# Padrão MVC - Java Magazine

*O artigo trata da importância em utilizar um padrão de projeto como o Model-View-Controller. Explica o que deve ser criado em cada uma das três camadas do padrão MVC e que isso proporciona uma melhor manutenção.*

O uso de um padrão de projeto tem como intuito seguir um modelo pré-determinado de desenvolvimento que busca a melhor adaptação às necessidades dos desenvolvedores.

---

*Guia do artigo:*


- Padrão de Projeto
- O Padrão MVC
- Model ou modelo
- View ou visão
- Controller ou controlador
- Vantagens e desvantagens do padrão MVC
- Desenvolvimento com o padrão MVC
- Sistema desktop utilizando o padrão MVC
- Sistema web utilizando o padrão MVC
- Camada Model (modelo)
- Camada View (visão)
- Camada Controller (controlador)
- Framework web MVC


---

Esse tema é útil para todos os desenvolvedores que queiram aprender, conhecer e utilizar o padrão MVC em seus projetos. Apresenta os benefícios do uso deste padrão e demonstra ainda que ele pode ser utilizado com outros padrões simultaneamente, como com o padrão DAO.

**O artigo apresenta o que é um padrão de projeto e foca seu estudo no modelo Model-View-Controller citando as vantagens e desvantagens na sua utilização**. Demonstra e exemplifica como fazer o uso correto das três camadas do padrão MVC. Aborda o que deve ser implementado em cada uma destas camadas para proporcionar aos desenvolvedores uma manutenção mais fácil e o possível reaproveitamento de classes e partes do projeto em projetos futuros.

O exemplo prático de uma aplicação desenvolvida para desktop – utilizando a API Swing – com base no modelo MVC é comentado e disponibilizado para o leitor.

O Engenheiro Civil Christopher Alexander criou o que se considera o primeiro padrão de projeto em meados da década de 70. É considerado um padrão de projeto uma solução já testada e documentada que possa resolver um problema específico em projetos distintos. Através do trabalho de Alexander, profissionais da área de desenvolvimento de software utilizaram tais conceitos para iniciar as primeiras documentações de padrões de projetos, tornando-as acessíveis para toda a área de desenvolvimento.

O então funcionário da corporação Xerox PARC, Trygve Reenskaug, iniciou em 1979 o que viria a ser o nascimento do padrão de projeto MVC. A implementação original foi descrita no artigo "Applications Programming in Smalltalk-80: How to use Model-View-Controller". A ideia de Reenskaug gerou um padrão de arquitetura de aplicação cujo objetivo é separar o projeto em três camadas independentes, que são o modelo, a visão e o controlador. Essa separação de camadas ajuda na redução de acoplamento e promove o aumento de coesão nas classes do projeto. Assim, quando o modelo MVC é utilizado, pode facilitar em muito a manutenção do código e sua reutilização em outros projetos.

Neste contexto, este artigo apresentará conceitos para a utilização do **padrão de projeto** Model-View-Controller (MVC), suas características e vantagens de uso. Também demonstrará dois pequenos exemplos de projetos utilizando o padrão MVC, um desktop e outro web. Além disso, fará uma rápida abordagem sobre o que é um Framework e quais deles podem auxiliar o programador na utilização dos conceitos MVC em projetos Web. O padrão DAO terá uma rápida abordagem no artigo, sendo demonstrado em qual camada ele deve ser aplicado quando usado em conjunto com o modelo MVC. Ademais, algumas vantagens serão abordadas quando se utiliza um ou mais padrões de projeto em desenvolvimento de software e porque isto se torna uma boa prática.

Acoplamento: é o grau em que uma classe conhece a outra. Se o conhecimento da classe A sobre a classe B for através de sua interface, temos um baixo acoplamento, e isso é bom. Por outro lado, se a classe A depende de membros da classe B que não fazem parte da interface de B, então temos um alto acoplamento, o que é ruim. Coesão: quando temos uma classe elaborada de forma que tenha um único e bem focado propósito, dizemos que ela tem uma alta coesão, e isso é bom. Quando temos uma classe com propósitos que não pertencem apenas a ela, temos uma baixa coesão, o que é ruim.

# Padrão de Projeto

Alexander descreveu um padrão como sendo um problema que se repete inúmeras vezes em um mesmo contexto e que contenha uma solução para resolver tal problema de modo que esta solução possa ser utilizada em diversas situações. O termo padrões de projeto ou Design Patterns, descreve soluções para problemas recorrentes no desenvolvimento de sistemas de software orientados a objetos. O Factory é um destes padrões. Ele é baseado em uma interface para criar objetos e deixar que suas subclasses decidam que classe instanciar. Deste modo, utilizasse o conceito de fábrica de objetos quando um objeto é utilizado para a criação de outros objetos. Algo assim foi implementado no Framework Hibernate para a criação de uma espécie de fábrica de sessões, ou seja, na criação de uma única SessionFactory teríamos acesso a vários objetos do tipo Session.

Padrões de projeto são estabelecidos por um nome, problema, solução e consequências. O nome deve ser responsável por descrever o problema, a solução e as consequências. Quando alguém na equipe de um projeto se refere a um padrão pelo nome, os demais membros da equipe devem relacionar este nome com um problema encontrado, a solução e as consequências na utilização de tal padrão. Encontrar um nome para um novo padrão é considerada uma etapa difícil, já que o nome deve proporcionar a ideia para a qual o padrão foi criado. O problema descreve quando devemos aplicar o padrão, qual o problema a que se refere e seu contexto. A solução descreve os elementos que fazem parte da implementação, as relações entre os elementos, suas responsabilidades e colaborações. Uma solução não deve descrever uma implementação concreta em particular e sim proporcionar uma descrição abstrata de um problema e como resolvê-lo. As consequências são os resultados, vantagens e desvantagens ao

utilizar o padrão. São importantes para avaliar as alternativas descritas bem como proporcionar uma ideia de custos e benefícios ao aplicá-lo.

Um exemplo existente e muito utilizado de padrão é o *Data Access Object*, ou simplesmente *DAO*. Foi uma solução encontrada para resolver problemas referentes à separação das classes de acesso a banco de dados das classes responsáveis pelas regras de negócio. Já o conceito principal do modelo MVC é utilizar uma solução já definida para separar partes distintas do projeto reduzindo suas dependências ao máximo.

Desenvolver uma aplicação utilizando algum padrão de projeto pode trazer alguns dos seguintes benefícios:

- Aumento de produtividade;
- Uniformidade na estrutura do software;
- Redução de complexidade no código;
- As aplicações ficam mais fácies de manter;
- Facilita a documentação;
- Estabelece um vocabulário comum de projeto entre desenvolvedores;
- Permite a reutilização de módulos do sistema em outros sistemas;
- É considerada uma boa prática utilizar um conjunto de padrões para resolver problemas maiores que, sozinhos, não conseguiriam;
- Ajuda a construir softwares confiáveis com arquiteturas testadas;
- Reduz o tempo de desenvolvimento de um projeto.

Nos dias atuais existem diversos padrões e cada um possui uma função bem específica dentro da estrutura do projeto. Deste modo podemos sem problema algum utilizar em um mesmo projeto de software mais de um padrão simultaneamente. Poderíamos fazer uso simultâneo, por exemplo, de padrões como o *Factory*, para criar uma fabrica de sessões com o banco de dados, o *DAO* para separar as classes de CRUD das regras de negócios e o MVC para dividir e diminuir a dependência entre módulos do sistema.

CRUD: A sigla tem origem na língua inglesa para Create, Retrieve, Update e Delete, em português teria o significado de Criar, Recuperar, Atualizar e Excluir. São as quatro operações básicas em um banco de dados.

# O Padrão MVC

O MVC é utilizado em muitos projetos devido à arquitetura que possui, o que possibilita a divisão do projeto em camadas muito bem definidas. Cada uma delas, o *Model*, o *Controller* e a *View*, executa o que lhe é definido e nada mais do que isso.

**A utilização do padrão MVC trás como benefício isolar as regras de negócios da lógica de apresentação, a interface com o usuário**. Isto possibilita a existência de várias interfaces com o usuário que podem ser modificadas sem que haja a necessidade da alteração das regras de negócios, proporcionando assim muito mais flexibilidade e oportunidades de reuso das classes.

Uma das características de um padrão de projeto é poder aplicá-lo em sistemas distintos. O padrão MVC pode ser utilizado em vários tipos de projetos como, por exemplo, desktop, web e mobile.

A comunicação entre interfaces e regras de negócios é definida através de um controlador, e é a existência deste controlador que torna possível a separação entre as camadas. Quando um evento é executado na interface gráfica, como um clique em um botão, a interface irá se comunicar com o controlador que por sua vez se comunica com as regras de negócios.

Imagine uma aplicação financeira que realiza cálculos de diversos tipos, entre eles os de juros. Você pode inserir valores para os cálculos e também escolher que tipo de cálculo será realizado. Isto tudo você faz pela interface gráfica, que para o **modelo MVC** é conhecida como View. No entanto, o sistema precisa saber que você está requisitando um cálculo, e para isso, você terá um botão no sistema que quando clicado gera um evento.

Este evento pode ser uma requisição para um tipo de cálculo específico como o de juros simples ou juros compostos. Fazem parte da requisição neste caso os valores digitados no formulário, como também a seleção do tipo de cálculo que o usuário quer executar sobre o valor informado. O evento do botão é como um pedido a um intermediador que prepara as informações para então enviá-las para o cálculo. Este intermediador nós chamamos de Controller. O controlador é o único no sistema que conhece o responsável pela execução do cálculo, neste caso a camada que contém as regras de negócios. Esta operação matemática será realizada pelo Model assim que ele receber um pedido do Controller.

O Model realiza a operação matemática e retorna o valor calculado para o Controller, que também é o único que possui conhecimento da existência da camada de visualização. Tendo o valor em "mãos", o intermediador o repassa para a interface gráfica que exibirá para o usuário. Caso esta operação deva ser registrada em uma base de dados, o Model se encarrega também desta tarefa.

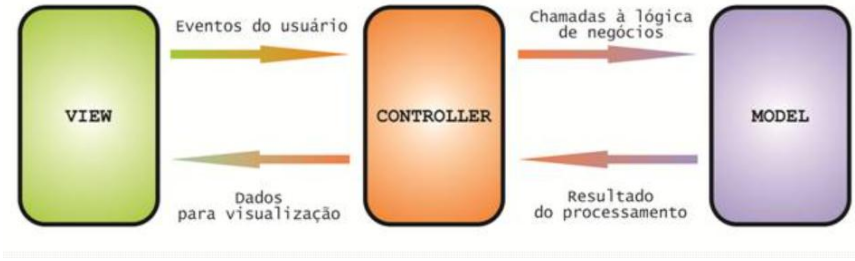


Figura 1. Diagrama de fluxo MVC

Analisando a **Figura 1** podemos ver que o processo de fluxo do MVC é muito simples. O usuário interage com a interface gráfica que é a camada View. A interface gráfica interage com um intermediador que é o Controller, e este interage com o Model que executa as regras de negócios do sistema.

Mas por que usar a camada Controller? Não seria possível fazer a comunicação direta entre View e Model?

Sim, seria possível, mas esta prática não é recomendada por algumas razões. A ligação direta entre as duas camadas acarretaria para o código da interface duas responsabilidades, gerenciar a interface e também lidar com a lógica da camada Model. Isto aumentaria o acoplamento entre as duas camadas, deixando-as muito dependentes e assim, não seria possível a reutilização da interface com outro Model sem que fosse preciso modificar a camada de visualização.

Quando se utiliza a camada Controller, a dependência entre o Model e a View são reduzidas ao máximo, possibilitando um projeto mais flexível, expansível e facilita futuras alterações.

# Model ou modelo

O model é a camada que possui a lógica da aplicação. Ele é o responsável pelas regras de negócios, persistência com o banco de dados e as classes de entidades. O model recebe as requisições vindas do controller e gera respostas a partir destas requisições.

Sempre que pensar em model, pense que ele terá conhecimento apenas dos dados armazenados no sistema e da lógica sobre eles. Estes dados podem estar armazenados em um banco de dados ou até mesmo em arquivos do tipo XML, TXT ou de qualquer outro tipo. É no model também que as operações de CRUD devem ser realizadas. Esta camada na verdade é responsável pelo motivo que a aplicação foi construída, ou seja, ela é o núcleo da aplicação.

# View ou visão

A view é a camada de visualização e representa a parte do sistema que interage com o usuário. É pela interface que haverá a entrada dos dados inseridos pelo usuário e também a saída de informações que serão exibidas para ele. Esses dados serão inseridos ou exibidos geralmente por formulários de entrada ou de saída, tabelas, grids, entre outras formas. A view não contém lógica de negócios, portanto todo o processamento é feito pela camada model e então a resposta é repassada para a view pelo controlador.

Em aplicações de plataforma Desktop, a visão pode ser representada por classes que são construídas com base em componentes do tipo SWING, AWT, SWT, entre outros. Já as aplicações de plataforma Web seriam representadas por páginas do tipo JSP, JSF, entre outros tipos e que são exibidas a partir de um navegador Web.

# Controller ou controlador

Já sabemos que as requisições são enviadas pela view e a lógica de negócios é representada pelo model. Para que haja a comunicação entre essas duas camadas de maneira organizada, é necessário construir a camada controller. Sua função é ser uma camada intermediária entre a camada de apresentação (View) e a camada de negócios (Model).

Deste modo, toda requisição criada pelo usuário deve passar pelo controller, e este então se comunica com o model. Se o model gerar uma resposta para essas requisições, ele enviará as respostas ao controller que por sua vez repassa à camada view.

O controlador serve como um intermediário que organiza os eventos da interface com usuário e os direciona para a camada de modelo, assim, torna-se possível um reaproveitamento da camada de modelo em outros sistemas já que não existe dependência entre a visualização e o modelo.

# Vantagens e desvantagens do padrão MVC

Como todo padrão de desenvolvimento de software, existem vantagens e desvantagens em utilizá-los. O padrão MVC oferece principalmente vantagens, mas pode também considerar-se que possui algumas desvantagens. Vamos relembrar algumas vantagens em desenvolver utilizando o padrão MVC:

- Separação muito clara entre as camadas de visualização e regras de negócios;
- Manutenção do sistema se torna mais fácil;
- Reaproveitamento de código, principalmente da camada de modelo, que pode ser reutilizada em outros projetos;
- As alterações na camada de visualização não afetam as regras de negócios já implementadas na camada de modelo;
- Permite o desenvolvimento, testes e manutenção de forma isolada entre as camadas;
- O projeto passa a ter uma melhor organização em sua arquitetura;
- Torna o entendimento do projeto mais fácil para novos programadores que não participaram de sua criação.

As desvantagens são poucas, mas alguns desenvolvedores acabam não usando o padrão MVC por conta delas, veja algumas:

- Em sistemas de baixa complexidade, o MVC pode criar uma complexidade desnecessária;
- Exige muita disciplina dos desenvolvedores em relação à separação das camadas;
- Requer um tempo maior para modelar o sistema.

Sempre que um sistema for desenvolvido, terá com certeza a necessidade de manutenção como melhorias ou correção de erros e também pode necessitar de varias atualizações. Embora alguns programadores achem que o padrão MVC é muito custoso para ser aplicado, ele se torna muito mais fácil quando há necessidades de executar manutenções e atualizações, já que possui sua estrutura bem definida e suas camadas separadas de forma a diminuir a dependência entre elas.

# Desenvolvimento com o padrão MVC

Utilizando como base para o desenvolvimento de um projeto com o padrão MVC, o sistema financeiro citado anteriormente se aplica como um bom exemplo. Neste sistema teríamos uma interface gráfica para a entrada dos dados referentes ao cálculo e também uma opção que seleciona qual o tipo de cálculo o usuário irá executar. Também é necessário um botão que irá confirmar o pedido da execução do cálculo pelo usuário. O resultado encontrado deverá ser apresentado para o usuário e também ser salvo em uma base de dados. A partir destas informações, vamos primeiramente criar tal sistema utilizando o padrão de projeto MVC em uma plataforma Desktop.

# Sistema desktop utilizando o padrão MVC

A arquitetura básica do projeto terá três pacotes principais, sendo que cada pacote representa uma camada específica do padrão MVC. Inicialmente, para o leitor, esta divisão em pacotes tornará mais fácil a distinção entre as camadas.

A camada `Model` será definida pelo pacote `ejm.appdesktop.model`. Este pacote terá ainda mais três pacotes internos, sendo responsáveis por organizar as classes da aplicação que devem fazer parte apenas da camada de modelo, ou seja, a camada de negócios. A separação de classes entre pacotes é considerada uma boa prática porque reúne em um mesmo pacote classes com objetivos em comum.

O pacote `ejm.appdesktop.model.entity` terá a classe de entidade que representa uma tabela no banco de dados, suas referências e dependências. Sempre que for preciso criar uma nova classe referente a uma tabela do banco de dados, essa classe deverá ser criada dentro deste pacote.

Entidade ou Tabela: onde todos os dados de um banco de dados relacional são armazenados.

Bean: uma classe JavaBean, também conhecida como POJO (Plain Old Java Objects), possui um construtor, atributos privados e os métodos getters e setters públicos.

Classe de Entidade: é uma classe do tipo Bean que faz referência a uma entidade do banco de dados, e cada instância desta entidade representa uma linha (registro ou tupla) na tabela (entidade).

Teremos também o pacote ejm.appdesktop.model.dao, que representará as classes que possuem relação direta com as ações de CRUD utilizando o padrão DAO. Este é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados. Em uma aplicação que utilize a arquitetura MVC, todas as funcionalidades de bancos de dados, tais como obter a conexão e as operações de CRUD, devem ser feitas por classes de DAO. Sempre que existir uma entidade deveremos implementar uma classe no pacote dao que a represente, assim, deixamos as regras de operações CRUD de cada entidade em uma classe específica do tipo DAO.

O último pacote referente à camada model a ser criado é o ejm.appdesktop.model.service, que possui classes com as regras de negócio da aplicação. É considerada uma boa prática ter uma classe do tipo service para cada entidade ou dao do projeto, podendo diminuir assim, a dependência das classes de entidades em relação às classes de CRUD, nos proporcionando maior facilidade de reuso e um código mais limpo. Veja que cada projeto possui objetivos diferentes, e deste modo as regras de negócios podem ser diferentes também. Por exemplo: uma classe do tipo Pessoa possui basicamente um construtor e métodos getters e setters. Caso uma instância da classe PessoaDao seja criada diretamente na classe Pessoa, existirá um alto grau de acoplamento entre Pessoa e PessoaDao. Com isso, se a classe Pessoa fosse reutilizada em um novo projeto que não possui acesso a banco de dados, haveria também a necessidade da classe PessoaDao neste novo projeto, isto porque, há uma instância de PessoaDao na classe Pessoa. Quando uma classe do tipo service é utilizada, esse acoplamento entre a entidade e a classe dao não se faz necessário. Isto se torna responsabilidade da classe service.

Neste momento vale ressaltar que algumas classes e métodos citados no artigo poderão não estar disponíveis nas listagens de códigos devido o foco ser o padrão MVC, porém o projeto apresentado estará acessível para download no site da revista em sua forma completa.

O objetivo do projeto é o desenvolvimento de um sistema financeiro que execute o cálculo de juros. Para isso, será criada uma classe de entidade que represente os atributos contidos nesta operação. Esta classe de entidade representará a tabela no banco de dados chamada Calculos, a qual armazena o resultado dos cálculos executados no sistema. Esta classe Calculo será implementada no pacote ejm.appdesktop.model.entity e pode ser visualizada na **Listagem 1**.

```
package ejm.appdesktop.model.entity;
  public class Calculo {
      private Long id;
      private TipoJuros tipo;
      private Double valorPrincipal;
      private Double taxa;
      private int meses;
      private Double montante;
      private Double totalJuros;
  }
```

**Listagem 1.** Classe de entidade: Calculo

Após implementar a classe Calculo, o foco agora passa a ser o pacote ejm.appdesktop.model.dao, que possui classes referentes à persistência com o banco de dados. Para a comunicação com a base de dados usaremos uma classe chamada Conexao, e o método implementado para abrir uma conexão com o banco de dados da aplicação utiliza a API Java Database Connectivity (JDBC) – apresentado na segunda Edição da Easy Java, no artigo "Persistindo dados em Java".

Já que estamos seguindo o padrão DAO, vamos criar uma classe genérica para as operações com o banco de dados. Uma classe genérica tem como objetivo codificar métodos que sejam comuns para várias classes, assim, não há a necessidade de recodificar os mesmos métodos sempre que for necessário utilizá-los. Geralmente as operações como salvar, alterar, excluir e algumas consultas básicas como por identificador ou a descrição de um campo, são implementadas nesta classe genérica. O padrão DAO sugere que seja criada uma interface e seus métodos sejam implementados em classes concretas. Uma maneira de utilizar este padrão é criar tal interface e em seguida implementar seus métodos em uma classe genérica para então, estender esta

classe genérica em classes mais específicas, acessando seus métodos através do uso de herança. Outra maneira seria criar uma classe genérica do tipo abstrata com os métodos mais básicos já codificados. Neste projeto será implementado o último caso citado, criando uma classe abstrata chamada `GenericDao` porque não necessitamos de uma instância dela e sim apenas que suas subclasses tenham acesso a seus métodos.

Por fim, no pacote dao, temos a classe `CalculoDao` que tem como objetivo a codificação de métodos específicos para as operações referentes à tabela `Calculos`. Nesta classe estendemos a classe abstrata `GenericDao` para acessar o método `save()`, descrito na **Listagem 2**. Como estamos usando `JDBC` puro, sem a existência de nenhum framework de persistência, teremos que criar o `SQL` referente ao `insert` e enviá-lo ao método `save()` da classe genérica. Para isso, codificamos um método chamado `salvar()` na classe `CalculoDao` e enviamos o `SQL` e mais alguns parâmetros para o método `save()` da classe `GenericDao`. Conforme descrito na **Listagem 3**, temos a classe `CalculoDao` com o método `salvar()`.

```
public void save(String insertSql, Object... parametros) {
    try {
      PreparedStatement pstmt = connection.prepareStatement(insertSql);
      for (int i = 0; i < parametros.length; i++) {
        pstmt.setObject(i + 1, parametros[i]);
      }
      pstmt.execute();
      pstmt.close();
    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      close();
    }
  }
```

*Listagem 2. Classe genérica GenericDao: método save()*

```
package ejm.appdesktop.model.dao;

  import ejm.appdesktop.model.entity.Calculo;

  public class CalculoDao extends GenericDao<Calculo> {
      public void salvar(Calculo c) {
          String insert = "INSERT INTO " +
                  "CALCULOS  (TIPO,  VALOR_PRINCIPAL,  TAXA,  MESES,  MONTANTE,
TOTAL_DE_JUROS) " +
                  "VALUES (?,?,?,?,?,?)";

          save(insert,    c.getTipo().getDescricao(),    c.getValorPrincipal(),
c.getTaxa(),
                  c.getMeses(), c.getMontante(), c.getTotalJuros()
          );
      }
  }
```

*Listagem 3. Classe de persistência: CalculoDao*

Resta ainda a classe `CalculoService` que pertence ao pacote `ejm.appdesktop.model.service` para finalizar a implementação da camada `Model`. Ela será responsável pelas regras de negócio da aplicação. É nesta classe que será realizado o

cálculo de juros simples, juros compostos e também é a partir dela que teremos acesso à classe de persistência `CalculoDao` – veja na **Listagem 4**. Utilizar uma classe do tipo `Service` é considerada uma boa prática por isolar regras de negócio das classes do tipo `DAO` ou de `entidades`, assim, estas classes passam a ter mais possibilidades de reuso em outros projetos.

```
package ejm.appdesktop.model.Service;

  import ejm.appdesktop.model.dao.CalculoDao;
  import ejm.appdesktop.model.entity.Calculo;

  public class CalculoService{
      private CalculoDao dao;

      public CalculoService() {
          this.dao = new CalculoDao();
      }

      public Calculo jurosSimples(Calculo calculo) {
          double principal = calculo.getValorPrincipal();
          double taxa = calculo.getTaxa() / 100;
          int meses = calculo.getMeses();
          double juros = principal * taxa * meses;
          double montante = principal * (1 + (taxa * meses));
          calculo.setTotalJuros(juros);
          calculo.setMontante(montante);
          salvar(calculo);
          return calculo;
      }

      public Calculo jurosCompostos(Calculo calculo) {
          double principal = calculo.getValorPrincipal();
          double taxa = calculo.getTaxa() / 100;
          int meses = calculo.getMeses();
          double montante = principal * Math.pow((1 + taxa), meses);
          double juros = montante - principal;
          calculo.setTotalJuros(juros);
          calculo.setMontante(montante);
          salvar(calculo);
          return calculo;
      }

      public void salvar(Calculo calculo) {
          dao.salvar(calculo);
      }
  }
```

**Listagem 4.** Classe de regras de negócio: CalculoService

Como nossa camada Model está terminada, vamos agora trabalhar na camada `Controller`, que está no pacote `ejm.appdesktop.controller`. Na **Listagem 5** temos a interface `IController` com a assinatura do método `executa()`. Será este método o responsável por realizar a comunicação entre a camada `Controller` e a camada `View`.

Na **Listagem 6** a classe concreta `CalculoController` implementa a interface `IController`. O método `executa()` recebe por parâmetro um objeto da classe de visualização. Através deste objeto recebemos os valores inseridos pelo usuário nos campos do formulário e também é por ele que o resultado do cálculo processado será retornado para a interface gráfica. Além disso, temos mais dois métodos na classe `CalculoController: calcular()`, que irá selecionar qual o tipo de cálculo será executado; e `doubleFormat()`, que irá formatar o resultado encontrado com apenas duas casas após da vírgula para uma melhor apresentação ao usuário.

```
package ejm.appdesktop.controller;

  public interface IController {
      public void executa(Object view);
  }
```

*Listagem 5. Interface do pacote controller: IController*

```
package ejm.appdesktop.controller;

  import ejm.appdesktop.model.Service.CalculoService;
  import ejm.appdesktop.model.entity.Calculo;
  import ejm.appdesktop.view.CalculoForm;

  import java.text.NumberFormat;
  import java.util.Locale;

  public class CalculoController implements IController {

      private Calculo calculo;
      private CalculoForm frame;

      public void executa(Object view) {
          frame = (CalculoForm) view;
          calculo = new Calculo();
          calculo.setValorPrincipal(
                  Double.parseDouble(frame.getTxtValorPrincipal().getText())
          );
          calculo.setTaxa(Double.parseDouble(frame.getTxtTaxa().getText()));
          calculo.setMeses(Integer.parseInt(frame.getTxtMeses().getText()));
          calculo.setTipo(frame.getRdBtnSimples().isSelected() ?
                  Calculo.TipoJuros.SIMPLES : Calculo.TipoJuros.COMPOSTOS
          );

          calcular(calculo);

          String montante = doubleFormat(calculo.getMontante());
          String juros = doubleFormat(calculo.getTotalJuros());

          frame.getTxtTotJuros().setText(juros);
          frame.getTxtMontante().setText(montante);
      }

      private Calculo calcular(Calculo calculo) {
          CalculoService service = new CalculoService();

          Calculo resultado;
          if (calculo.getTipo().getValor() == 1) {
```

```
            resultado = service.jurosSimples(calculo);
        } else {
            resultado = service.jurosCompostos(calculo);
        }
        return resultado;
    }

    private String doubleFormat(Double aDouble) {
        NumberFormat nf =
                NumberFormat.getCurrencyInstance(new Locale("pt", "BR"));

        nf.setMaximumFractionDigits(2);
        return nf.format(aDouble);
    }
}
```

## Listagem 6. Classe do pacote controller: CalculoController

Vamos agora passar para a interface gráfica que deve ser desenvolvida no pacote `ejm.appdesktop.view`, utilizando a `API Swing`. A classe `CalculoForm`, conforme a **Listagem 7**, será responsável pela codificação desta interface com o usuário. O resultado visual será como o exemplo da **Figura 2**.

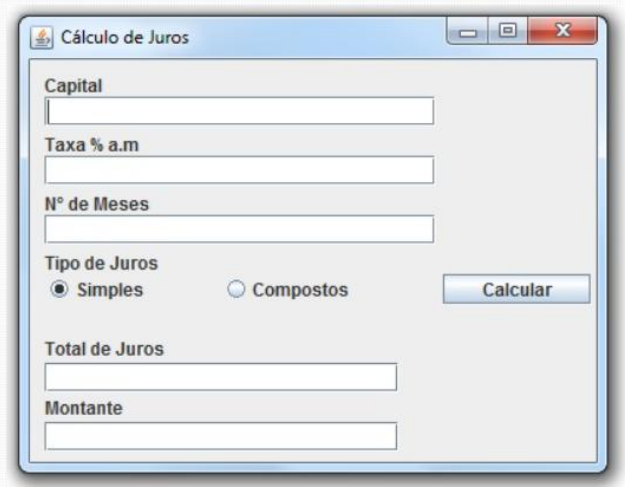


FIgura 2. Interface com o usuário - padrão desktop

A interface é bastante simples, possui alguns campos para a entrada de dados, um seletor do tipo de cálculo que deve ser executado e um botão para realizar o cálculo. Os dois últimos campos, `Total de Juros` e `Montante`, serão atualizados com os seus valores respectivos após o cálculo ser efetuado pelo `Model`.

Veja que temos na classe `CalculoForm` uma ligação apenas com o `Controller`, através da interface `IController`. Esta classe é responsável apenas por gerar o formulário e não conhece mais nada da aplicação a não ser o controlador. Após os dados serem inseridos e um clique no botão `Calcular`, fazemos uma chamada ao método `executa()` e passamos como parâmetro um objeto da classe `CalculoForm`. Assim que essa requisição chegar ao `Controller`, será selecionado o tipo de cálculo que deve ser executado e o método `calcular()` fará uma chamada ao método específico da classe `CalculoService`. Após o cálculo ser executado, os dados serão persistidos no banco de dados. Em seguida a execução volta para a classe `CalculoController` e um novo estado do objeto calculo passa a possuir o valor do total de juros e do montante. Estes valores serão formatados com duas casas após a vírgula pelo método `doubleFormat()` e inseridos no objeto frame. Por fim, com a alteração no estado do objeto frame, os valores de juros e do montante passam a ser exibidos na interface gráfica.

```
package ejm.appdesktop.view;
```

```
import                          ejm.appdesktop.controller.CalculoController;import
ejm.appdesktop.controller.IController;
import javax.swing.*;import java.awt.*;import java.awt.event.ActionEvent;import
java.awt.event.ActionListener;
public class CalculoForm extends JFrame {
    private  JLabel  lbValorPrincipal,  lbTaxa,  lbTipo,  lbMeses,  lbMontante,
lbTotJuros;
    private  JTextField  txtValorPrincipal,  txtTaxa,  txtMeses,  txtMontante,
txtTotJuros;
    private JRadioButton rdBtnSimples, rBtnComposto;
    private JButton btnCalcular;

    //métodos get foram omitidos

    public CalculoForm() {
        //código omitido
    }

    private void onClickCalculo() {
        controller = new CalculoController();
        controller.executa(this);
    }

    public static void main(String[] args) {
        CalculoForm form = new CalculoForm();
        form.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }}
```

**Listagem 7.** Pacote View: Classe da interface com o usuário

Nesta primeira parte do artigo, foi apresentada uma visão geral sobre Padrões de Projeto com o foco principal no padrão MVC. Uma breve abordagem **sobre o padrão DAO** também foi apresentada, demonstrando que este padrão, quando usado simultaneamente com o MVC, deve ser implementado na camada de modelo.

Por fim, o artigo exemplifica a construção de uma pequena aplicação financeira na plataforma desktop, com o uso da API Swing, baseada nos conceitos do padrão MVC. Na segunda parte do artigo, o leitor poderá acompanhar o desenvolvimento de uma nova aplicação, porém, desta vez, na plataforma web. Serão utilizados os conceitos de reuso de classes e camadas entre projetos sempre com foco no padrão de projeto Model-View-Controller.

# Sistema web utilizando o padrão MVC

Na primeira parte do artigo desenvolvemos uma aplicação financeira na plataforma desktop (Swing) que tinha como objetivo o cálculo de juros simples e juros compostos utilizando o padrão de projeto MVC. Nesta segunda parte, o artigo irá demonstrar como aplicar os conceitos do padrão MVC em uma aplicação financeira, desta vez na plataforma web. Vamos analisar como tratar cada uma das três camadas do padrão MVC e apresentar que diferenças podem ser encontradas, na camada de visão e na camada de controle, quando um projeto desktop é reestruturado para o ambiente web.

# Camada Model (modelo)

No projeto da aplicação desktop a camada *model* foi desenvolvida no pacote *ejm.appdesktop.model*. Esta camada é responsável pela lógica de negócios da aplicação, sendo constituída pelos pacotes *ejm.appdesktop.model.entity* (*classes de entidades*), *ejm.appdesktop.model.dao* (*classes de acesso a banco de dados*) e *ejm.appdesktop.model.service* (classe com as regras de negócio). Veja mais sobre estes pacotes no artigo anterior.

Como o projeto web terá o mesmo objetivo do projeto desktop (aplicação financeira), vamos reaproveitar as classes da camada *model*. Isto é possível porque as classes do modelo não possuem nenhum tipo de dependência com a camada de visualização (interface gráfica) e também não são dependentes do controlador. É o controlador que deve se adequar ao modelo e não o contrário. Deste modo os programadores têm um ganho significativo no tempo de desenvolvimento da nova aplicação, já que a camada de modelo está pronta e não precisa ser codificada novamente.

A maneira apropriada para reutilizar classes Java é através do uso de bibliotecas. Uma biblioteca não é nada mais que um arquivo do tipo *JAR* contendo um grupo de classes. Por exemplo, o Framework *Hibernate* é uma biblioteca que possui inúmeras classes que formam o framework – veja na Edição 4 da *Easy Java*, o artigo "*Dominando o Hibernate*". Quando um projeto utiliza o *Hibernate* para persistência de dados, não se faz necessária a implementação destas classes e sim apenas adicionar na aplicação a biblioteca que possui as classes que formam o *Hibernate*.

Java Archive (ou JAR) é um arquivo compactado usado para distribuir um conjunto de classes Java, um aplicativo Java, ou outros itens como imagens, XMLs, entre outros. É usado para armazenar classes compiladas e metadados associados que podem constituir um programa.

Como o pacote model da aplicação desktop será reutilizado integralmente, vamos montar uma biblioteca que contenha suas classes. Deste modo deixa de ser necessário reprogramar essas mesmas classes para a aplicação web. Para montar esta biblioteca em um arquivo *JAR* devemos realizar os seguintes passos (imaginando que o projeto desktop esteja armazenado no diretório C:\AppDesktop):

- **1**. Crie no mesmo nível do pacote *src* uma pasta chamada *dist* e dentro dela um arquivo chamado *manifest.txt*;
- **2**. Abra o prompt de comando e entre no diretório *src* da aplicação;
- **3**. Digite os seguintes comandos para criar as classes e os pacotes no diretório *dist*:

```
C:\AppDesktop\src>javac -d ../dist ejm/appdesktop/model/entity/*.java
C:\AppDesktop\src>javac -d ../dist ejm/appdesktop/model/dao/*.java
C:\AppDesktop\src>javac -d ../dist ejm/appdesktop/model/service/*.java
```

- **4**. Agora que geramos os arquivos *.class*, vamos criar o arquivo *lib-model.jar* executando o seguinte comando:

```
C:\AppDesktop\dist>jar -cvmf manifest.txt lib-model.jar ejm
```

Após executar o último comando citado, temos no diretório *dist* o arquivo *lib-model.jar*. Este arquivo possui todas as classes e pacotes da camada *Model* e deve ser adicionado ao projeto da aplicação Web. Essa biblioteca será inserida no diretório de bibliotecas externas da aplicação, normalmente este diretório é chamado de *lib*. Bibliotecas externas são aquelas que possuem classes que não estão disponíveis no JRE (*Java Runtime Environment*) e são necessárias para o funcionamento da aplicação.

# Camada View (visão)

No projeto anterior, o Swing foi utilizado para construção da interface com o usuário já que tínhamos uma aplicação na plataforma desktop. Quando se trabalha com páginas web não se pode fazer uso do Swing, então é necessária uma interface gráfica que possa ser construída para rodar em um navegador web (Firefox, Chrome, IE, entre outros).

Em Java essa interface pode ser desenvolvida através de páginas **JSP (JavaServer Pages)**, que é uma tecnologia de geração dinâmica de documentos HTML que integra a plataforma **Java EE**. O JSP é um arquivo texto contendo **código HTML** e tags JSP específicas para a geração dinâmica do documento. Essa tecnologia permite separar a programação lógica (parte dinâmica) da programação visual (parte estática), facilitando o desenvolvimento de aplicações, onde programador e designer podem trabalhar no mesmo projeto, mas de forma independente.

O Java EE (Java Enterprise Edition) é a plataforma Java voltada para redes, internet, intranets e semelhantes, sendo assim, ela contém bibliotecas especialmente desenvolvidas para o acesso a servidores. O Java EE foi desenvolvido para suportar uma grande quantidade de usuários simultâneos. Esta plataforma contém uma série de especificações, como: JDBC, JSP (Java Server Pages), Servlets, entre outras.

O uso de Tags aumenta a produtividade e simplifica o desenvolvimento de páginas dinâmicas JavaServer Pages. A biblioteca padrão de tags JSP, chamada de "Standard Actions" é identificada pelo prefixo jsp: <jsp:forward page="index.jsp" />.

No desenvolvimento de aplicações web em Java, temos uma execução no lado do servidor (Classes Java – Servlet) e uma execução no lado cliente (Browser – Código HTML). O usuário precisa informar no lado cliente os valores que serão usados para a operação de cálculo (capital, taxa de juros, meses e tipo de juros) e o lado servidor é responsável por executá-la. Para informar tais valores é necessário criar um formulário que contenha esses campos para que o usuário possa preenchê-los e ao clicar em um botão esses dados sejam enviados a um **Servlet**. Um Servlet responde por uma determinada URL, portanto, só precisamos indicar que ao clicar no botão devemos enviar uma requisição para esse Servlet.

Para isso, vamos criar uma página JSP – chamada calculoForm.jsp – contendo um formulário HTML (**Figura 1**) para que o usuário possa preencher os dados da operação de cálculo de juros (**Listagem 1**). O código dessa página é apresentado na **Listagem 1** e o resultado de sua execução pode ser visto na **Figura 1**.

```
<jsp:include page="cabecalho.jsp"/><html><head><title>Cálculo de
Juros</title></head><body>
    <form action="calculo" method="post">
        <div id="div2">
            <div align="center">
                Capital:<input type="text" name="valorPrincipal" value=""
size="20">
            </div>
```

```
            <div align="center">
                Taxa % a.m:<input type="text" name="taxa" value="" size="15">
            </div>
            <div align="center">
                N. de Meses:<input type="text" name="meses" value="" size="14">
            </div>
            <div align="center">Tipo de Juros:</div>
            <div align="center">
                <input       type="radio"       name="tipoJuros"       value="1"
checked="true">Juros Simples
                <input type="radio" name="tipoJuros" value="2">Juros Compostos
            </div>
            <br>

            <div align="center">
                <input type="reset" value="Limpar">
                <input type="submit" value="Calcular">
            </div>
        </div>
    </form></body></html><jsp:include page="rodape.jsp"/>
```

**Listagem 1.** *calculoForm.jsp – Código HTML da interface gráfica para entrada de dados para o cálculo de juros*

Na linha 01 será incluída uma página cabeçalho (**Figura 2**) e na linha 31 uma página rodapé (**Figura 3**), apenas para deixar a página da aplicação com uma melhor aparência. Note que da linha 05 até a linha 28 temos o formulário que é criado por meio da tag `<form>`. Como parâmetros desta tag temos o método da requisição (`method="post"`) e a URL para onde será submetida a requisição (`action="calculo"`). Entre as linhas 06 e 20 temos o código HTML que irá gerar os campos para a entrada de dados. Todas as informações digitadas pelo usuário na interface gráfica serão enviadas para o `Servlet` através da URL após o clique no botão `Calcular`. Como foi usado o método `POST`, esses valores não ficam visíveis na URL.

Caso fosse usado o método do tipo `GET`, esses valores seriam visíveis na URL do navegador, como no exemplo a seguir: "`http://localhost:8080/AppWeb/calculo?valorPrincipal=2&taxa=2&meses=2&tipoJuros=1`".


**Figura 1.** Página JSP – Interface para entrada de dados para o cálculo de juros


Figura 2. Página JSP - cabecalho.jsp


Figura 3. Página JSP - rodape.jsp

Após os dados do formulário *calculoForm.jsp* serem inseridos pelo usuário e ele clicar no botão *Calcular*, as informações serão enviadas e processadas pela classe *CalculoController* na camada *controller*. O cálculo requerido será executado pelas classes da camada model e o resultado do cálculo será exibido ao usuário pela página *resultadoForm.jsp* da camada de visão, apresentado na **Listagem 2** e demonstrado no browser na **Figura 4**. Nesta página foi adicionada uma tabela que irá exibir o resultado do cálculo de juros através da marcação *${atributo}*. Cada linha da tabela é referente a um desses atributos: *valorPrincipal*, *taxa*, *meses*, *tipo*, *juros* e *montante*. Para recuperar, por exemplo, o valor do atributo montante (linha 29), usasse a seguinte forma: *${montante}*. Desta maneira podemos recuperar o valor de qualquer atributo que seja enviado para a página pelo controlador. Na linha 33 da **Listagem 2** temos um link para retornar à página *calculoForm.jsp* e realizar um novo cálculo. Assim chega ao fim o desenvolvimento da camada de visualização, que é toda criada com base em arquivos JSP.

```
<jsp:include page="cabecalho.jsp"/><html><head><title>Cálculo de
Juros</title></head><body>
    <div id="div2">
        <table cellspacing="0" cellpadding="2" align="center">
            <tr bgcolor="#5f9ea0">
                <td>Capital:</td>
                <td>${valorPrincipal}</td>
            </tr>
            <tr bgcolor="#f0f8ff">
                <td>Taxa % a.m:</td>
                <td>${result.taxa}</td>
```

```
            </tr>
            <tr bgcolor="#5f9ea0">
                <td>N. de Meses:</td>
                <td>${result.meses}</td>
            </tr>
            <tr bgcolor="#f0f8ff">
                <td>Tipo de Juros:</td>
                <td>${result.tipo}</td>
            </tr>
            <tr bgcolor="#5f9ea0">
                <td>Total de Juros:</td>
                <td>${juros}</td>
            </tr>
            <tr bgcolor="#f0f8ff">
                <td>Montante:</td>
                <td>${montante}</td>
            </tr>
            <tr>
                <td></td>
                <td        align="right"><a        href="/AppWeb/index.jsp">Novo
Calculo</a></td>
            </tr>
        </table>
    </div></body></html><jsp:include page="rodape.jsp"/>
```

*Listagem 2. resultadoForm.jsp – Código HTML da interface gráfica para exibir o resultado do cálculo de juros*


**Figura 4.** Página Jsp - Interface para retornar ao usuário o resultado do cálculo de juros

# Camada Controller (controlador)

Após o termino do desenvolvimento da camada de visão, partimos para entender como a camada de controle irá se comunicar com as camadas de visualização e de modelo. Nesta aplicação web, o controlador se faz presente no pacote `ejm.appweb.controller`, onde temos a classe chamada `CalculoController`. Esta classe – apresentada na **Listagem 3 –** estende a classe `HttpServlet`, que possui alguns métodos necessários para trabalhar com aplicações web. Um destes métodos é o `service()`, que será usado para receber da página`calculoForm.jsp` os valores informados pelo usuário. Através deste método também será possível redirecionar o resultado do cálculo para a página `resultadoForm.jsp` – **Figura 4** – da camada de visualização.

Uma página quando é criada com a tecnologia JSP, depois de adicionada em um servidor de aplicação compatível com a tecnologia `Java EE` (`Tomcat`, `JBoss`, `GlassFish`, entre outros), é transformada em um `Servlet`. Um `Servlet` nada mais é que uma classe Java que implementa uma interface, a `HttpServlet`. Os servlets normalmente utilizam o protocolo `HTTP`, apesar de não serem restritos a ele. Quando um `Servlet` recebe uma requisição (`GET` ou `POST`) ela pode capturar os seus parâmetros e executar qualquer processamento de uma classe Java. Na linha 05 da **Listagem 1**, veja que foi utilizado o método `POST` para enviar os dados necessários. Na classe `CalculoController` (**Listagem 3**) o método `service()` possui dois parâmetros: `HttpServletRequest` e `HttpServletResponse`. É através destes parâmetros que é possível realizar a troca de informações entre páginas JSP e classes Java.

Para o controlador se comunicar com a camada de modelo e acessar os métodos que realizam os cálculos de juros (simples ou compostos), será necessário instanciar um objeto da classe `CalculoService`. Para o acesso a essa classe e qualquer outra classe da camada de modelo no controlador, devemos importá-las da biblioteca `lib-model.jar` adicionada ao projeto. Essa importação pode ser visualizada na **Listagem 3** nos dois primeiros imports da classe `CalculoController`. Na aplicação desktop a comunicação entre controlador e modelo foi realizada no método `calcular()`. Este método também será utilizado na aplicação web sem a necessidade de nenhuma modificação. Já o método `executa()` (da aplicação desktop) será substituído pelo método `service()` (da aplicação web). A lógica usada nestes métodos é muito semelhante, porém a maneira como os dados da camada de visualização são recuperados pelo controlador é diferente. Estes valores agora são recuperados através do parâmetro `HttpServletRequest`. Utilizando o método `getParameter`(“`nome-do-campo`”) é possível recuperar o valor de cada campo do formulário `calculoForm.jsp` e inseri-lo em um objeto da classe `Calculo` da camada `model`.

Após a operação do cálculo requisitado pelo usuário, é preciso enviar o resultado para a camada de visualização, especificamente para página `resultadoForm.jsp`. Este processo é realizado também pelo método `service()`, no entanto precisamos enviar os valores atuais da operação de cálculo de juros, e para isso devemos utilizar o objeto `HttpServletRequest`. Desta vez outro método deste objeto será chamado, o `setAttribute`(“`campo`”, “`valor`”). Ele trabalha com dois parâmetros, no primeiro é informado o nome do campo no formulário `resultadoForm.jsp` que receberá o valor e o segundo parâmetro é o valor que queremos exibir na página JSP.

Tendo todos os valores inseridos no objeto `HttpServletRequest`, será necessário criar um objeto do tipo `RequestDispatcher`. Este objeto será responsável por enviar os dados do controlador para a camada de visualização. Para indicar qual página JSP deverá ser exibida ao usuário com o resultado da operação, usasse o método `getRequestDispatcher()` do objeto `HttpServletRequest` passando como parâmetro uma String com a página que deverá ser exibida. Este envio é realizado através do método `forward()` do objeto `RequestDispatcher`, que recebe como parâmetro os objetos `HttpServletRequest` e `HttpServletResponse`.

Vimos então como realizar a comunicação entre a camada de visualização e o controlador. Este processo é um pouco diferente em relação ao utilizado no exemplo do primeiro artigo, onde a camada view era uma classe do tipo `Swing`. Já a comunicação do

controlador com a camada model foi exatamente igual em relação às plataformas desktop e web, o que proporcionou um significativo ganho de tempo para desenvolver a aplicação web.

```
package ejm.appweb.controller;
import                      ejm.appdesktop.model.Service.CalculoService;import
ejm.appdesktop.model.entity.Calculo;
import                                javax.servlet.RequestDispatcher;import
javax.servlet.ServletException;import     javax.servlet.http.HttpServlet;import
javax.servlet.http.HttpServletRequest;import
javax.servlet.http.HttpServletResponse;import        java.io.IOException;import
java.text.NumberFormat;import java.util.Locale;
public class CalculoController extends HttpServlet {

    private Calculo calculo;

    public    String    executa(HttpServletRequest    request,HttpServletResponse
response)throws Exception {

        int tipo = Integer.parseInt(request.getParameter("tipoJuros"));

        calculo = new Calculo();


calculo.setValorPrincipal(Double.parseDouble(request.getParameter("valorPrincip
al")));
        calculo.setTaxa(Double.parseDouble(request.getParameter("taxa")));
        calculo.setMeses(Integer.parseInt(request.getParameter("meses")));
        calculo.setTipo(tipo    ==    1    ?    Calculo.TipoJuros.SIMPLES    :
Calculo.TipoJuros.COMPOSTOS);

        calcular(calculo);

        request.setAttribute("valorPrincipal",
doubleFormat(calculo.getValorPrincipal()));
        request.setAttribute("taxa", doubleFormat(calculo.getTaxa()));
        request.setAttribute("meses", calculo.getMeses());
        request.setAttribute("tipo", calculo.getTipo().getDescricao());
        request.setAttribute("juros", doubleFormat(calculo.getTotalJuros()));
        request.setAttribute("montante", doubleFormat(calculo.getMontante()));

        RequestDispatcher                    dispatcher                    =
request.getRequestDispatcher("resultadoForm.jsp");

        dispatcher.forward(request, response);
    }

    private Calculo calcular(Calculo calculo) {
        //idêntico ao projeto desktop
    }

    private String doubleFormat(Double aDouble) {
        //idêntico ao projeto desktop
    }}
```

**Listagem 3.** Pacote controller da aplicação web: Classe CalculoController

# Framework web MVC

O reuso de software tem sido um dos principais objetivos da Engenharia de Software, e reutilizar software não é nada simples. Quando falamos em framework, estamos falando em reuso, que significa reutilizar o que já foi feito e não reinventar. A maior vantagem disto é a economia do tempo gasto no desenvolvimento de um projeto.

Utilizar um framework que seja baseado no padrão Model-View-Controller pode simplificar bastante no desenvolvimento de aplicações para a plataforma Web em Java. Os frameworks são construídos com base em diversos padrões de projeto e boas práticas, assim, abstraem da equipe de desenvolvimento partes mais complexas envolvidas no processo de desenvolvimento de aplicações.

Existem hoje diversos frameworks disponíveis para desenvolvimento utilizando o padrão MVC e escolher um deles não é considerada uma tarefa fácil, já que cada um possui propriedades particulares e distintas. Alguns dos frameworks bastante utilizados no mercado de desenvolvimento de software são:

- JavaServer Faces (JSF), que é baseado em componentes e possui facilidades para o desenvolvimento da interface gráfica;
- Struts 2, apresenta o uso de anotações, suporte a Ajax, entre outras;
- VRaptor 3, framework baseado em convenções ao invés de configurações, minimizando o uso de XML e de anotações. Possui sua documentação disponível em português;
- Spring MVC, um dos projetos do renomado Spring Framework, proporciona grande parte de sua configuração através do uso de anotações e muito pouco XML.

Hoje em dia o mercado de trabalho, como é bastante voltado à área de desenvolvimento web, exige que os programadores tenham conhecimento de pelo menos um framework web que dê suporte ao **padrão MVC**.

# Conclusões

Padrões de projeto é uma prática considerada essencial no desenvolvimento de aplicações, trazendo inúmeras vantagens para o sucesso do projeto. Entre esses vários padrões existentes, o artigo abordou especificamente o padrão MVC. **E para exemplificar o uso do MVC em um projeto web**, desenvolvemos uma aplicação financeira e analisamos a função de cada uma das camadas do padrão. Foi demonstrado também que o reuso de classes entre sistemas diferentes pode ser uma boa prática aplicada pelos programadores para economizar tempo no processo de desenvolvimento de um sistema, e que o padrão MVC é bastante favorável para isso.

Um exemplo prático de como criar bibliotecas Java em arquivos do tipo JAR, a partir de classes que podem ser reutilizadas em sistemas diferentes, foi apresentado. Por fim, foi citada a existência de alguns frameworks que auxiliam os programadores no desenvolvimento de aplicações web baseadas no padrão MVC.

Referências:

**Livro voltado para o aprendizado sobre a perspectiva de desenvolvimento de aplicações através de padrões de projetos.** Use a Cabeça!: Padrões de Projeto (Design Patterns), Eric Freeman & Elisabeth Freeman - Editora Alta Books – 2005.

**Foco principal do livro são padrões, as melhores práticas, as estratégias de design e soluções aprovadas utilizando as principais tecnologias J2EE.**

Core J2EE Patterns, Deepak Alur & John Crupi & Dan Malks – Editora Campos – 2004.

---

Referência deste artigo:

de Souza, M. B. (2011, August 9). *Padrão MVC (Model-View-Controller) tutorial*. DevMedia. https://www.devmedia.com.br/padrao-mvc-java-magazine/21995

-Pequenas adaptações para o formato pdf.