Implementado Design Pattern na camada de Controller

Objetivo:

Conhecer os Design Patterns Factory Method e Command e aplica-los para a elaboração de uma classe controladora (Servlet).

Introdução Design Pattern

Um design pattern é uma técnica de modelagem de classes e objetos que resolve um problema comum a diversos tipos de aplicativos. Um design pattern representa uma forma de compartilhar conhecimentos sobre programação orientada a objetos além de otimizar a comunicação entre desenvolvedores através de uma terminologia padronizada.

Podemos de forma simples entender por design patterns, como uma maneira de organizar sua(s) classe(s) diante determinada circunstância.

Podemos citar duas principais categorias de design patterns popularmente utilizados com Java e J2EE:

1. GoF patterns;
2. J2EE design patterns.

Os design patterns GoF (Gang of Four) são mais antigos e utilizados com Smalltalk, C++, Java, entre outras linguagens. Singleton, Factory Method, Memento, Proxy, Adapter, Iterator, Observer e muitos outros fazem parte dos patterns GoF.

Os patterns J2EE são mais focados em problemas encontrados com os padrões de arquitetura e forma de componentização imposta pela plataforma Java 2 Enterprise Edition.

De uma maneira geral podemos afirmar que os diversos design patterns, apesar de servirem para resolver problemas distintos, acabam tendo como objetivo comum, a busca por:

- Código menos extenso possível;
- Código mais legível possível;
- Código mais flexível possível;
- Divisão de responsabilidades entre objetos;
- Aplicação do conceito de reutilização de objetos.

Design Pattern Command

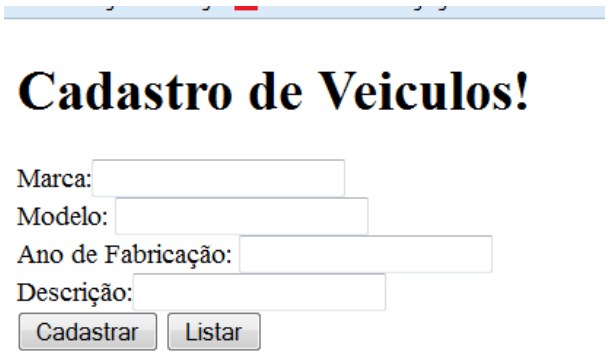
Este design pattern é bastante aplicado em diversos frameworks de interfaces gráficas e interfaces Web.

Aplicamos o Command para encapsular ações que geralmente estão associadas a um objeto de interface gráfica, como por exemplo um botão.

Na Web podemos lembrar do framework Jakarta Struts que encapsula cada ação da Web em uma sub-classe denominada Action. Por este motivo se você já utilizou AWT, Swing ou Struts, com certeza e já foi usuário deste pattern.

Ant-Pattern

Vamos considerar o cenário onde nossa aplicação disponibiliza uma tela com as opções de cadastrar um novo Veículo e Listar os veículos já cadastrados, como a imagem a seguir:




```jsp
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cadastro</title>
    </head>
    <body>
        <h1>Cadastro de Veiculos!</h1>
        <form action="ControleVeiculo" method="Post">
            Marca:<input  type="text" name="txtMarca"><br/>
            Modelo: <input  type="text" name="txtModelo"><br/>
            Ano de Fabricação: <input  type="text" name="txtAnoFabricacao"><br/>
            Descrição:<input  type="text" name="txtDescricao"><br/>
            <input type="submit" name="acao" value="Cadastrar">
            <input type="submit" name="acao" value="Listar">
        </form>
    </body>
</html>
```

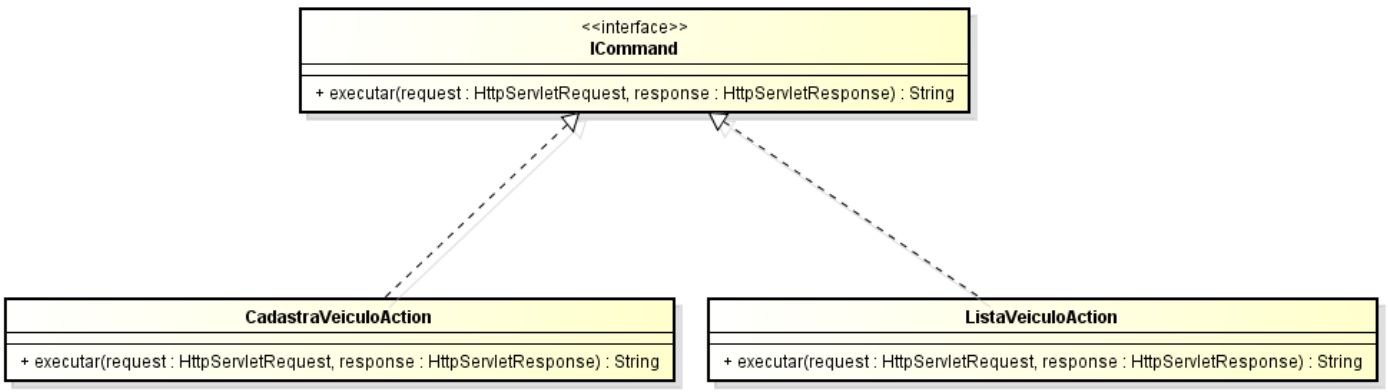
Este form submete as informações juntamente com o comando (botão) acionado pelo usuário para um camada controladora da aplicação, como um Servlet ControleVeiculo, para tratar esta requisição e executar a lógica necessária.

```java
public class ControleVeiculo extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        //recupera a ação do usuário
        String acao = request.getParameter("acao");

        //verifica a lógica que deve ser executada
        if (acao.equals("Cadastrar")) {
            //excuta a lógica para o cadastro
        } else
            if (acao.equals("Listar")) {
            //executa a lógica para listagem
        }
        /*
         * AINDA PODERIAMOS TER:UMA LOGICA PARA CONSULTA,EXCLUSÃO, ETC..
         */
    }

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

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

Esta resolução, é considerada um Ant-Pattern, pois no lugar de modelarmos uma classe por comando temos todas as lógicas agrupadas em uma só classe. O acréscimo de funcionalidades, tais como: Consultar e Excluir, implicará na construção de vários outro IF-else, tornando a modelagem menos flexível em termos de reuso / extensão e manutenção de código.

Aplicando o Design Pattern

Ao aplicarmos este design pattern tipicamente modelamos uma hierarquia de objetos que representarão as ações do nosso aplicativo. Podemos optar por uma pela implementação de um Interface ou classe Abstrata com um método chamado executar, passando como parâmetro objetos de HttpServletRequest e HttpServletResponse para cada classe que implementa uma ação.



Prof. Wolley W. Silva

Interface ICommand

```
13  */
    public interface ICommand {
        public String executar(HttpServletRequest request, HttpServletResponse response) throws Exception;
16  }
17
```

Classe que Implementa a Ação de Cadastrar Veiculo

```
14  public class CadastrarVeiculoAction implements ICommand {
15
        @Override
17      public String executar(HttpServletRequest request, HttpServletResponse response) throws Exception {
18
19          /*
20           * 1 - recupera os paramentos do request
21           * 2 - cria um modelo Veiculo
22           * 3 - popula o modelo Veiculo
23           * 4 - cria um modelo VeiculoDAO
24           * 5 - executa o método cadastrar
25           */
26          //indica a pagina que deve ser exibida
27          return "sucesso_cadastro.jsp";
28      }
29
30  }
```

Classe de Implementa a Ação de Listar Veiculos

```
14  public class ListarVeiculoAction implements ICommand{
15
        @Override
17      public String executar(HttpServletRequest request, HttpServletResponse response) throws Exception {
18
19          /*
20           * 1. Cria um objeto VeiculoDAO
21           * 2. Executa método listar que retorna um List
22           * 3. Adiciona o List no request
23           *
24           */
25
26          //indica a pagina de resposta
27          return "lista_veiculo.jsp";
28      }
29
30  }
```

Agora temos que gerar um vinculo entre nossa Servlet de Controle e a classe de ação correspondente. Este vinculo em geral recorre a outros patterns, como o factory.

Factory Method

Um dos design patterns popularmente utilizado em conjunto com o Command é Factory Method. Factories são classe responsáveis por criar objetos de uma determinada hierarquia através de um sistema de vinculo fraco que permitirá a adição de novas classes Commands sem a alteração de classes que a utilizam.

Quando temos uma hierarquia de classes definidas, como a hierarquia de Commands, fatalmente temos a necessidade de criar as sub-classes conforme um determinado comando foi acionado.

O ideal é que o vinculo esteja apenas com a classe de base Command, como a inteface ICommand, e não com suas demais classes, com isso poderíamos pensar que a criação de um novo comando não acarretaria na alteração de nosso Servlet por exemplo, facilitando a extensibilidade da aplicação.

Ant-Pattern

Podemos imaginar que o seguinte código representa o anti-pattern do Factory Method:

```
14  public class ServletController extends HttpServlet {
15      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
16              throws ServletException, IOException {
17       try{
18           //recupera a ação do usuário
19          String acao = request.getParameter("acao");
20          //cria um Command
21          ICommand commandAction = null;
22
23          //verifica a lógica que deve ser executada
24          if (acao.equals("Cadastrar")) {
25              //excuta a lógica para o cadastro
26              commandAction = new CadastraVeiculoAction();
27          } else
28              if (acao.equals("Listar")) {
29              //executa a lógica para listagem
30                  commandAction = new ListaVeiculoAction();
31          }
32          /*
33           * AINDA PODERIAMOS TER:UMA LOGICA PARA CONSULTA,EXCLUSÃO, ETC..
34           */
35
36          //executa a Action com a lógica
37          String pageDispatcher = commandAction.executar(request, response);
38          RequestDispatcher rd = request.getRequestDispatcher(pageDispatcher);
39          rd.forward(request, response);
40       }catch(Exception e){
41           //trata Exception...
42       }
43
44   }
```

Perceba o forte acoplamento entre a Servlet e cada um dos seus comandos. Esta modelagem conta com a vantagem de encapsular o código de cada comando em classes distintas, porem o código continua crescendo no formato “spaguetti”, com IFs e Elses. Nossa ServletControle precisa contar com um sistema mais dinâmico de fabricação de objetos.

Prof. Wolley W. Silva

Aplicando o Pattern

Grande parte das linguagem oferecem algum sistema de meta-programação ou programação reflexiva para podermos criar classes mais independentes e consequentemente frameworks. Java oferece a Reflection para este fim. Com Reflection temos classes que representam classes, métodos, construtores, entre outros, e podemos chamar métodos ou criar objetos sem vinculo em tempo de compilação.

Vejamos a implementação de Factory Method utilizando Java Reflection:

```
14   public class ServletController extends HttpServlet {
15       protected void processRequest(HttpServletRequest request, HttpServletResponse response)
16               throws ServletException, IOException {
17        try{
18            //recupera a ação do usuário
19           String paramAction = request.getParameter("acao");
20           //monta o nome completo e qualificado da classe
21           String nomeDaClasse = "br.com.commandfactory.controller."+paramAction+"VeiculoAction";
22           //cria um classe de representação (meta-programação)
23           Class classeAction = Class.forName(nomeDaClasse);
24           //instancia a classe utilizando a Factory do objeto Class
25           ICommand commandAction = (ICommand)classeAction.newInstance();
26           //executa a Action
27           String pageDispatcher = commandAction.executar(request, response);
28           RequestDispatcher rd = request.getRequestDispatcher(pageDispatcher);
29           rd.forward(request, response);
30        }catch(Exception e){
31            RequestDispatcher rd = request.getRequestDispatcher("erro.jsp");
32            request.setAttribute("erro", e);
33           rd.forward(request, response);
34        }
35
36    }
```

Uma das grandes vantagens de trabalhar com Factory é que podemos controlar o ciclo de vida, quantidade de objetos e também podemos fazer sistemas de cache e pooling. A grande maioria dos frameworks Java para diversas disciplinas trabalham com Factories para garantir reaproveitamento de objetos e evitar a construção e destruição excessiva de objetos, operação que conforme comentamos, apresenta um alto custo para máquina virtual em frente a grandes demandas.

Design Pattern Java2EE- Front Controller

Ao desenvolvermos componentes Servlets e JSPs para atender aos requisitos necessários do projeto, cada Servlet e cada JSP será um potencial ponto de acesso ao seu aplicativo.

Front Controller propõe a centralização das regras de atendimento às requisições dos usuários com a intenção de promover maior facilidade no controle de requisições que poderão ser complexas e contínuas e poderão necessitar de diversos serviços comuns.

Aplicando o Pattern

Como na ServletController nos utilizamos meta-programação para referenciar as classe Actions que implementam as lógicas de negócio, podemos defini-la com o único ponto de entrada de toda a nossa aplicação, evitando assim duplicação de código.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>ServletController</servlet-name>
        <servlet-class>br.com.commandfactory.controller.ServletController</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletController</servlet-name>
        <url-pattern>/controller.do</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Exercício:

1. Crie um projeto web e aplique neste projeto os padrões de projeto Command, Factory Method e Font Controller.
2. A aplicação deve permitir o cadastro e a listagem de alguma entidade, exemplo: cliente, produto, veiculo, etc...

Referencias

Livro: Padrões de Projeto – GoF

Disponível em: http://online.minhabiblioteca.com.br/#/books/9788577800469

Livro: 33 Desing Pattern aplicados com Java – GlobalCode

Disponível em: http://www.slideshare.net/vsenger/33-design-patterns-com-java

Apostila Caelum: Java para desenvolvimento web

Disponível em: http://www.caelum.com.br/apostila-java-web/

Prof. Wolley W. Silva