um botão Novo, para criar novos clientes.

EditaCliente

Que mostrará os campos para serem editados. Esta tela também irá ser utilizada para criar um novo cliente.

Erro

Esta tela será utilizada para exibir mensagens de erro..

08

Primeiramente, copie o conector jdbc do **MySql mysql-connector-java-5.1.34-bin.jar** para a pasta **WebContent\WEB-INF\lib** do projeto **ex3Clientes**.

Crie a classe **ListaClientes.jsp**. Para isto, vá em WebContent, clique com botão esquerdo e escolha a opção **New → JSP File**. Para que esta opção esteja aparecendo você deve estar com a perspectiva **Java EE** ativada.

O conteúdo do jsp será o seguinte:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Lista de clientes</title>
</head>
<body>
   <h1>Lista de clientes</h1>
   <form action="listaClientes">
      <label for="filtro">Filtrar por:</label><br/>
      <input type="text" name="filtro"/>
      <input type="submit" value="Filtrar"/>
      <input type="submit" value="Novo"/>
      <br>
      <table border=1>
      <s:iterator value="clientes">
        <tr>
          <td>
                  <s:property />
                  </td>
                  <td>
                        <input type="submit" value="Editar"/>
                  </td>
                  <td>
                    <input type="submit" value="Excluir"/>
                  </td>
                </tr>
          </s:iterator>
          </table>
   </form>
</body>
</html>
```

09

Note que estamos criando uma action **listaClientes**. Para isto, precisamos criar a classe desta action. Vamos criar então uma package**clientesPack** para as nossas actions relacionadas com clientes. E dentro dela, criaremos uma nova classe **ListaClientesAction**, que terá o seguinte código:

```
package clientesPack;

import java.util.ArrayList;
import java.util.List;

public class ListaClientesAction {
        private List<String> clientes;

   public String execute() throws Exception {

            clientes = new ArrayList<String>();
            clientes.add("Fulano");
            clientes.add("Sicrano");
            clientes.add("Beltrano");

            return "success";
   }

   public List<String> getClientes() {
      return this.clientes;
   }
}
```

Como podemos ver, há uma lista fixa de clientes. Por enquanto utilizaremos esta lista, posteriormente iremos obtê-la do banco de dados.

10

Falta alguma coisa? Sim! Precisamos redefinir o arquivo **struts.xml** para mapear a classe de action com a url. Então, o struts.xml fica assim (o código adicionado foi marcado):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
   <constant name="struts.devMode" value="true" />
   <package name="clientesPack" extends="struts-default">
         <action name="listaClientes" class="clientesPack.ListaClientesAction"
method="execute">
               <result name="success">/ListaClientes.jsp</result>
         </action>
      </package>
</struts>
```

Você já pode executar o programa para ver como está ficando. Ative o servidor, coloque o ex3Clientes nele e abra no browser o link http://localhost:8080/ex3Clientes/listaClientes para testar.

Entendeu o que fizemos? Criamos uma Action chamada listaClientes que retorna uma lista de clientes. Essa lista é recebida pela página ListaCliente.jsp (View) que imprime na tela com os botões de edição e exclusão:

Obviamente, os botões ainda não funcionam, mas a seguir mostraremos como fazer tudo isso funcionar.

11

3 - JAVABEANS

Javabeans são classes que representam entidades de negócio.

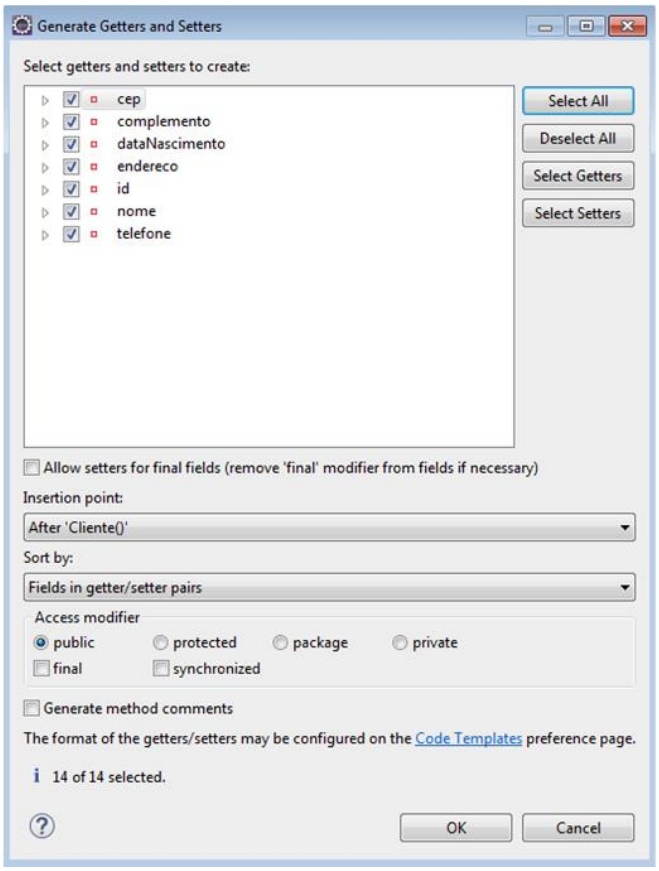
No nosso caso, o cliente seria um ótimo candidato para ser uma classe Javabeans. Estas classes nada mais são do que uma classe simples java onde cada propriedade armazena um campo de negócio (tal como nome, endereço etc.) e cada uma possui um método set e get, para alterar e retornar o valor de cada campo.

Então vamos criar o Javabeans **Cliente** que terá o seguinte código:

```
package clientesPack;

public class Cliente {
        private int id;
        private String nome;
        private int dataNascimento;
        private String endereco;
        private String complemento;
        private String cep;
        private String telefone;

        public Cliente() {
        }
}
```

12

Você precisa gerar os sets e os gets de cada propriedade. Para isto, posicione o cursor após o construtor, e escolha a opção **Source → Generate Setters and Getters**... Aparecerá a tela abaixo:



Aperte **Select All** e a seguir **OK**. O código dos sets e gets serão gerados automaticamente pelo Eclipse.

Pronto, seu primeiro javabeans está pronto.

Atenção: não confunda javabeans com EJB (Enterprise Java Beans), que é um javabeans mais sofisticado, que contém estruturas para serialização e outras.

13

4 - Obtendo a lista de clientes do banco de dados

Agora, na action ListaClientes, vamos corrigir o código de getClientes para que ele obtenha a lista de clientes diretamente do banco de dados. Para isto, iremos construir uma classe que será responsável

por se comunicar com o banco de dados. Estas classes são comumente chamadas de **DAO** (Data Access Objects).

Então, vamos criar uma classe chamada **ClienteDao**, que será responsável pela persistência dos clientes! Esta classe deverá ter o seguinte código:

```
package clientesPack;

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

public class ClienteDao {
	private Connection connection;

	public ClienteDao() throws SQLException, ClassNotFoundException {
		Class.forName("com.mysql.jdbc.Driver");

		String url = "jdbc:mysql://localhost/MyDB";
		String usuario = "root";
		String senha = "admin";

		connection = DriverManager.getConnection(url, usuario, senha);
	}

	public List<Cliente> carregaClientes() throws SQLException {
		PreparedStatement stmt = null;
		ResultSet rs = null;

		List<Cliente> lista = new ArrayList<Cliente>();

		try {
			stmt = connection.prepareStatement("select * from
MyDB.Cliente");

			rs = stmt.executeQuery();

			while (rs.next()) {
				Cliente c = new Cliente();
				c.setId(rs.getInt("idCliente"));
				c.setNome(rs.getString("nome"));
				c.setDataNascimento(rs.getInt("dataNascimento"));
				c.setEndereco(rs.getString("endereco"));
				c.setComplemento(rs.getString("complemento"));
				c.setCep(rs.getString("cep"));
				c.setTelefone(rs.getString("telefone"));
				lista.add(c);
			}
```

```
            }
            finally {
                  if (rs != null)
                        rs.close();
                  if (stmt != null)
                        stmt.close();
            }

            return lista;
      }

      public void close() throws SQLException {
            connection.close();
      }
}
```

Como podemos ver, o método **carregaClientes** carrega todos os clientes que estão na tabela (“select * from MyDB.Cliente”) e os coloca em uma lista de javabeans clientes.

Nesta classe também serão implementados os métodos que são responsáveis por criar, alterar e excluir clientes. Mas primeiro, vamos mostrar os clientes que estão na base de dados. Ainda falta instruir o struts e a tela jsp entender o javabeans cliente. Vamos em frente?

14

A action ListaClienteAction precisa ser adaptada da seguinte forma:

```
package clientesPack;

import java.util.List;

public class ListaClientesAction {
        private List<Cliente> clientes;

   public String execute() throws Exception {

           boolean result = false;

           ClienteDao dao = null;

           try {
                   dao = new ClienteDao();
                   clientes = dao.carregaClientes();
                   result = true;
           }
           finally {
                   if (dao != null)
                           dao.close();
           }

           if (result)
```

```
                    return "success";

            return "error";
    }

    public List<Cliente> getClientes() {
        return this.clientes;
    }
}
```

A tela **ListaClientes.jsp** também precisa ser adaptada, para mostrar o campo **nome**. Porque agora a action não retorna uma lista de strings, mas uma lista de javabeans **Cliente**. Veja a diferença:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Lista de clientes</title>
</head>
<body>
   <h1>Lista de clientes</h1>
   <form action="listaClientes">
      <label for="filtro">Filtrar por:</label><br/>
      <input type="text" name="filtro"/>
      <input type="submit" value="Filtrar"/>
      <input type="submit" value="Novo"/>
      <br>
      <table border=1>
      <s:iterator value="clientes">
        <tr>
          <td>
                  <s:property value="nome"/>
                  </td>
                  <td>
                        <input type="submit" value="Editar"/>
                  </td>
                  <td>
                    <input type="submit" value="Excluir"/>
                  </td>
                </tr>
          </s:iterator>
          </table>
   </form>
</body>
</html>
```

15

Para fazer um teste melhor, adicione alguns clientes. Utilize o Workbench do MySql para inserir alguns clientes. Execute o script SQL abaixo para inserir clientes:

```
insert into MyDB.Cliente values(1,"João",1980,"Rua tal, 123", "", "12345-000",
"5555-5555");
insert into MyDB.Cliente values(2,"Maria",1980,"Rua 1, 123", "", "23456-000",
"5555-6666");
insert into MyDB.Cliente values(3,"José",1980,"Rua 2, 123", "", "34567-000", "5555-
7777");
```

Agora execute a aplicação web ex3Clientes para ver como está ficando. Se tudo estiver certo, o browser exibirá uma página com a seguinte aparência:

Que tal incluir mais alguns clientes?

```
insert into MyDB.Cliente values(4,"Pedro",1973,"Rua tal, 123", "", "12345-000",
"5555-5545");
insert into MyDB.Cliente values(5,"Clara",1984,"Rua 1, 143", "", "23456-000",
"5585-8966");
insert into MyDB.Cliente values(6,"Marcos",1990,"Rua 2, 123", "", "34567-000",
"5695-1234");
```

Veja novamente o que ocorreu. Viu que bacana? Os novos clientes já estão aparecendo na nossa aplicação!

16

5 - INCLUINDO, ALTERANDO E EXCLUINDO CLIENTES

Agora iremos implementar o principal: a manutenção da tabela de clientes. Até aqui aprendemos:

- **View**
  - Tela JSP
  - Html
  - Tags do struts

- **Controller**
  - Classes Actions

- Struts.xml

- **Model**
  - Javabeans
  - Classes DAO

Cada um destes componentes terá seus códigos afetados ou deverão ser criados:

- **Tela JSP:** botões Novo, Editar e Excluir
- **Classes Action:** criar actions para incluir, editar e excluir clientes
- **Classe DAO:** criar métodos para incluir, editar e excluir clientes

17

Vamos primeiro modificar a classe DAO, que deverá ganhar quatro novos métodos:

```
public void incluirCliente(Cliente c) throws SQLException {
                PreparedStatement stmt = null;

                try {
                        stmt = connection.prepareStatement("insert into
MyDB.Cliente (idCliente, nome, dataNascimento, endereco, complemento, cep,
telefone)
                        values (?,?,?,?,?,?,?)");

                        stmt.setInt(1,c.getId());
                        stmt.setString(2,c.getNome());
                        stmt.setInt(3,c.getDataNascimento());
                        stmt.setString(4, c.getEndereco());
                        stmt.setString(5, c.getComplemento());
                        stmt.setString(6, c.getCep());
                        stmt.setString(7, c.getTelefone());

                        stmt.execute();
                }
                finally {
                        if (stmt != null)
                                stmt.close();
                }
        }

        public void alterarCliente(Cliente c) throws SQLException {
                PreparedStatement stmt = null;

                try {
                        stmt = connection.prepareStatement("update MyDB.Cliente
set "
                                        + "nome = ?,"
                                        + "dataNascimento = ?, "
                                        + "endereco = ?, "
                                        + "complemento = ?, "
```

```
                                        + "cep = ?, "
                                        + "telefone = ? where idCliente = ?");

                        stmt.setString(1,c.getNome());
                        stmt.setInt(2,c.getDataNascimento());
                        stmt.setString(3, c.getEndereco());
                        stmt.setString(4, c.getComplemento());
                        stmt.setString(5, c.getCep());
                        stmt.setString(6, c.getTelefone());
                        stmt.setInt(7,c.getId());

                        stmt.execute();
                }
                finally {
                        if (stmt != null)
                                stmt.close();
                }
        }

        public void excluirCliente(Cliente c) throws SQLException {
                PreparedStatement stmt = null;

                try {
                        stmt = connection.prepareStatement("delete from
MyDB.Cliente where idCliente = ?");

                        stmt.setInt(1,c.getId());

                        stmt.execute();
                }
                finally {
                        if (stmt != null)
                                stmt.close();
                }
        }

        public Cliente carregarCliente(int id) throws SQLException {
                PreparedStatement stmt = null;
                ResultSet rs = null;

                Cliente result = null;

                try {
                        stmt = connection.prepareStatement("select * from
MyDB.Cliente where idCliente = ?");

                        stmt.setInt(1,id);

                        rs = stmt.executeQuery();

                        if (rs.next()) {
                                Cliente c = new Cliente();
                                c.setId(rs.getInt("idCliente"));
                                c.setNome(rs.getString("nome"));
```

```
                                c.setDataNascimento(rs.getInt("dataNascimento"));
                                c.setEndereco(rs.getString("endereco"));
                                c.setComplemento(rs.getString("complemento"));
                                c.setCep(rs.getString("cep"));
                                c.setTelefone(rs.getString("telefone"));
                                result = c;
                        }
                }
                finally {
                        if (rs != null)
                                rs.close();
                        if (stmt != null)
                                stmt.close();
                }

                return result;
        }
```

18

A tela ListaClientes.jsp terá que ser alterada para implementar os botões Novo, Editar e Excluir. Seu código ficará da seguinte forma:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Lista de clientes</title>
</head>
<body>
   <h1>Lista de clientes</h1>
   <form action="listaClientes">
      <label for="filtro">Filtrar por:</label><br/>
      <input type="text" name="filtro"/>
      <input type="submit" value="Filtrar"/>
      <button type="submit" name="comando" value='Novo'>Novo</button>
      <br>
      <table border=1>
      <s:iterator value="clientes">
        <tr>
          <td>
                  <s:property value="nome"/>
                  </td>
                  <td>
                    <button type="submit" name="comando"
                   value='Editar<s:property value="id"/>'>Editar</button>

                  </td>
                  <td>
```

```
                    <button type="submit" name="comando"
                    value='Excluir<s:property
value="id"/>'>Excluir</button>
                </td>
            </tr>
        </s:iterator>
        </table>
    </form>
</body>
</html>
```

Note que os botões passaram a utilizar a tag *button*. Isto porque usaremos o 'value' do button para passar o id do registro de cliente que queremos excluir ou alterar.

Repare também que os botões têm o 'name' definido como 'comando'. Isto significará o "comando" para o qual a action ListaClientes deverá redirecionar para tratar a requisição.

19

Para tratar estes comandos, o código fonte da ListaClientesAction ficou assim:

```
package clientesPack;

import java.util.List;

public class ListaClientesAction {
    private List<Cliente> clientes;
        private String comando = null;
        private String id = null;

    public String execute() throws Exception {

            boolean result = false;

            ClienteDao dao = null;

            if (comando != null) {
                        if (comando.equals("Novo")) {
                                return "novo";
                        }
                        else if (comando.indexOf("Editar") == 0) {
                                id = comando.substring(6);
                                return "editar";
                        }
                        else if (comando.indexOf("Excluir") == 0) {
                                id = comando.substring(7);
                                return "excluir";
                        }
                }

            try {
                        dao = new ClienteDao();
```

```
                    clientes = dao.carregaClientes();
                    result = true;
            }
            finally {
                    if (dao != null)
                            dao.close();
            }

            if (result)
                    return "success";

            return "error";
    }

    public String getId() {
                return id;
        }

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

    public List<Cliente> getClientes() {
       return this.clientes;
    }

    public void setFiltro(String filtro) {
            this.filtro = filtro;
    }

    public String getFiltro() {
            return filtro;
    }

    public void setComando(String comando) {
            this.comando = comando;
    }

    public String getComando() {
            return comando;
    }
}
```

Repare que a action ganhou 2 atributos: id e comando. Comando contém um identificador para indicar para qual action será redirecionado, id contém o valor do registro que deverá ser alterado ou excluído.

Os códigos de retorno do método execute são: **novo, editar, excluir, success e erro**. Através destes códigos de retorno, o redirecionamento é mapeado em struts.xml.

20

O struts.xml (já com as actions para os respectivos comandos) ficará assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
   <constant name="struts.devMode" value="true" />
   <package name="clientesPack" extends="struts-default">
      <action name="listaClientes" class="clientesPack.ListaClientesAction"
method="execute">
            <result name="success">/ListaClientes.jsp</result>
                        <result name="novo">/NovoCliente.jsp</result>
                        <result name="editar" type="redirectAction">
                              <param name="actionName">editarCliente</param>
                              <param name="location">/EditarCliente.jsp</param>
                              <param name="id">${id}</param>
                              <param name="comando">Carregar</param>
                        </result>
                        <result name="excluir" type="redirectAction">
                              <param name="actionName">excluirCliente</param>
                              <param name="location">/ExcluirCliente.jsp</param>
                              <param name="id">${id}</param>
                              <param name="comando">Carregar</param>
                        </result>
      </action>
      <action name="novoCliente" class="clientesPack.NovoClienteAction"
method="execute">
            <result name="success">/ListaClientes.jsp</result>
      </action>
      <action name="editarCliente" class="clientesPack.EditarClienteAction"
method="execute">
            <result name="carregado">/EditarCliente.jsp</result>
            <result name="success">/ListaClientes.jsp</result>
      </action>
      <action name="excluirCliente" class="clientesPack.ExcluirClienteAction"
method="execute">
                        <result name="carregado">/ExcluirCliente.jsp</result>
                        <result name="success">/ListaClientes.jsp</result>
      </action>
   </package>
</struts>
```

No primeiro código marcado, podemos ver o redirecionamento ser realizado a partir do “result”. Equivale a interpretar da seguinte forma:

- Se retorno de ListaClientesAction.execute for igual a “success”, voltar para tela ListaClientes.jsp
- Se retorno de ListaClientesAction.execute for igual a “novo”, redirecionar browser para NovoCliente.jsp
- Se retorno de ListaClientesAction.execute for igual a “editar”, redirecionar browser para action editarCliente, com a tela EditarCliente.jsp, levando como parâmetro o id igual ao id que está na action atual (ListaClientesAction)

- Se retorno de ListaClientesAction.execute for igual a “excluir”, redirecionar browser para action excluirCliente, com a tela ExcluirCliente.jsp, levando como parâmetro o id igual ao id que está na action atual (ListaClientesAction)

21

Repare também que as actions editarCliente e excluirCliente são ligeiramente diferentes: elas precisam carregar os dados do cliente para exibi-los na tela.

Para isto, existe as linhas na definição da ListaClientesAction:

```
<result name="editar" type="redirectAction">
                <param name="actionName">editarCliente</param>
                <param name="location">/EditarCliente.jsp</param>
                <param name="id">${id}</param>
                <param name="comando">Carregar</param>
            </result>
            <result name="excluir" type="redirectAction">
                <param name="actionName">excluirCliente</param>
                <param name="location">/ExcluirCliente.jsp</param>
                <param name="id">${id}</param>
                <param name="comando">Carregar</param>
            </result>
```

Isto instrui as actions editarCliente e excluirCliente a terem seu atributo “comando” pré-carregado com a string “Carregar”. Desta forma, ao chamar EditarClienteAction.execute, por exemplo, poderemos checar o valor do atributo “comando” para sabermos se devemos carregar ou salvar o cliente. Podemos ver melhor isto no código das actions editarCliente e excluirCliente. Clique nos links a seguir.

EditarClienteAction.java

ExcluirClienteAction.java

EditarCliente.jsp

ExcluirCliente.jsp

Agora basta lançar a aplicação e verificar se está tudo funcionando como previsto! Teste algumas entradas de dados e edições. Busque verificar se o programa aceita qualquer tipo de entrada e tente corrigir eventuais erros. É um excelente ponto de partida para qualquer aplicação web, portanto pode alterá-la à vontade!

EditarClienteAction.java

```
package clientesPack;

public class EditarClienteAction {
      private int id;
      private String nome;
```

```
	private int dataNascimento;
	private String endereco;
	private String complemento;
	private String cep;
	private String telefone;

	private String comando = null;

	public String execute() throws Exception {
		boolean result = false;
		ClienteDao dao = null;

		if (comando == null) {
			return "success";
		}
		else if (comando.equals("Cancelar")) {
			return "success";
		}
		else if (comando.equals("Carregar")) {
			try {
			   dao = new ClienteDao();
			   Cliente c = dao.carregarCliente(id);

			   setId(c.getId());
			   setNome(c.getNome());
			   setDataNascimento(c.getDataNascimento());
			   setEndereco(c.getEndereco());
			   setComplemento(c.getComplemento());
			   setCep(c.getCep());
			   setTelefone(c.getTelefone());

			   result = true;
		    }
		    finally {
			   if (dao != null)
				  dao.close();
		    }

			if (result)
			      return "carregado";

			return "error";
		}
		else if (comando.equals("Salvar")) {
			Cliente c = new Cliente();
		    c.setId(id);
		    c.setNome(nome);
		    c.setDataNascimento(dataNascimento);
		    c.setEndereco(endereco);
		    c.setComplemento(complemento);
		    c.setCep(cep);
		    c.setTelefone(telefone);

		    try {
```

```
                dao = new ClienteDao();
                dao.alterarCliente(c);
                result = true;
            }
            finally {
                if (dao != null)
                    dao.close();
            }

            if (result)
                return "success";

            return "error";
        }

        return "error";
    }

    public int getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

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

    public int getDataNascimento() {
        return dataNascimento;
    }

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

    public String getEndereco() {
        return endereco;
    }

    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }

    public String getComplemento() {
        return complemento;
    }
```

```
	public void setComplemento(String complemento) {
		this.complemento = complemento;
	}

	public String getCep() {
		return cep;
	}

	public void setCep(String cep) {
		this.cep = cep;
	}

	public String getTelefone() {
		return telefone;
	}

	public void setTelefone(String telefone) {
		this.telefone = telefone;
	}

	public String getComando() {
		return comando;
	}

	public void setComando(String comando) {
		this.comando = comando;
	}
}
```

ExcluirClienteAction.java

```
package clientesPack;

public class ExcluirClienteAction {
	private int id;
	private String nome;
	private int dataNascimento;
	private String endereco;
	private String complemento;
	private String cep;
	private String telefone;

	private String comando = null;

	public String execute() throws Exception {
		boolean result = false;
		ClienteDao dao = null;

		if (comando == null) {
			return "success";
		}
```

```
        else if (comando.equals("Cancelar")) {
              return "success";
        }
        else if (comando.equals("Carregar")) {
              try {
                 dao = new ClienteDao();
                 Cliente c = dao.carregarCliente(id);

                 setId(c.getId());
                 setNome(c.getNome());
                 setDataNascimento(c.getDataNascimento());
                 setEndereco(c.getEndereco());
                 setComplemento(c.getComplemento());
                 setCep(c.getCep());
                 setTelefone(c.getTelefone());

                 result = true;
            }
            finally {
                 if (dao != null)
                       dao.close();
            }

              if (result)
                    return "carregado";

              return "error";
        }
        else if (comando.equals("Excluir")) {
              Cliente c = new Cliente();
              c.setId(id);

            try {
                 dao = new ClienteDao();
                 dao.excluirCliente(c);
                 result = true;
            }
            finally {
                 if (dao != null)
                       dao.close();
            }

            if (result)
                 return "success";

            return "error";
        }

        return "error";
  }

  public int getId() {
        return id;
  }
```

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

public String getNome() {
	return nome;
}

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

public int getDataNascimento() {
	return dataNascimento;
}

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

public String getEndereco() {
	return endereco;
}

public void setEndereco(String endereco) {
	this.endereco = endereco;
}

public String getComplemento() {
	return complemento;
}

public void setComplemento(String complemento) {
	this.complemento = complemento;
}

public String getCep() {
	return cep;
}

public void setCep(String cep) {
	this.cep = cep;
}

public String getTelefone() {
	return telefone;
}

public void setTelefone(String telefone) {
	this.telefone = telefone;
}

public String getComando() {
```

```
            return comando;
      }

      public void setComando(String comando) {
            this.comando = comando;
      }
}
```

EditarCliente.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Editar Cliente</title>
</head>
<body>
      <h1>Editar Clientes</h1>
   <form action="editarCliente">
       <input type="hidden" name="id" value='<s:property value="id"/>'/><br/>
       <label for="id">ID:</label><br/>
       <input type="text" name="idVisivel" disabled="disabled" value='<s:property
value="id"/>'/><br/>
       <br/>
       <label for="nome">Nome:</label><br/>
       <input type="text" name="nome" value='<s:property value="nome"/>'/><br/>
       <br/>
       <label for="dataNascimento">Data de nascimento:</label><br/>
       <input type="text" name="dataNascimento" value='<s:property
value="dataNascimento"/>'/><br/>
       <br/>
       <label for="endereco">Endereço:</label><br/>
       <input type="text" name="endereco" value='<s:property
value="endereco"/>'/><br/>
       <br/>
       <label for="complemento">Complemento:</label><br/>
       <input type="text" name="complemento" value='<s:property
value="complemento"/>'/><br/>
       <br/>
       <label for="cep">Cep:</label><br/>
       <input type="text" name="cep" value='<s:property value="cep"/>'/><br/>
       <br/>
       <label for="telefone">Telefone:</label><br/>
       <input type="text" name="telefone" value='<s:property
value="telefone"/>'/><br/>
       <br/>
       <input type="submit" name="comando" value="Salvar"/>
       <input type="submit" name="comando" value="Cancelar"/>
```

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

ExcluirCliente.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Excluir Cliente</title>
</head>
<body>
      <h1>Excluir Cliente</h1>
   <form action="excluirCliente">
       <input type="hidden" name="id" value='<s:property value="id"/>'/><br/>
       <label for="id">ID:</label><br/>
       <input type="text" name="idVisivel" disabled="disabled" value='<s:property
value="id"/>'/><br/>
       <br/>
       <label for="nome">Nome:</label><br/>
       <input type="text" name="nome" value='<s:property value="nome"/>'/><br/>
       <br/>
       <label for="dataNascimento">Data de nascimento:</label><br/>
       <input type="text" name="dataNascimento" value='<s:property
value="dataNascimento"/>'/><br/>
       <br/>
       <label for="endereco">Endereço:</label><br/>
       <input type="text" name="endereco" value='<s:property
value="endereco"/>'/><br/>
       <br/>
       <label for="complemento">Complemento:</label><br/>
       <input type="text" name="complemento" value='<s:property
value="complemento"/>'/><br/>
       <br/>
       <label for="cep">Cep:</label><br/>
       <input type="text" name="cep" value='<s:property value="cep"/>'/><br/>
       <br/>
       <label for="telefone">Telefone:</label><br/>
       <input type="text" name="telefone" value='<s:property
value="telefone"/>'/><br/>
       <br/>
       Tem certeza que deseja excluir este cliente?<br/>
       <input type="submit" name="comando" value="Excluir"/>
       <input type="submit" name="comando" value="Cancelar"/>
   </form>
</body>
```

```
</html>
```

22

RESUMO

O objetivo desse módulo foi apresentar o desenvolvimento completo de uma aplicação MVC usando o framework Struts. O struts é apenas uma das tecnologias que implementam o MVC. Existem outras com maior ou menor complexidade de utilização. Para cada uma das camadas do MVC existem diferentes tecnologias. Estudam-se cada vez mais técnicas que minimizem o trabalho de manutenção do código-fonte.

Por ser separada, a camada View pode ser feita ou melhorada por um especialista em interface gráfica (tal como um Web Designer). A camada Controller por um desenvolvedor java. Já a camada Model pode ter a participação de um DBA ou modelador de base de dados.

UNIDADE 4 – TÓPICOS AVANÇADOS EM PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 3 – UML

01

1 - UML E DIAGRAMAS

UML (*Unified Modeling Language*) é uma linguagem gráfica para representação de um sistema, de forma padronizada, com tipos de diagramas próprios para várias dimensões distintas deste sistema, tais como: comportamento do sistema, classes de dados, casos de uso etc.

Assim como um engenheiro civil planeja a construção de uma casa fazendo plantas (estrutural, elétrica, hidráulica, etc.), o projetista de*software* utiliza diferentes diagramas UML para visualizar e planejar os diferentes aspectos de um *software*.

O UML é útil também para comunicar a uma equipe de desenvolvedores como um *software* será estruturado e implementado.

Podemos utilizar o UML também para descrever cenários, partes de um *software*, um detalhe de funcionamento ou utilização, como se dá interação deste *software* com o usuário etc. A UML permite que façamos isto de um jeito formal (sem descrição textual sujeita a ambiguidades do idioma) numa linguagem gráfica e padronizada.

Os principais diagramas previstos na UML 2.0 dividem-se em dois grandes grupos, que são:

Diagramas estruturais	Diagramas comportamentais

• Diagrama de classes • Diagrama de objetos • Diagrama de componentes • Diagrama de instalação/implantação • Diagrama de pacotes	• Diagrama de casos de uso • Diagrama de estados • Diagrama de componentes • Diagrama de atividade • Diagrama de sequência

Veremos os subtipos de cada um deles a seguir.

Diagramas estruturais
Como o próprio nome sugere, abordam o aspecto estrutural tanto do ponto de vista do sistema quanto das classes. Servem para especificar, visualizar, construir e documentar os aspectos estáticos de um sistema, ou seja, a representação de seu arcabouço e estruturas relativamente estáveis.

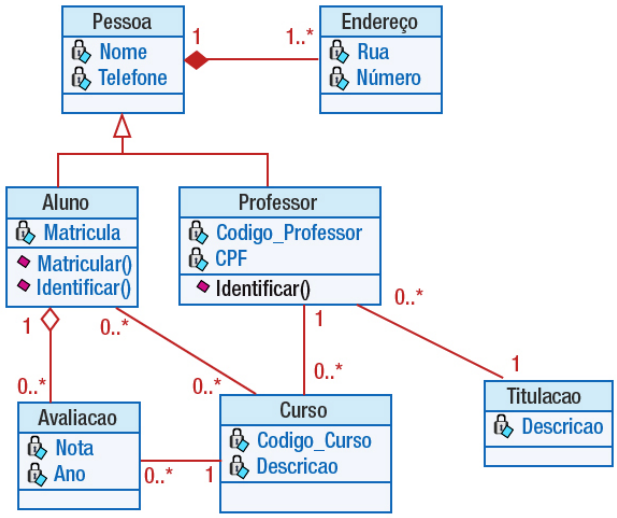
Diagramas comportamentais
Visam descrever o sistema computacional modelado quando em execução, isto é, como a modelagem dinâmica do sistema. Servem para especificar, visualizar, construir e documentar os aspectos dinâmicos de um sistema que é a representação das partes que sofrem modificações constantes, como por exemplo, o fluxo de mensagens ao longo do tempo ou a movimentação física de componentes em uma rede.

02

1.1 - Diagramas estruturais

- **Diagrama de classes**

É a representação da estrutura das classes e seus relacionamentos (herança, associação etc.).

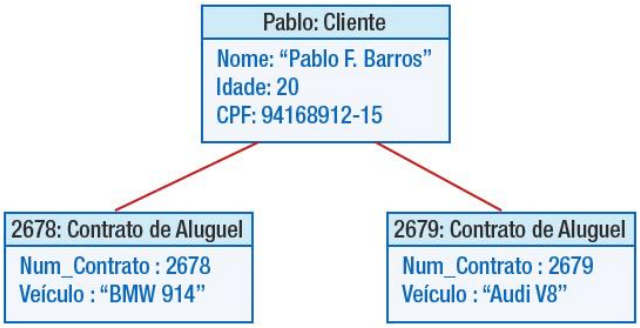


03

- **Diagrama de objetos**

Representa uma "foto" de objetos (instância das classes) para exibir uma determinada estrutura instanciada dos objetos em um sistema num dado momento.

É útil para exemplificar como as classes são criadas e usadas dinamicamente.

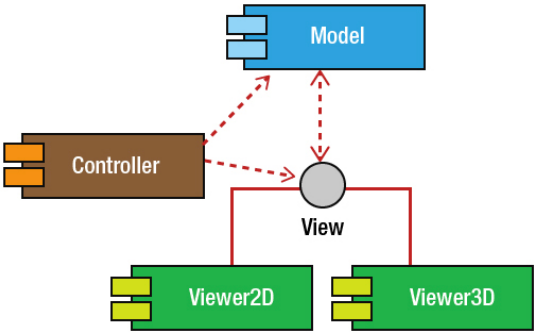


04

- **Diagrama de components**

Representa como as classes são agrupadas funcionalmente no código de *software*. Componentes são agrupamentos de classes que são utilizadas para implementar uma determinada função no *software*.

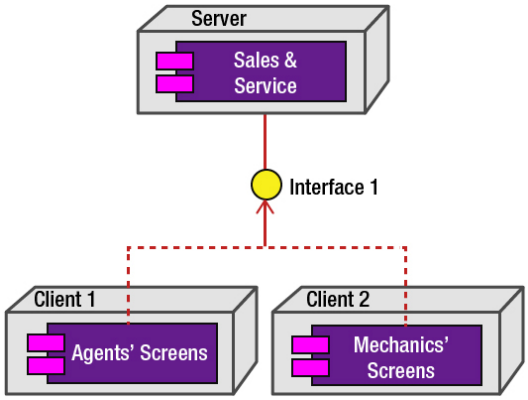
Por exemplo: um componente pode estar relacionado a um conjunto de bibliotecas, com classes que implementam visualização 3D de um sistema CAD. Os relacionamentos dos componentes têm a ver com a comunicação realizada entre eles.



05

- **Diagrama de instalação/implantação:**

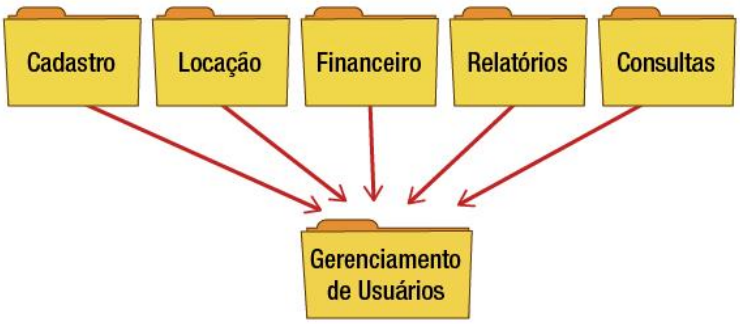
Apresenta os componentes de *hardware* e *software* e sua interação com outros elementos necessários ao funcionamento do sistema, como por exemplo: banco de dados, processadores, servidor externo etc.



06

- **Diagrama de pacotes**

Descreve agrupamentos lógicos de classes de um sistema e a dependência entre eles. Diferentemente do diagrama de componente, o diagrama de pacotes dá ênfase na dependência.



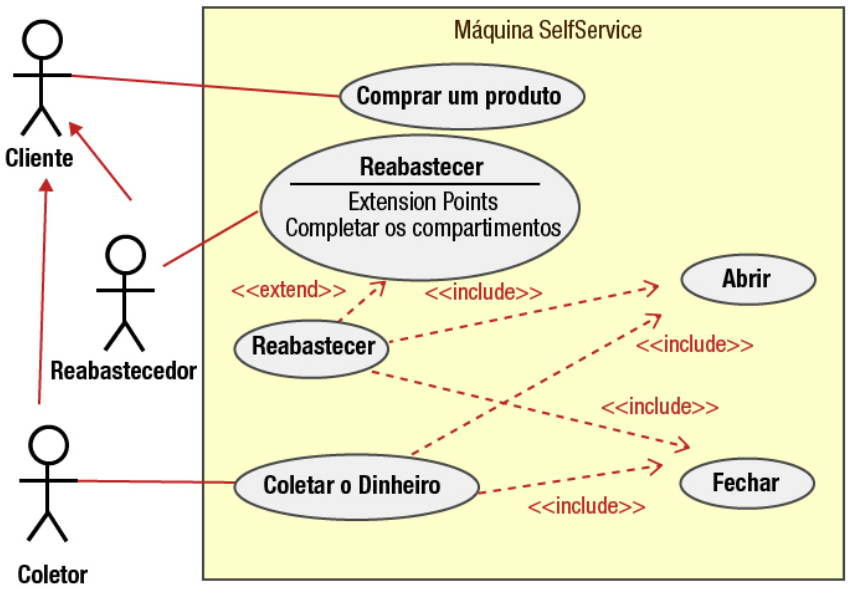
07

1.2 - Diagramas comportamentais

- **Diagrama de casos de uso**

Descreve as funções do sistema, e sua interação com os atores do mundo externo (usuários, instituições, outros sistemas etc.). Um caso de uso é uma narrativa sobre a sequência de eventos de uma interação entre um ator e o sistema.

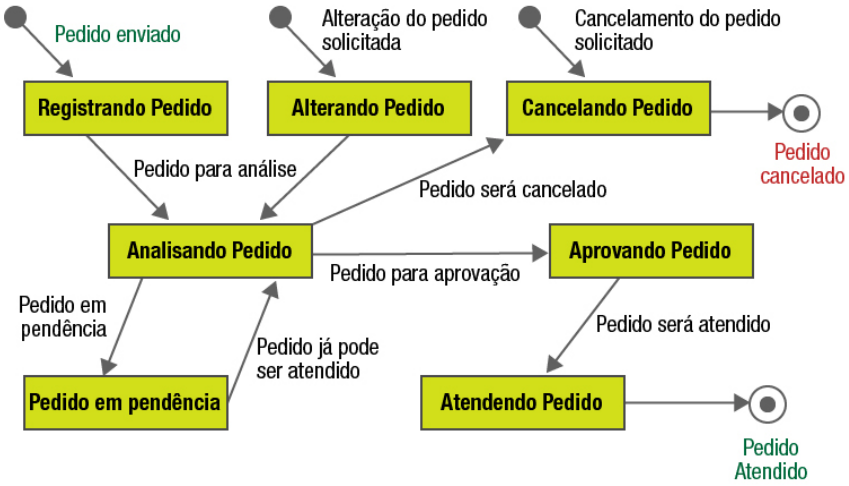
Os casos de uso podem estar diretamente relacionados com as opções de menu de um sistema, por exemplo.



08

- **Diagrama de estados**

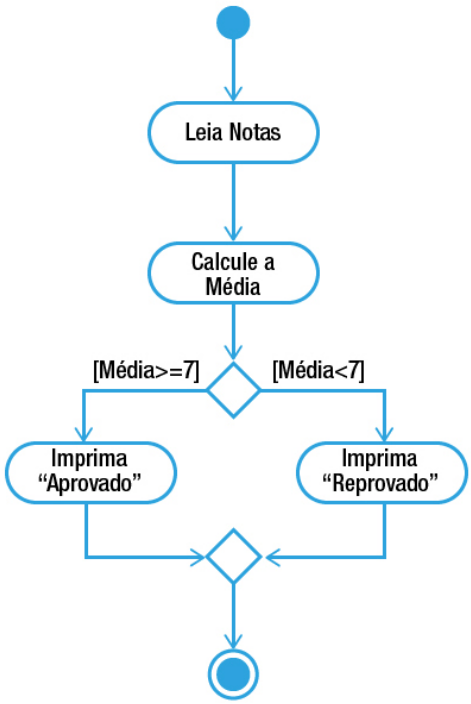
Descreve os estados que um objeto pode estar e a sequência prevista mediante condições e ações.



09

- **Diagrama de atividade**

Equivale a um fluxograma, demonstrando o controle e a dinâmica de um sistema (ou parte dele), que ocorre ao longo de um determinado processamento..

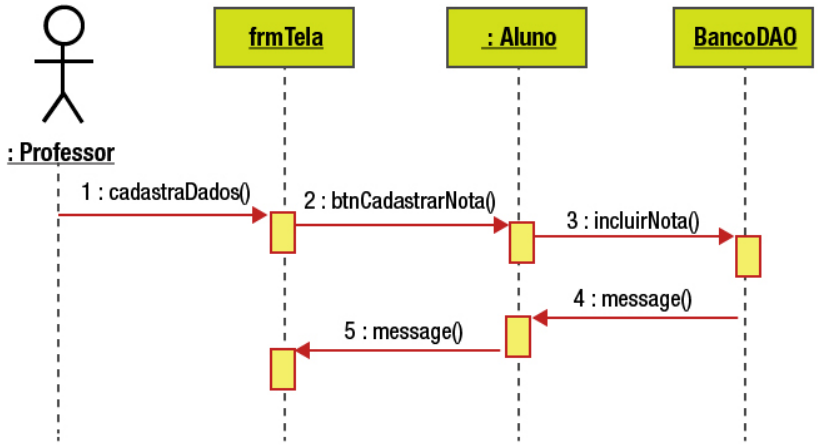


10

- **Diagrama de sequência**

Representa a sequência de processos, através da demonstração das mensagens que são trocadas entre objetos, em determinada atividade de um sistema.

Podemos interpretar “mensagens” como chamadas de métodos de classe de determinado objeto, por exemplo. O diagrama de sequência aborda aspectos como execução de tal método ao longo do tempo, e a interatividade entre objetos, num determinado cenário.



11

Que tal criar seu próprio diagrama?
Para você criar seus diagramas, utilize algumas das ferramentas UML disponíveis na internet:

- http://www.umldesigner.org/download/
- http://sourceforge.net/projects/argouml/

- Outras: http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

12

2 - Usando o UML para planejar um sistema de cadastro de clientes

Iremos agora criar um sistema para cadastrar clientes de um estabelecimento comercial, que permita arquivar os dados destes clientes, e que possibilite gerar relatórios e envios de e-mail para anunciar promoções e outras informações.

A UML não define uma sequência pela qual devemos desenhar os diagramas. Ela é apenas uma linguagem de diagrama. Porém, se estamos criando um sistema do zero, é interessante começar pelo diagrama de casos de uso, pois permite definir o tamanho (escopo) do sistema que queremos fazer, uma vez que nos força a dizer quais funções o sistema terá.

2.1 Casos de Uso

Para construir este diagrama precisamos perguntar:

1. Quais são as funcionalidades do sistema? (**casos de uso**)
2. Quem irá interagir com o sistema? (**atores**)
3. Como será a interação destes itens? (**relacionamentos**)

⇒ Atores

Podemos elencar os seguintes atores:

- Usuário do sistema
- Cliente
- Serviço de e-mail

Os atores são representados como um boneco.

13

⇒ Casos de uso

Se quisermos nos aproximar mais do sistema físico, podemos criar os casos de uso já pensando nas opções de menu que ele irá oferecer. Podemos também criar casos de uso de mais alto nível, sem estar totalmente associado a uma opção. Neste exemplo, utilizaremos a primeira opção.

Diante da descrição de nossa intenção, podemos visualizar a função "**Cadastrar clientes**". Quem poderá executar esta função? O usuário do sistema? O serviço de e-mail? Neste momento, já estamos identificando o tamanho de nosso sistema, porém o papel do UML não é ditar as regras de projeto, ou de definir o escopo, mas simplesmente visualizar o que se quer.

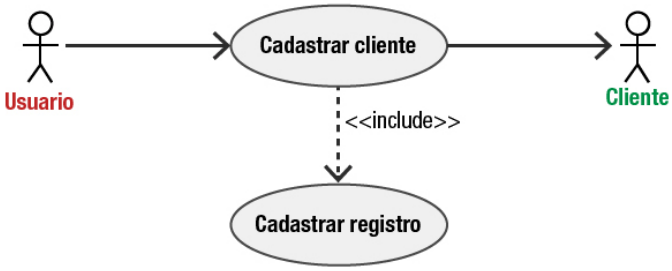
Então, para simplificar, vamos criar um sistema simples onde o usuário cadastra clientes. Então podemos desenhar este caso de uso da seguinte forma:

A linha indica o relacionamento "Associação". Em diagramas de caso de uso podemos ter os seguintes relacionamentos: ***include*** ou ***extend***, conforme veremos adiante.

14

Para demonstrar isto, podemos criar um segundo caso de uso que é incluído pelo "Cadastrar cliente", que é o "Cadastrar registro".

Note que estes relacionamentos entre casos de uso são bem úteis no sentido de evidenciar partes do sistema (outros casos de uso) que podem vir a serem genéricos ou reutilizáveis.



Por fim, ligamos os atores aos casos de uso no qual eles se interagem.

Usando o ArgoUML, desenhe um diagrama de casos de uso para um sistema de compras on-line, onde clientes podem acessar o site, escolher produtos, e finalizar a compra, colocando seus dados de cartão de crédito.
O sistema deve interagir com as empresas de cartão, a fim de validar os dados.

15

⇒ **Associação**

É a seta que liga um ator a um caso de uso, ou casos de uso com outros casos de uso, demonstrando a sequência do processo.

⇒ **Generalização**

É uma seta aberta, que denota herança entre dois casos de uso ou dois atores. Por exemplo, um "funcionário" poderia ser um tipo de ator especializado de usuário. Esta especialização seria representada da seguinte maneira:

16

⇒ Dependência

É uma seta tracejada que denota que um caso de uso depende de outro para o seu funcionamento completo. Pode ser de dois tipos (*stereotypes*):

1. Include

Denota que um caso de uso utiliza outro em sua composição. O caso utilizado é genérico, e evitar que seja repetido várias vezes no modelo, utilizamos esta dependência.

2. Extends

Denota que um caso de uso é uma variação (extensão) de um caso de uso mais básico.

a) Usando uma ferramenta de designer UML, desenhe um diagrama de casos de uso para um sistema de compras on-line, onde clientes podem acessar o site, escolher produtos, e finalizar a compra, colocando seus dados de cartão de crédito. O sistema deve interagir com as empresas de cartão, a fim de validar os dados.

b) Revendo o sistema de cadastro de clientes já realizado neste curso, faça um diagrama de casos de uso daquele sistema. Considere cada botão da tela (novo, editar, excluir e filtrar) como sendo um caso de uso.

17

2.2 Diagrama de estados/atividade

Apesar de serem semelhantes, **diagramas de estados** servem para modelar o comportamento discreto entre estados finitos, enquanto que**diagramas de atividades** modelam aspectos comportamentais de um processo.

Podemos dizer que o diagrama de estados modela processos de mais baixo nível (como por exemplo, o estado de uma porta serial com relação aos bits recebidos) e diagrama de atividades modela processos tal qual um fluxograma ou *workflow*.

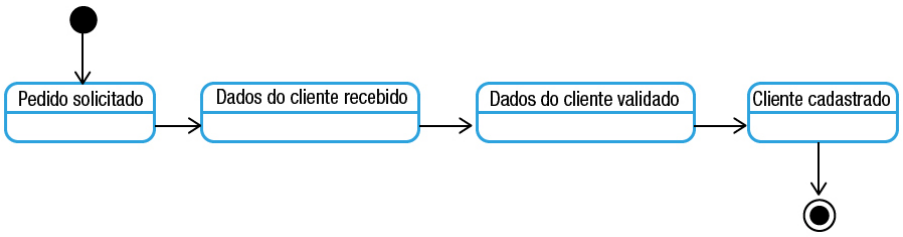
Utilizando o cenário do sistema de cadastro de clientes, vamos criar o diagrama de atividade. Para isto, procuramos enumerar e sequenciar quais passos necessários para executar um caso de uso, por exemplo.

Criaremos então o diagrama de estados para o caso de uso “Cadastrar cliente”. Podemos focar o aspecto da interação humana como também aspectos internos do sistema, e desmembrar da seguinte forma:

Cadastrar cliente

1. Cliente solicita cadastramento
2. Usuário informa dados do cliente no sistema
3. Sistema checa se dados são válidos
4. Sistema confirma o cadastramento

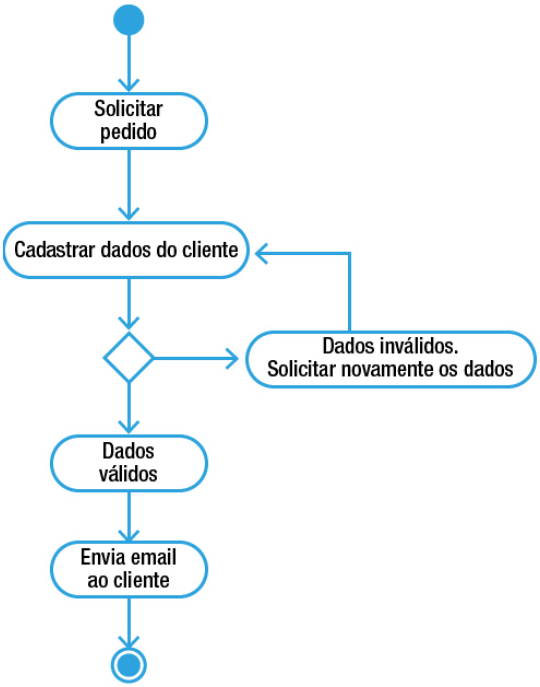
Considerando que os itens acima provocam a mudança de estados, no diagrama de estados eles são representados por setas, que significam **Transições**.



18

Enquanto diagramas de estados focam quais transições são possíveis a partir de um estado de um objeto, os diagramas de atividade podem conter **decisões** que interferem no fluxo do processo.

Abaixo, vemos um exemplo de como um diagrama de atividades poderia ser construído para o caso de cadastrar clientes:



Note também que, enquanto no diagrama de estados cada balão denota um estado de um objeto, no diagrama de atividades o balão tem conceito mais amplo, e pode denotar também uma atividade a ser realizada.

19

a) Usando uma ferramenta de designer UML crie um diagrama de estados para um sistema que controla uma linha de produção. Procure vislumbrar os estados de um produto, desde a recepção da matéria-prima até sua entrega final.

b) Usando uma ferramenta de designer UML crie um diagrama de atividade para um sistema que controla um fluxo de documento em uma empresa que realiza publicações em jornal. Considere que o documento possa ter os seguintes estados: em autoria, em revisão, publicado ou arquivado. Considere também os fluxos onde o documento em revisão volta para autoria, para o autor refazer seu conteúdo, e também o caso de documentos que, ao serem cancelados, vão para o arquivo.

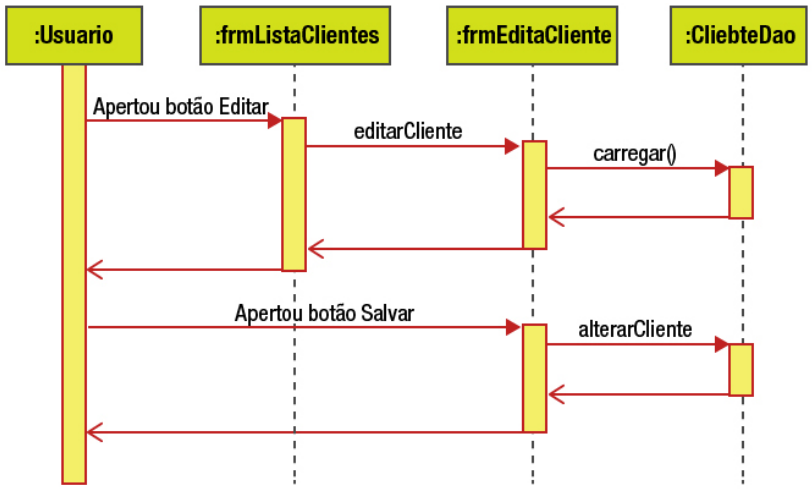
20

2.2 Diagrama de sequência

Diferentemente dos diagramas de caso de uso, atividade e estados, diagramas de sequência estão mais próximos da implementação propriamente dita. Eles denotam as mensagens trocadas em objetos,

classes ou atores ao longo do tempo, em uma determinada interação que desejamos descrever com o diagrama. É útil para descrever ciclos de vida de objetos e a sequência das mensagens e suas repostas no tempo (síncrono ou assíncrono).

Os objetos são representados com “raias” da seguinte forma:



No diagrama acima, os objetos frmListaClientes e frmEditaCliente são de classes de implementação de um formulário na web. A objeto ClienteDao corresponde a uma classe que realiza a leitura e salvamento de dados de um cliente no banco de dados.

As setas denotam as mensagens (no caso, chamada de action, botão ou método) trocadas entre os objetos. Todas elas são síncronas, porque para cada mensagem o emissor fica bloqueado até que se cheque sua resposta.

21

Elementos que compõem um diagrama de sequência

Os seguintes elementos podem ser utilizados:

1. Atores
2. Objetos, multiobjetos e classes, e suas linhas de vida
3. Mensagens (síncronas ou assíncronas)
4. Criação e destruição dos objetos

⇒ Atores

Os atores têm o mesmo significado que os atores no caso de uso e também enviam mensagens para o sistema. As mensagens podem ser eventos de uma tela, tal como “apertou um botão”, “clicou numa opção de menu”, “fez uma requisição ao webservice” etc. Também são representados por um boneco.

⇒ Objetos e linha de vida

Os objetos são representados por caixas e raias, da seguinte forma:

A linha tracejada representa a **linha de vida** do objeto.

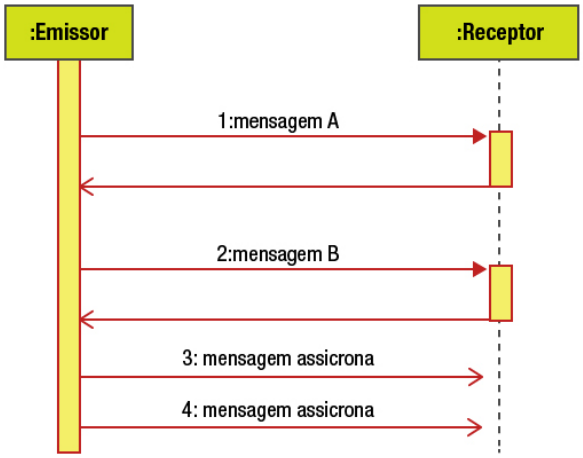
A notação para o nome do objeto é "**Nome : Classe**", onde o nome pode ser vazio (nas linguagens de programação geralmente não damos nome a objetos, somente às classes).

22

⇒ **Mensagens**

Mensagens são representadas por **setas**. A seta aponta para o objeto que recebe a mensagem.

Mensagens também podem ser consideradas **eventos**. O texto que descreve uma mensagem pode ser simplesmente um texto ou ter uma notação diferenciada descrevendo tipo de sincronização, condição, sequência, parâmetros, tipos de retorno etc. Por exemplo:

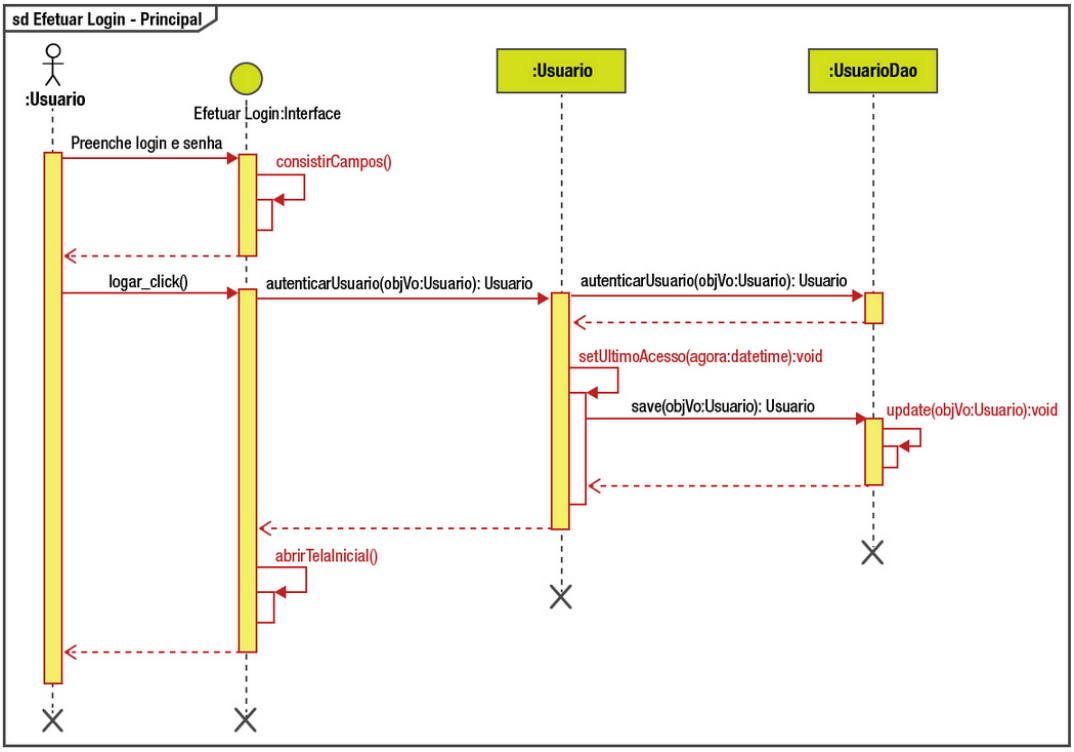


No caso, as mensagens estão com a sequência em que serão executadas (1, 2, 3 e 4). Note que mensagens assíncronas não possuem mensagens de retorno.

23

⇒ **Criação e destruição de objetos**

O exemplo abaixo mostra a destruição dos objetos (X), ao final de sua utilização.



Os retângulos sobre a linha de vida de cada objeto denotam **atividade** naquele objeto. A sobreposição de retângulos num mesmo objeto (como é o caso do objeto *:Usuario* denota que este objeto chama um método de sua própria classe (mensagem para si mesmo), como podemos ver no exemplo a chamada para o método **setUltimoAcesso**.

Cabe ressaltar que nem todas as ferramentas UML disponíveis possuem as mesmas notações. O padrão UML é amplo e permite adaptações e especializações com recursos para extensão do modelo, tal como *stereotypes*, por exemplo.

24

3 - USANDO OS DIAGRAMAS ESTRUTURAIS

3.1 Diagrama de Classes

Para quem desenvolve em linguagens orientadas a objeto, o diagrama de classes é o que mais se aproxima dos códigos fonte destas linguagens. Uma classe pode ser diretamente espelhada em seu código.

Basicamente as classes são constituídas de **atributos** e **métodos**. No UML, entretanto, as classes possuem outros relacionamentos além da herança: as associações, que podem ser utilizadas para descrever relacionamentos de agregação e composição.

Diagramas de classes são bem úteis para fazer engenharia reversa de sistemas. Existem analisadores que conseguem gerar os diagramas diretamente do código fonte. Existem também ambientes de

programação que permitem que se gere código fonte a partir de diagramas desenhados. Além disto, estes ambientes também mantém o código integrado ao diagrama, de tal forma que se houver uma alteração no código, o diagrama é atualizado, e vice-versa.

Para muitos desenvolvedores, o diagrama de classes é o ponto de partida no desenvolvimento de um novo sistema. O diagrama de classes pode também ser utilizado para projeto de diagramas de entidade-relacionamento.

Abordaremos aqui os elementos que compõem um diagrama de classes e como se espelha no código fonte.

25

⇒ **Classe**

É um elemento que representa um conjunto de objetos. Ela é formada por atributos, métodos e relacionamento com outras classes.

⇒ **Atributos**

Define as características da classe. São "campos" e para cada instância (objeto) de uma tal classe, possuem um estado (valor). Eles podem ter diferentes visibilidades, que, já conhecemos na linguagem java. No UML utilizamos os símbolos (+) (-) (#) e (~):

- **(+) Pública**: outras classes tem acesso ao atributo. No java utilizamos **public**.
- **(-) Privada**: somente a própria classe tem acesso ao atributo. No java utilizamos **private**.
- **(#) Protegida**: o atributo pode ser acessado pela própria classe e por todas as classes filhas

⇒ **Operações (ou métodos)**

Define o comportamento da classe. Também possuem visibilidade idêntica a dos atributos, entretanto, não se referem a dados, mas sim, a procedimentos, funções ou métodos. A definição da operação é composta pelo nome, valor de retorno e parâmetros. Também chamamos este conjunto (nome, retorno e parâmetros) de **assinatura do método**.

26

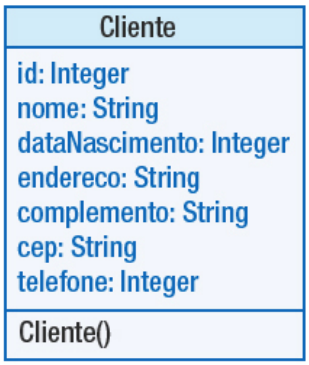
Lembram-se da classe **Cliente.java** vista anteriormente em MVC? Ela ficaria assim em UML:

Java

```
public class Cliente {
  private int id;
  private String nome;
  private int dataNascimento;
  private String endereco;
  private String complemento;
```

```
    private String cep;
    private String telefone;

    public Cliente() {
    }
}
```

UML



Algumas ferramentas UML mostram os símbolos (+), (-) e (#) ao lado de cada atributo ou método. Outras apresentam pequenos ícones.

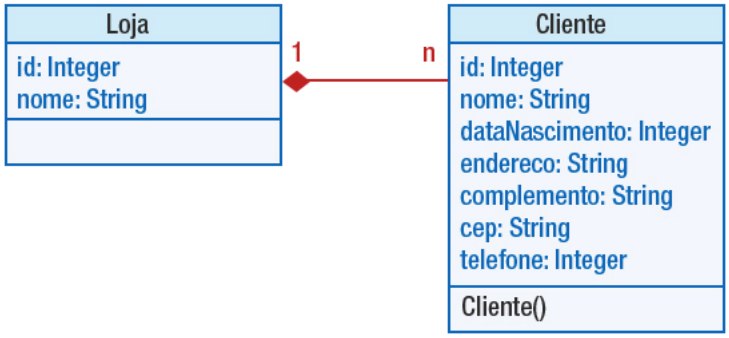
27

⇒ **Relacionamentos (Associação)**

Os relacionamentos são representados por linhas, ligando duas classes. Eles podem ser conceituais ou físicos (que possuem um espelhamento em código fonte). Os principais tipos de associações são:

• **Agregação**

Agregação é o relacionamento todo-parte, onde a parte pode existir independentemente se o todo existir. Por exemplo:



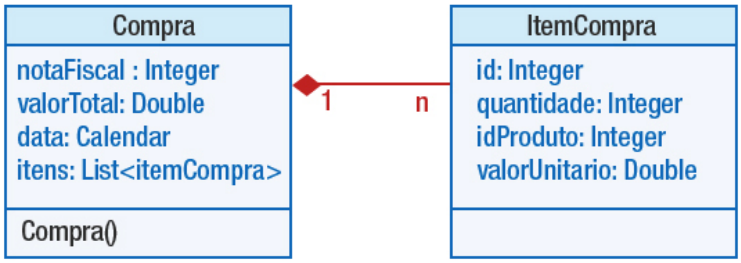
O cliente pode existir, mesmo se não estiver associado a uma loja. Ele pode comprar pela internet, e não estar relacionado a loja física...

Este relacionamento pode ser espelhado no java como um atributo **Lista** na classe Loja, que contem os clientes que compram naquela loja.

28

- **Composição**

Neste caso, a composição estabelece que a classe “todo” só faz sentido se as classes “parte” existirem e vice-versa. Por exemplo:

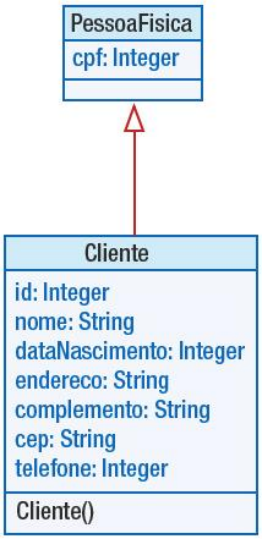


Percebemos claramente que não existe compra sem itens, e vice-versa. Note que o relacionamento é implementado com uma lista de itens na classe **Compra**.

29

- **Especialização (Herança)**

Tem o mesmo significado da implementação da herança do java. A classe filha é uma especialização da classe pai. Todos os atributos e métodos da classe pai são copiados para a classe filha, entretanto a classe filha não tem acesso aos atributos e métodos **private** da classe pai.

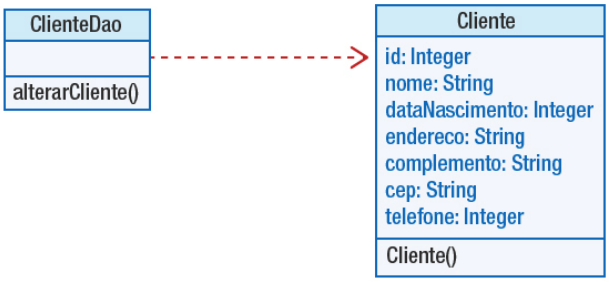


30

- **Dependência**

Serve para indicar que uma classe utiliza os atributos ou métodos de outra classe. Isto significa dizer que, se a classe utilizada sofrer uma modificação, a classe que a utiliza também terá que sofrer uma modificação. É um relacionamento muito útil para identificar o que será afetado por determinada modificação, em determinada parte de um *software*.

A dependência pode ser mapeada no java de muitas formas. Se num método de uma classe houver uma utilização de outra classe, a primeira já dependerá da segunda.



31

⇒ **Interfaces**

Interfaces podem ser consideradas um tipo especial (estereótipo) de classe. Entretanto, interfaces não possuem atributos, somente métodos. Interfaces também não possuem implementação, elas dependem de uma classe que as implementam. Em java, uma interface e a classe que a implementa é feita da seguinte forma:

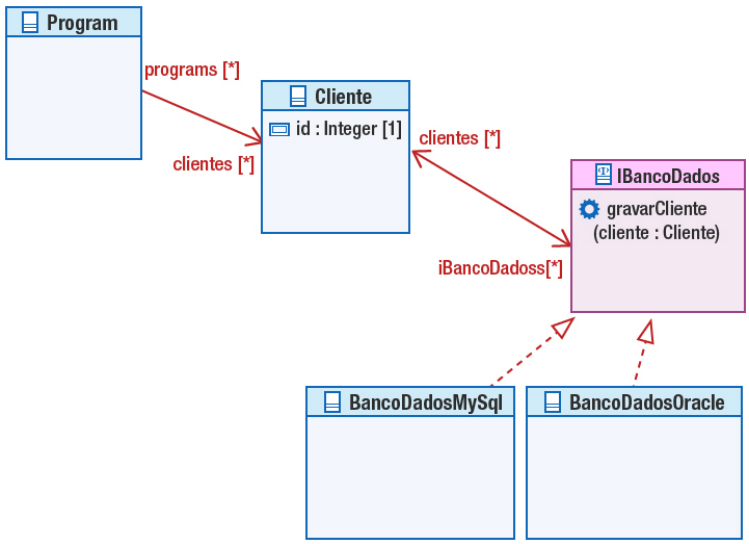
```
public interface IObjetoPersistente {
        public void grava(IObjetoPersistente obj);
        public IObjetoPersistente le();
}

public class Cliente implements IObjetoPersistente {

        @Override
        public void grava(IObjetoPersistente obj) {
                ...
        }

        @Override
        public IObjetoPersistente le() {
                ...
        }
}
```

Note que não há implementação na interface **IObjetoPersistente**. É a classe **Cliente** que implementa a interface **IObjetoPersistente**, com suas implementações de **grava** e **le**.

Interfaces são importantes para se criar isolamento entre classes, a fim de permitir baixo acoplamento (veremos mais sobre isto no próximo módulo).

32

A seguir, um exemplo prático de utilização de interface. Observe a interface **IBancoDados** como é visualizada em UML:



Cliente é uma classe que utiliza a interface **IBancoDados** (relacionamento associação), pois tem um método que faz referência a um método da interface. Já as classes **BancoDadosMySql** e **BancoDadosOracle** são classes que implementam os métodos da interface **IBancoDados**.

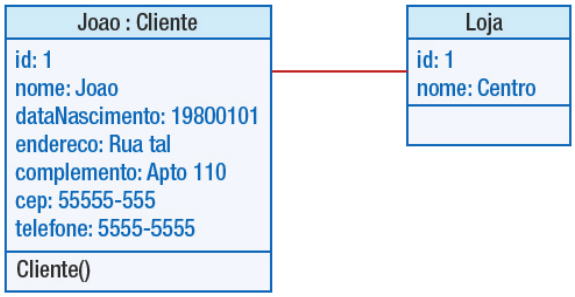
33

3.2 Diagrama de objetos

São semelhantes ao diagrama de classes, porém, com objetos, ou seja, uma classe com atributos "preenchidos" (instância). Enquanto o diagrama de classes descreve um modelo, o diagrama de objetos descreve uma "foto" de como instâncias de classes estão organizadas num dado momento.

Sua notação é similar ao diagrama de classes, contendo os mesmos elementos: associação, dependência, atributos e métodos (objeto) etc.

No exemplo abaixo vemos uma classe **Cliente** instanciada como objeto **João**, que está associado a loja do centro:



34

3.3 Diagrama de componentes

Este tipo de diagrama denota a estrutura física da arquitetura de um *software*: como ele está organizado e dividido. Pode descrever, por exemplo: como o código fonte está encapsulado em bibliotecas, como ele é distribuído, quais pacotes, arquivos externos, tabelas, banco de dados etc.

Os seguintes elementos são usados no diagrama de componentes:

1. Componentes
2. Interfaces
3. Relacionamentos: dependência, agregação e composição.

⇒ Componentes

Os componentes são os artefatos utilizados para compor o sistema inteiro. Os componentes podem ser:

- Bibliotecas: arquivos JAR, arquivos DLL etc.
- Banco de dados: tabelas, scripts etc.
- Arquivos de configuração

Componentes podem representar agrupamentos de classes, e estão associadas à implementação de uma função dentro de um *software*.

⇒ Interfaces

Assim como em classes, podemos definir interfaces para componentes. Por exemplo, um arquivo JAR, que corresponde a um driver JDBC, que permite acesso a uma base de dados Oracle, é um componente, com interface padrão JDBC. A interface de um componente, neste caso, é mais ampla que uma classe, pois pode abranger várias classes.

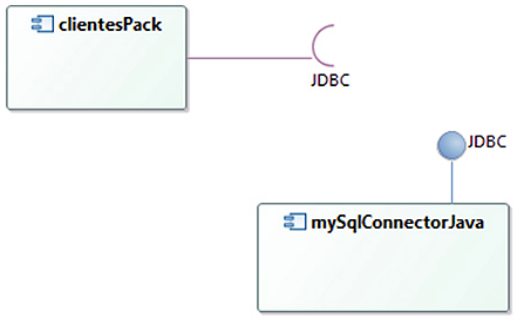
35

⇒ Relacionamentos

Tem o mesmo efeito dos relacionamentos entre classes:

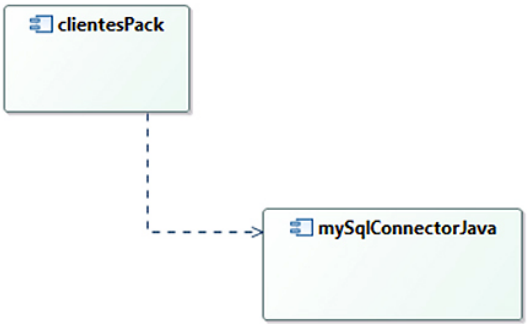
- Dependência;
- Agregação/Composição.

No exemplo abaixo, vemos um diagrama de componentes (criado no UML Designer) ilustrando o uso o JDBC, pelo clientePack (pacote do sistema Aula5Cliente criado na unidade 2 MVC):



MySqlConnectorJava implementa (realiza) uma interface JDBC. E o clientesPack utiliza uma interface JDBC para acessar os dados (que é através da classe SqlConnection).

Poderia também criar um diagrama de componente mais simplificado, como o abaixo:



O diagrama indica que o componente clientesPack depende do componente mySqlConnectorJava, uma vez que as queries SQL implementadas só funcionam no MySql, hipoteticamente.

Dependência
Um componente pode depender de outro componente. Em outras palavras, se uma única classe de um componente realizar uma chamada de um método de uma única classe de outra classe, então existe uma dependência.

Agregação/Composição
Um componente pode ser composto de vários outros componentes. Por exemplo: um componente

que implementa um framework de gerenciamento de banco de dados pode ser composto por vários componentes de acesso a vários tipos diferentes de banco de dados.

36

3.4 Diagrama de instalação

Descreve como um sistema, e seus componentes são dispostos nos componentes de *software* e *hardware*, em seus relacionamentos. Representa a arquitetura e sua configuração e como estão ligados aos componentes. Em outras palavras, mostra o layout físico de um sistema, ou seja, o que é instalado e onde.

Os elementos utilizados no diagrama de instalação são os seguintes:

⇒ Nós

É algo que pode conter *software*. Há dois tipos de nós:

- Dispositivo;
- Ambiente de execução.

O nó é representado por um cubo, e pode ter rótulos, componentes dentro dele etc.

⇒ Caminho de comunicação

Um caminho de comunicação denota um canal de comunicação entre nós e é representado por uma linha. Ele pode ter um rótulo descrevendo o protocolo de comunicação utilizado etc.

Dispositivo
O dispositivo é um *hardware*. Pode ser um computador, um celular, uma impressora, etc.

Ambiente de execução
O ambiente de execução é um *software* que pode conter outros *software*s. Ex.: SGBD, sistema operacional, processo *container* etc.

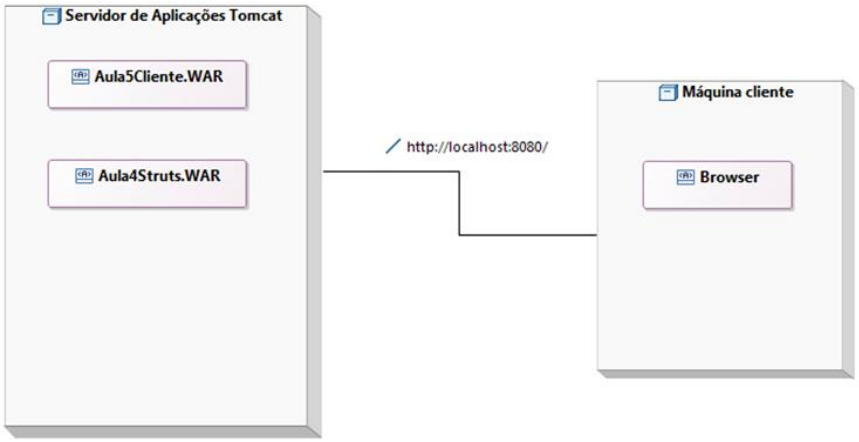
37

⇒ Artefatos

Artefatos são manifestações físicas dos *softwares*. Podem ser os próprios arquivos que compõem o *software*, por exemplo. Podem ser: arquivos de configurações, arquivos executáveis, arquivos de

dados, documentos html etc. Eles se localizam dentro do cubo de um nó, para demonstrar que estão instalados naquele nó.

A seguir, podemos ver um exemplo de diagrama de instalação correspondente aos exercícios que foram feitos nos módulos 1 e 2 (MVC) deste curso:



38

3.5 Diagrama de pacotes

Denota pacotes e suas relações. Em uma abordagem UML abrangente pacote pode conter qualquer elemento UML. Mas em um diagrama de pacotes, um pacote tenta encapsular elementos (como classes) que fazem sentido estarem juntas por causa de um critério lógico e funcional. O principal objetivo do diagrama de pacotes é, então, evidenciar a dependência entre eles.

Os elementos são:

⇒ **Pacotes**

Podem conter outros pacotes, e quaisquer elementos UML dentro dele.

⇒ **Dependência**

Um pacote pode depender de outro pacote. Isso é indicado com uma linha tracejada, que denota um relacionamento “dependência” de um pacote por outro.

Exemplo:

Os elementos internos do pacote podem ter visibilidade diferenciada. No exemplo acima, os sinais "+", "-" e "#" representam publico, privado e protegido.

As dependências podem ter estereótipos como:

- Dependência com estereótipo «access»;
- Dependência com estereótipo «import».

Dependência com estereótipo **«access»**
O pacote de origem (dependente) acessa a elementos exportados pelo pacote de destino

Dependência com estereótipo **«import»**
O conteúdo público do pacote de destino é adicionado ao pacote de origem (dependente)

39

RESUMO

Vimos nesse módulo que o UML (Unified Model Language) é uma linguagem que permite a representação de estruturas e comportamentos de um sistema computacional. Essa linguagem se utiliza de diversos diagramas (representações gráficas) os quais, utilizados da maneira adequada, permitirão ao desenvolvedor o descrever um cenário, desenhar um modelo ou facilitar o entendimento do problema que o sistema deve resolver.

Os principais diagramas previstos na UML 2.0 dividem-se em dois grandes grupos, que são:

- Diagramas estruturais:
  - Diagrama de classes
  - Diagrama de objetos
  - Diagrama de componentes
  - Diagrama de instalação/implantação
  - Diagrama de pacotes

- Diagramas comportamentais:
  - Diagrama de casos de uso
  - Diagrama de estados
  - Diagrama de atividade
  - Diagramas de sequência

UNIDADE 4 – TÓPICOS AVANÇADOS EM PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 4 – PADRÕES DE PROJETO

01

1 - O QUE É *DESIGN PATTERN*?

O *Design Pattern* ou **padrão** de projeto é um formato geral utilizado para implementar/resolver uma categoria de projetos ou problemas, no contexto de projeto de *software*.

Podemos ter um padrão de projeto para sistemas ou para pequenos trechos de código, tais como componentes. Um padrão de projeto pode definir um modo de organizar:

- classes e interfaces dentro de um sistema orientado a objetos;
- sistemas Web;
- Interface gráfica com elementos reutilizáveis;
- Templates para desenvolvimento de ferramentas matemáticas;
- Frameworks para desenvolvimento de software;
- Melhores práticas de desenvolvimento.

Note que **padrões não são códigos prontos**, mas práticas ou princípios, como se fossem descrições de "boas maneiras" para implementar um projeto. Eles visam facilitar a reutilização, a documentação e inclusive manter um vocabulário comum (genérico) a respeito do projeto.

O MVC, que vimos em módulos anteriores, é um padrão de projeto.

Os padrões de projeto são definidos por:

- Nome
- Problema que se propõe resolver
- Solução
- Consequências/Forças

Padrão
Há diversas definições para a palavra "padrão", quase todas remetem a um conceito central, que é a busca por regularidade, repetição ou possibilidade de reprodução, mesmo que a repetição não seja exata ou perfeita.

02

Origens

Os padrões de projetos para *software* foram inicialmente propostos no livro *Design Patterns - Elements of Reusable Object-Oriented Software*. Seus autores ficaram conhecidos como a gangue dos quatro (Gang of Four – GoF), formada inicialmente por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides).

Este livro propôs um primeiro **catálogo de soluções**, que encerra um conjunto de padrões a serem reutilizados em projetos de *software*. Também são conhecidos como padrões GoF.

Outra bem sucedida proposta de padrões de projetos foi o **GRASP** (*General Responsibility Assignment Software Patterns*) que também define um catálogo de padrões e/ou princípios, concebido pelo cientista de computação Craig Larman, que é assunto principal do livro *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development*.

Design Patterns
E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

03

1.1 Padrões GoF

O GoF define 23 padrões, que se dividem em 3 grupos: criação, estrutural e comportamental. São eles:

Criação	Estrutural	Comportamentais
Padrões relacionados à criação de objetos.	Padrões que tratam das associações entre classes e objetos.	Padrões que tratam das interações e divisões de responsabilidades entre as classes ou objetos.
• **Abstract Factory** • Builder • **Factory Method** • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template Method • Visitor

Ao longo deste módulo veremos como utilizar alguns desses padrões em nossos projetos.

04

1.2 Padrões GRASP

Os padrões GRASP são um conjunto de práticas para atribuição de responsabilidades a classes e objetos em projetos orientados a objeto.

O GRASP define nove princípios, nos quais podem ser derivados padrões. São eles:

- Creator (criador);
- Controller (controlador);
- Information Expert (especialista na informação);
- **Low Coupling (baixo acoplamento)**;
- **High Cohesion (alta coesão)**;
- Polymorphism (polimorfismo);
- Pure Fabrication (invenção pura);
- Indirection (indireção);
- Protected Variations (variações protegidas).

Em negrito estão os que utilizaremos ao longo desta unidade.

05

2 - Coesão e Acoplamento

O GRASP define pelos princípios *low coupling* e *high cohesion* que as classes em um sistema devem ser definidas para ter **baixo acoplamento**e **alta coesão**. Mas o que é acoplamento e coesão? Vejamos a seguir.

a) Coesão

Coesão significa uma divisão clara de responsabilidades: cada classe deve ter sua responsabilidade bem definida (de preferência que seja uma única responsabilidade).

Digamos o seguinte código:

```
public class FormCadastro {

    public FormCadastro() {
    }

    public void gravarCliente() {
    }

    public void gravarFornecedor() {
    }

    public void enviarEmail() {
```

```
    }
}
```

A classe **FormCadastro** está coesa? A responsabilidade de gravar clientes ou fornecedores e enviar e-mail deveriam ser desta classe? Como suas responsabilidades não estão bem definidas, dizemos que ela está com **baixa coesão**.

06

Agora um exemplo positivo de **alta coesão**:

```
public class FormCadastro {

    public FormCadastro() {
    }
}

public class Cliente {
    public Cliente() {
    }

    public void gravar (ClienteDAO dao) {
    }
}

public class Fornecedor {
    public Fornecedor(){
    }

    public void gravar (FornecedorDAO dao) {
    }
}
```

Este exemplo tem alta coesão porque as responsabilidades estão mais bem distribuídas e definidas nas diferentes classes. A gravação é responsabilidade da classe de persistência (DAO), por exemplo.

Por ser princípio, você deve segui-lo quando estiver desenvolvendo um *software*. Por quê? Porque se ignorarmos, corremos o risco de criar um *software* difícil de manusear e de se realizar manutenção, além de impossibilitar a reutilização de suas classes. *Softwares* mal estruturados têm sido a causa de incontáveis prejuízos na indústria de desenvolvimento.

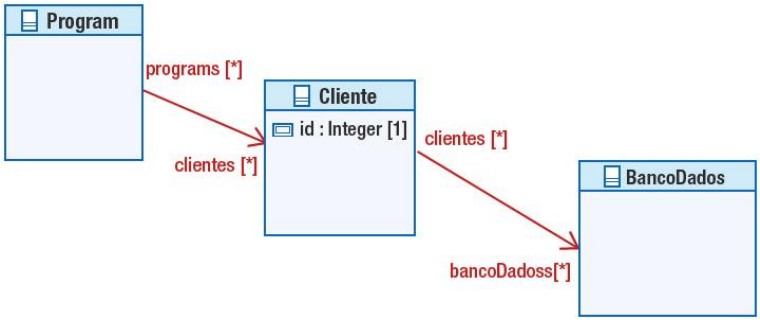
07

b) Acoplamento

Acoplamento significa o quanto uma classe depende de outra para funcionar. Quando uma classe depende muito de outra classe, dizemos que estas duas classes estão fortemente acopladas.

Por princípio, devemos evitar o alto acoplamento entre classes (*low coupling*). Pois se tivermos um grande sistema, com várias classes, e precisarmos modificar uma classe, fica difícil determinar quais classes serão afetadas por aquela modificação, tornando a manutenção complicada, uma vez que fica muito fácil de ocorrer erros de implementação em classes dependentes.

Por exemplo, observe as seguintes classes abaixo:

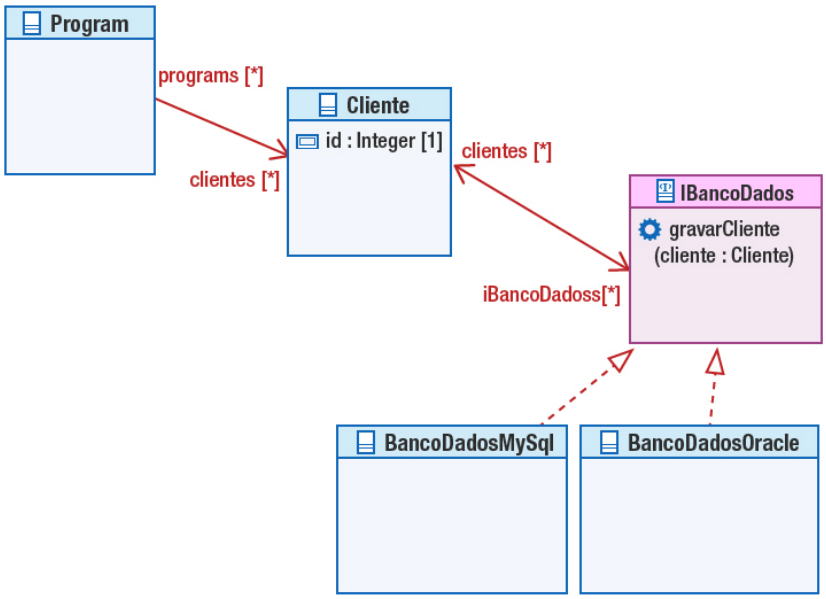


A classe **Program** usa a classe **Cliente**, que usa a classe **BancoDados**, que realiza a gravação de um cliente no banco de dados. Uma alteração na classe **BancoDados** pode gerar implicações nas classes Cliente e Program, por estes estarem relacionadas (dependentes). Isto indica forte acoplamento.

08

Digamos que a classe BancoDados precisasse salvar os dados não só no MySql, mas também em Oracle, o que aconteceria com estas classes? A classe BancoDados começaria a ficar complexa, e sua manutenção difícil.

Para minimizar o acoplamento, poderíamos utilizar *interfaces*. A classe **Cliente** poderia ser ligada a uma interface **IBancoDados**. E então implementaríamos duas classes: BancoDadosMySql e BancoDadosOracle, que implementam a interface IBancoDados. Como se faz isso? Veja abaixo:



Isto faz com que se reduza o acoplamento entre a classe Cliente e as classes que implementam o acesso ao banco de dados. A vantagem é que o desenvolvedor não precisa se preocupar com o que acontece com as classes Cliente e Program enquanto está mexendo no acesso ao banco de dados. Sua única preocupação neste momento será implementar a interface IBancoDados.

09

Vamos praticar?
Digamos que você esteja implementando um jogo tipo **Asteroides**, que tem uma nave que deve atirar em rochas no espaço. Quando o tiro acerta as rochas, elas se dividem em rochas menores (pelo menos umas duas vezes). Utilizando os princípios de baixo acoplamento e alta coesão, modele em diagrama de classes UML, as classes que você utilizaria para criar o jogo, com seus atributos, métodos, relacionamentos e interfaces.
Dica: pense que, no futuro, outros objetos além de rochas poderiam ser colocados no cenário.

10

3 – FÁBRICAS

Segundo o GoF, um método fábrica é definido como “Um padrão que define uma interface para criar um objeto, mas permite às classes decidirem qual classe instanciar.”

Realmente não é uma definição muito simples de entender. Por isso vamos mostrar o exemplo a seguir. Sejam as classes abaixo:

```
public abstract class Carro {
        public String nome;
        public String tipo;
        }

        class sedan extends Carro {
        public sedan(String nome) {
        this.nome = nome;
        System.out.println(“Este sedan é um  ” + this.nome);
        }
        }
        class hatch extends  Carro{
        public hatch(String nome) {
        this.nome = nome;
        System.out.println(“Este veículo hatch é um ” + this.nome);
        }
        }
```

Suponha agora que o usuário deverá em tempo de execução escolher o tipo de carro que ele quer comprar. Veja a seguir.

11

Poderíamos implementar da seguinte forma:

```
         Carro meucarro;
        Scanner scan = new Scanner(in);
        System.out.println(“Entre com o tipo do carro (1-sedan/2-hatch):“) ;
        int tipo = scan.getNextInt();

        if(tipo==1)
          meucarro = new sedan(“”);
        else
        meucarro = new sedan(“”);
```

Veja que, dependendo do tipo, criaremos um objeto de um tipo diferente. Esse código é exemplo de implementação, mas não é a forma mais correta. Para isso usamos o método fábrica.

Esse método permite retornar em tempo de execução o objeto do tipo correto. Uma forma de implementá-lo seria a seguinte:

```
Class fabricadecarros {
        public Carro getCarro(int tipo){
        switch(tipo){
        case 1: return sedan(“");
        break;
        case 2:  return hatch(“");
        }
        }
        }
```

Vejam que esse método então retornará de forma instantânea o objeto correto em tempo de execução, dependendo do argumento do método getCarro. Isso melhora a coesão entre as classes, pois qualquer alteração como a adição de novos carros só precisaria ser feita nesse método.

12

3.1 Abstract Factory

Nos padrões GoF podemos encontrar dois padrões para criação de instâncias de classes:

- Abstract Factory e
- Factory Method.

São padrões para se permitir um baixo acoplamento entre classes e o código fonte responsável por criação de instâncias destas classes.

O padrão **Abstract Factory** especifica como providenciar uma interface para criação de uma família de objetos relacionados sem especificar sua classe concreta.

Para que usar isto? Digamos que vamos fazer um sistema Desktop que possui formulários. Gostaríamos que o código deste sistema suportasse diferentes layouts de tela (Motif, Windows etc.). Para se criar uma tela em um layout, talvez o código de criação de formulário seja diferente do código de criação do mesmo formulário em outro layout. Para separar este código de criação de formulário relativo à arquitetura de cada layout, podemos utilizar uma interface genérica de criação (factory), que isola o código.

Assim, ao invés de chamarmos o construtor de um formulário num dado layout com **new**, usamos um método **factory**.

13

Em java, podemos exemplificar o factory da seguinte maneira:

```
public interface AbstractFormFactory {
    public Form createForm();
}

public interface Form {
    public void Show();
}

public class WindowsFormFactory implements AbstractFormFactory {
    public Form createForm() {
        return new WindowsForm();
    }
}

public class WindowsForm implements Form {
```

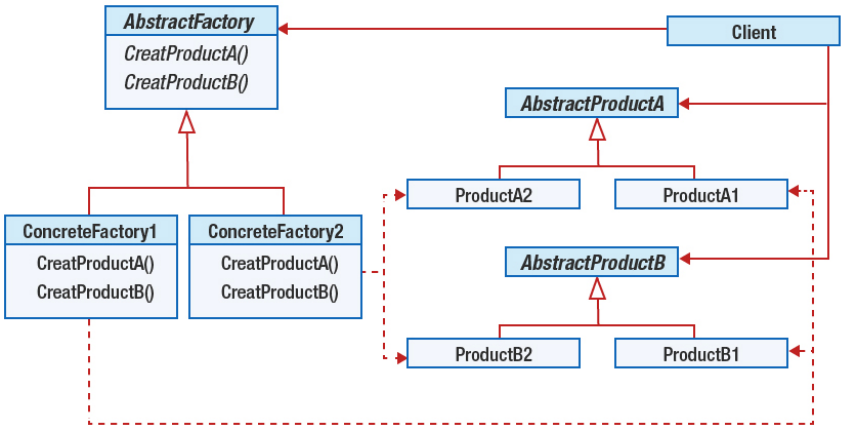
```
    public WindowsForm() {
        // algum código windows aqui
    }

    @Override
    public void Show() {
        // algum código windows aqui
    }
}

public class MotifFormFactory implements AbstractFormFactory {
    public Form createForm() {
        return new MotifForm();
    }
}

public class MotifForm implements Form {
    public MotifForm() {
        // algum código motif aqui
    }

    @Override
    public void Show() {
        // algum código motif aqui
    }
}

public class Programa {
    public static void main(String[] args) {
        AbstractFormFactory fabricaEscolhida;
        if (args[1] == "Windows") {
            fabricaEscolhida = new WindowsFormFactory();
        }
        else
        {
            fabricaEscolhida = new MotifFormFactory();
        }

        // criar janelas
        Form meuForm1 = fabricaEscolhida.createForm();
        Form meuForm2 = fabricaEscolhida.createForm();
        Form meuForm3 = fabricaEscolhida.createForm();
        Form meuForm4 = fabricaEscolhida.createForm();
    }
}
```

Observe o programa principal **Programa**. Escolhemos uma fábrica e não nos preocupamos mais no código se o formulário a ser exibido é no layout Windows ou Motif. Se no futuro quisermos adicionar um novo layout (tipo LayoutDiferente), basta implementarmos duas classes: DiferenteForm e DiferenteFormFactory, e todo o restante do sistema estará compatível com este novo layout.

Facilita ou não facilita a manutenção?

14

Em UML, este padrão é descrito assim:



15

3.2 Factory Method

A diferença deste padrão com o anterior, Abstract Factory, é que este encapsula a criação de objetos deixando que as subclasses decidam quais objetos criar.

Para padrões, você já deve ter percebido que a melhor explicação vem com um exemplo:

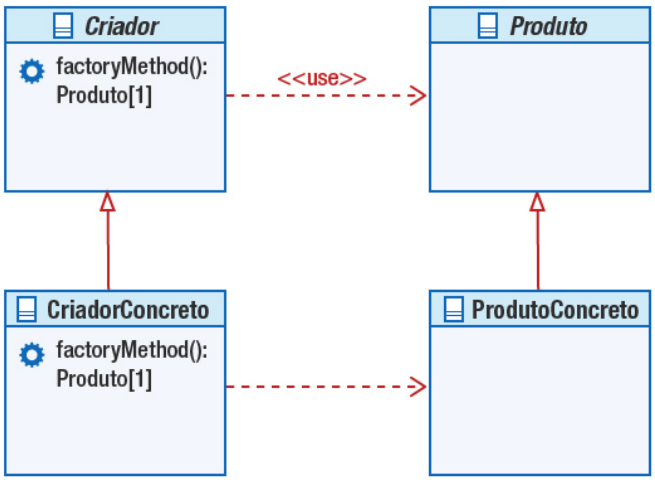
```
public class Cliente  {
    public int idCliente;
    public double renda;
    public Cliente(int idCliente, double renda) {
        this.idCliente = idCliente;
        this.renda = renda;
    }
}

public class ClienteMais extends Cliente {
    public String email;
    public ClienteMais(int idCliente, double renda) {
        super(idCliente,renda);
        email = "Cadastrar email";
    }
}

public class ClienteFactory {
    public Cliente getCliente(int idCliente, double renda) {
        if (renda > 20000)
            return new ClienteMais(idCliente, renda);
        else
            return new Cliente(idCliente, renda);
```

```
    }
}
```

Veja que o método da fábrica **getCliente** decide qual tipo de cliente irá ser criado.

16

Em UML, o modelo geral deste padrão se descreve assim:



Vamos praticar?
Considere que você precise implementar um formulário que visualize uma pessoa, de um cadastro de pessoas. Considere que há diferentes tipos de pessoas: clientes, funcionários e fornecedores. Quando você abrir um formulário para visualizar uma pessoa, este formulário deve ser adequado para o tipo da pessoa que se deseja exibir, pois cada tipo de pessoa tem diferentes dados. Utilizando os padrões Abstract Factory e Factory Method crie classes (e seus relacionamentos) que você utilizaria para implementar estes formulários (não há necessidade de colocar detalhes de implementação da interface gráfica).

17

4 - Outros padrões/princípios GRASP

O GoF e o GRASP tem os padrões e princípios listados na aula 1. Alguns são bem simples e outros complexos. Alguns são implementados por palavras-chaves de linguagens de programação, como por exemplo, o **polimorfismo**, que já foi estudado em java. É mera utilização de classes filhas generalizadas por classes pai, e utilização de métodos virtuais. É um princípio que deve ser seguido.

Os padrões GRASP são mais fundamentais que os padrões GoF, ou seja, se aproximam mais de "princípios" do que "padrões" propriamente ditos.

4.1 Creator

Define um padrão de agregação, no qual um objeto é o próprio criador dos objetos agregados por ele. O objeto criador é um "container" dos objetos criados pelo primeiro. Por exemplo: um "pool" de processos.

18

4.2 Information Expert

Define um padrão onde a responsabilidade de uma classe deve ser completamente providenciada por toda a informação contida naquela classe.

Usando o exemplo anterior, digamos que o sistema precisa saber quais são as threads que estão ativas no pool. Podemos, então, implementar um método *getAllThreads* que retorna a lista de threads ativas no pool:

19

4.3 Controller

Define um padrão que impõe uma separação entre requisições geradas pela camada de interface de um sistema, com as classes de negócio.

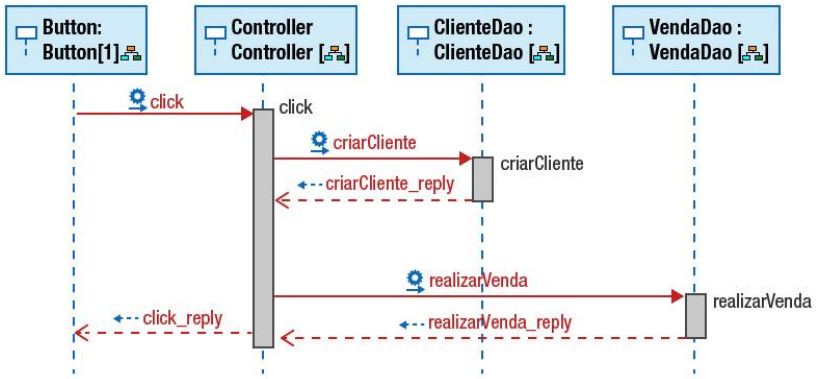
Neste padrão existe uma classe *Controller* que funciona como um "despachante" de requisições. Essa classe é responsável por enviar um determinado tipo de requisição para uma classe/objeto que atende tal requisição.

Com o *Controller*, podemos ter objetos que:

- Representam todas as operações que podem ser realizadas por um sistema inteiro;
- Representam casos de uso.

Um exemplo deste padrão é usado no MVC, implementado no struts, que vimos anteriormente.

Abaixo um diagrama de classes que ilustra o *Controller*:



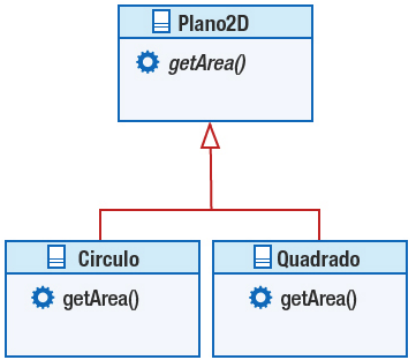
A operação *click* é um botão que, digamos, é utilizado para finalizar uma venda. Digamos que o cliente seja novo e precisa ser criado e salvo em base de dados. O *controller* organiza as várias requisições da camada de negócios em uma única operação (*action*).

20

4.4 Polymorphism

É o padrão que se utiliza o conceito de herança, juntamente com as operações *virtuais*, tal qual já conhecemos em java.

Segue o exemplo:

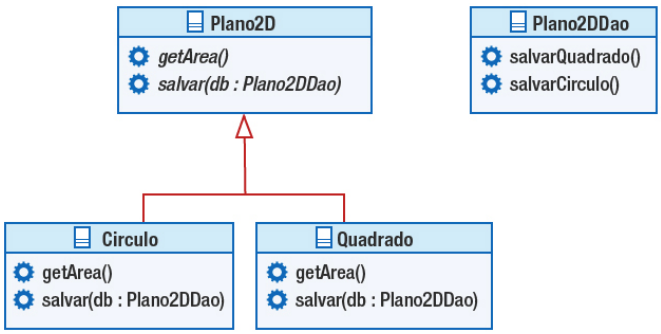


21

4.5 Pure Fabrication

Princípio que define a separação entre as classes de negócio e as classes “artificiais” (técnicas), para promover alta coesão, baixo acoplamento e reuso das classes.

Por exemplo: digamos que no exemplo anterior desejamos salvar as classes Plano2D, Circulo ou Quadrado em um banco de dados. Uma classe “Plano2DDao” deve ser criada, no qual as operações de salvamento e persistências dos objetos seriam implementadas.



22

4.6 Indirection

Este padrão é usado para encapsular um algoritmo ou função que não se encaixa bem com outras classes. Assim como o padrão anterior, implica em habilitar a criação de classes para seu reuso em outros sistemas.

Por exemplo, um sistema de vendas vai adicionando itens de uma compra para fechar o resultado no final em uma única nota fiscal. Com*Indirection*, o cálculo das taxas poderia ser colocado em uma classe separada (TaxasCalculadora), das classes NotaFiscal e ItemCompra.

23

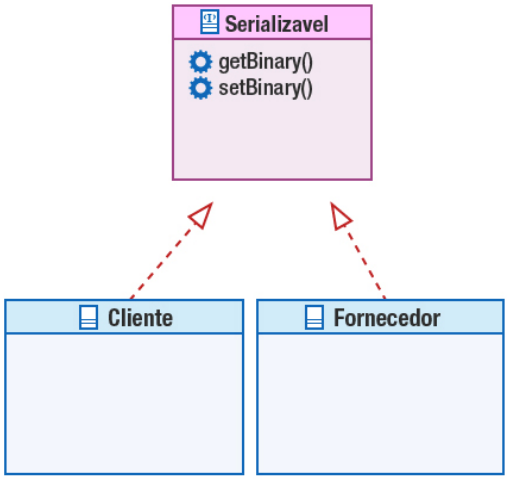
4.7 Protected Variations

Este padrão é usado para resolver o seguinte problema: como criar elementos (objetos, classes ou sistemas) que, em caso de alteração, variação ou instabilidade, ofereça o mínimo de impacto nos elementos que o utilizam.

Ele propõe a utilização de *interfaces*. Tenta-se identificar nos elementos pontos onde a variação de código-fonte futura é previsível para se criar uma interface para mapeamento de responsabilidades.

Interfaces estáveis reduzem o impacto nas dependências dos elementos “interfaceados”. O próprio polimorfismo também é realização deste princípio.

Por exemplo, digamos que você tenha um conjunto de classes javabeans que você gostaria de serializá-las para enviar por socket ou salvar em arquivo.Você pode implementar uma interface chamada **Serializável**, que provê operações padronizadas de serialização (tipo getBinary ou setBinary).



Vamos praticar?

Para cada um dos padrões GRASP, crie exemplo de classes que realizem o princípio proposto por cada padrão.

24

RESUMO

Vimos neste módulo uma introdução ao conceito de padrões de projeto (*design patterns*). Um padrão de projeto consiste em práticas e princípios que visam facilitar a reutilização, a documentação e manter um vocabulário comum em um projeto. São usados normalmente a forma de organizar, classes, interfaces, sistemas web e frameworks.

Apresentamos dois conjuntos de padrões: padrões GoF (*Group of Four*) e padrões GRASP (*General Responsibility Assignment Software Patterns*). O GoF define 23 padrões, que se dividem em três famílias: criação, estrutural e comportamental. Para ilustrar o uso do conjunto de padrões GoF vimos em detalhes o uso de dois padrões fundamentais para a criação de instâncias de classes: **Abstract Factory** e **Factory Method**. São padrões utilizados para permitir um baixo acoplamento entre classes e o código fonte responsável por criação de instâncias destas classes.

Vimos ainda nesse módulo que o GRASP, por sua vez, define 9 princípios, nos quais podem ser derivados padrões, são eles: creator, controller, Information Expert, **baixo acoplamento, alta coesão**, polimorfismo, Pure Fabrication, Indirection e Protected Variations.

O uso de padrões de projeto permite reduzir o tempo de desenvolvimento, melhorar a documentação e garantem uma padronização de desenvolvimento. Esses e outros padrões de projeto serão vistos em mais detalhes ao longo do curso.