# CRIANDO MVC com Java e desktop

## Introdução

Nesse artigo iremos abordar um padrão de projeto muito interessante, o **MVC** (*Model, View, Controller*). Quem nunca criou um "sistema-linguiça", aquele que tem meia duzia de arquivos e cada arquivo faz tudo, processa, acessa banco, mostra resultados na tela e estoura pipoca, que atire a primeira pedra! É muito comum fazermos esse tipo de programação quando estamos começando a desenvolver, geralmente são classes “Bom-Bril”, mil e uma utilidades, todas amarradas com scripts SQL misturados com tomadas de decisões e apresentações gráficas, um caos na hora de dar manutenção, parece um castelo de cartas. Tudo bem, e o que o MVC tem a ver com isso?
Segundo Erich Gamma, *"A abordagem MVC separa Visão e Modelos pelo estabelecimento de um protocolo do tipo inserção/notificação (subscribe/notify) entre eles. Uma visão deve garantir que a sua aparência reflita o estado do modelo"*
O padrão MVC coloca ordem nisso tudo, estipulando regras de separação do código de acordo com as funcionalidades, distribuindo a **aplicação em camadas** e fazendo com que elas sejam o mais independente possível umas das outras.

## Exemplo modular

Por exemplo, você pode fazer uma aplicação que se baseia em: receber um valor, processar e retornar uma resposta, sem salvar nada em banco de dados nem exibir graficamente. Agora imagine que você resolva acoplar a esse sistema um modulo que salve no banco de dados o resultado do processo e depois outro módulo para exibir o resultado com interface gráfica. Se isso não for bem estruturado, pode lhe dar uma tremenda dor de cabeça para implementar e outra maior ainda para dar manutenção.

## Analogia com o mundo real

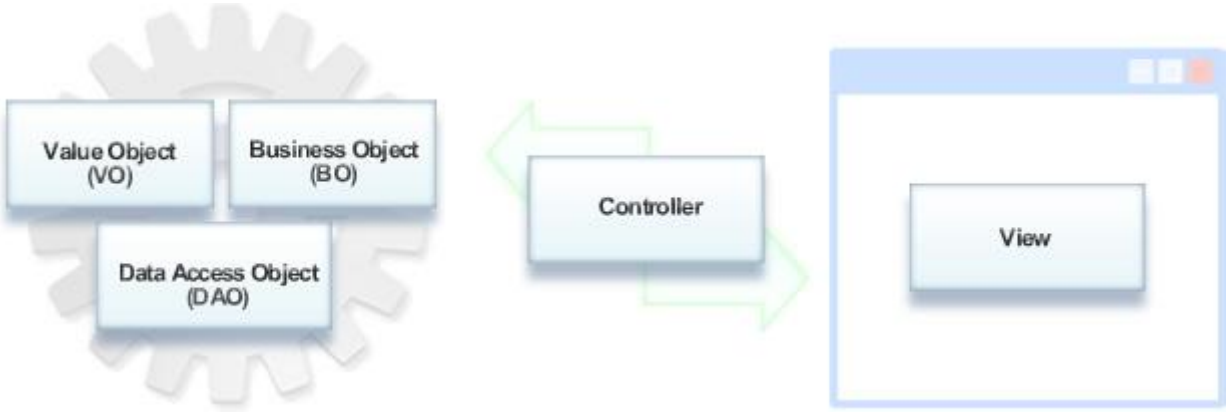
Uma analogia do MVC com o mundo real poderia ser o funcionamento de um carro. No carro temos o motor que faz o processo principal, gerar força mecanica. Temos também os pedais e câmbio de marchas. Além disso, temos o painel de controle do carro que exibe informações de como está o seu funcionamento, como temperatura, pressão do óleo e medidor de rotação do motor.

Colocando o exemplo do carro no padrão MVC temos a seguinte estrutura:
O motor do carro é certamente a camada ***model***, pois se trata do núcleo da aplicação (carro), exercendo o maior trabalho. Os pedais, cambio e painel fazem parte da camada ***view***, embora o painel seja diferente dos pedais e câmbio já que é responsável em exibir dados e os outros em colher dados (interação com o usuário).
Agora imagine, já temos o motor e os elementos de interação com o usuário mas está faltando algo. Se não houvesse nada entre esses elementos e o motor, nada aconteceria, você poderia pisar o quanto quisesse no acelerador, trocar qualquer marcha que nada aconteceria, o painel seria apenas um monte de ponteiros e luzes que não funcionariam para nada. É nesse ponto que entra a camada de ***controller***. Essa camada corresponderia aos sensores de temperatura, de rotação do motor e de pressão do óleo do motor, responsaveis em fazer a interação entre a camada ***model*** e a ***view***, fazendo com que a pressão exercida nos pedais interfira no funcionamento do motor e que o painel mostre o estado do motor.

## Estrutura MVC

A estrutura básica do MVC é a seguinte:

- Model
  - Value Object
  - Business Object
  - Data Access Object
- View
- Controller




Esquema MVC

Se você encontrar outras estrutura MVC's diferentes mundo a fora, não se assuste, afinal de contas futebol, religião e MVC são difíceis de se encontrar o melhor e o pior.

Na próxima parte do artigo iremos por a mão na massa e desenvolver um "pequeno" sistema que irá usar todas as camadas faladas aqui, além de explicar a fundo a camada ***model*** e suas integrantes.

## Camada model

Explicado todos esses conceitos, vamos agora falar da principal camada da aplicacação, a camada ***model***. Como vimos anteriormente, ela é dividida em três tipos de classes, As *Value Objects*, *Business Object* e *Data Access Object*. Sua função é prover todas as funcionalidades do software independente de interação com o usuário ou parte gráfica.

## Value Objects (VO)

Os objetos de valores (*Values Objects*) são classes que contém variáveis e metodos de acessos, além de construtores. Um exemplo desse tipo de classe seria assim:

```
01 package br.edu.qi;

02 public class Pessoa
   {

03     //Variaveis

04     private String cpf;
```

```
05    private String nome;
06
07    //Construtor
08    public Pessoa(String cpf, String nome) {
09        this.cpf = cpf;
10        this.nome = nome;
11    }
12
13    //Get and Set
14    public String getCpf() {
15        return cpf;
16    }
17
18    public void setCpf(String cpf) {
19        this.cpf = cpf;
20    }
21
22    public String getNome() {
23        return nome;
24    }
25
26    public void setNome(String nome) {
27        this.nome = nome;
28    }
29 }
```

Objetos de valor tem a função de modelar uma entidade, uma abstração do mundo real. Por exemplo a classe pessoa, precisa conter todos os atributos que lhe interessar a respeito de uma pessoa. Esse tipo de classe é muito utilizado para representar uma tabela do banco de dados, por exemplo, se você tiver uma tabela de cliente, precisará de uma classe desse tipo com os mesmos

atributos que contém na tabela do banco de dados. É muito mais organizado e reutilizável trafegar todos os atributos encapsulados dentro de um objeto do que passar uma "enchorrada" de variáveis de um canto ao outro.
Os metodos de acesso (*Get and Set*) servem para garantir uma flexibilidade na hora de impor alguma regra no acesso aos atributos, embora você **talvêz** nunca utilize-os. Por exemplo, digamos que hoje o método ***getNome()*** apenas retorne o valor da variável nome e é exibido na tela. Derrepente você decidiu que todo mundo terá o título de Sr. e quer implementar essa regra, basta ir ao método***getNome()*** e mudar o retorno assim:

```
1 public String getNome() {

2     return "Sr. "+nome;

3 }
```

Pronto, você acaba de impor uma regra no acesso ao atributo nome e poderá fazer isso em qualquer outro método.

Você pode dar o nome que quiser para seus métodos de acesso à atributos, poderia ser **obtemNome()** e **defineNome()** que iria funcionar do mesmo jeito, porém existe uma padronização em utilizar os prefixos *get* e *set*.
Com o uso do construtor fica mais "elegante" criar um objeto, por exemplo, em qualquer lugar que precisar de uma instância de **Pessoa**, basta fazer assim:

```
1 Pessoa p = new Pessoa("430.376.565-14", "Joaquim da silva");
```

Dessa forma você terá um objeto da classe **Pessoa** com todos os atributos já com valores definidos e poderá acessá-los a qualquer momento a partir dos metodos de acesso.

```
1 Pessoa p = new Pessoa("430.376.565-14", "Joaquim da silva");

2 System.out.println(p.getNome());
```

Isso imprimiria no console o nome da pessoa.

### Data Access Object (DAO)

As classes do tipo DAO são encarregadas de fazer o **acesso à dados**, seja eles em um fluxo de rede, arquivo ou banco de dados. Por exemplo, métodos responsáveis em fazer acesso ao banco de dados devem estar nesse tipo de classe, assim como métodos que manipulam arquivos ou que enviam e recebem dados pela rede. Lembre-se, **entrada e saída de dados**!

### Business Object (BO)

Esse tipo de classe também compõem a ***model*** da aplicação, assim como os dois tipos de classe ditos anteriormente. A especialidade das classes BO's é resolver operações complexas, são os processos principais da aplicação, digamos que o "miolo" do software. Nessas classes são processadas regras de negócio e tomadas de decisão.

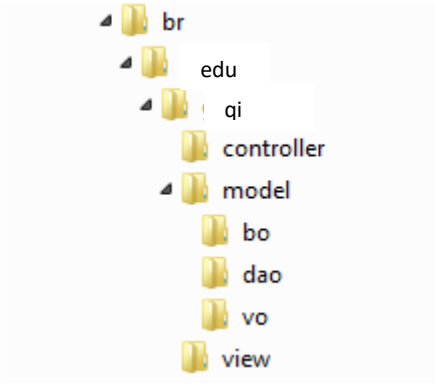
### Mãos à obra!

Vamos agora desenvolver um software que implemente o MVC usando apenas a camada ***model***, mais a frente terminaremos o software adicionando as outras camadas. Esse software será capaz de ordenar uma sequência de dígitos numéricos e salvar dois arquivos de texto, um contendo

todo os passos que foram nescessários para que os números fossem ordenados e um outro com o resultado.
Primeiro, vou considerar que você já tenha o Java instalado na sua máquina e que saiba se virar com alguma IDE ou utilize um editor de texto comum (se você for guerreiro!), eu vou utilizar o NetBeans.
Para começarmos, crie um projeto Java-SE (Standard Edition, aplicação desktop). No meu caso o projeto se chama 'MVC'. Em seguinda crie uma estrutura de pastas com os seguintes nomes:

- br.edu.qi
- br.edu.qi.**controller**
- br.edu.qi.**model**
  - br.edu.qi.**model.bo**
  - br.edu.qi.model.**dao**
  - br.edu.qi.model.**vo**
- br.edu.qi.**controller**
- br.edu.qi.**view**

Por padrão a estrutura de pastas são nomeadas com o domínio da empresa ou do desenvolvedor de forma invertida (**br.edu.qi**). Se você ainda não tem um domínio, invente! Coloque o seu nome ou o do projeto com diretório principal.
Se você criou os diretórios através de uma IDE, irá notar que na prática foi criado mais diretórios do que parecem. Se você navegar pelas pastas fora da IDE, irá ver que ficou assim:




Estrutura de pastas MVC

Isso é o correto, se a sua estrutura não ficou assim, tem algo errado.

## Implementando a VO

Vamos utilizar duas classes desse tipo, uma chamada **Passos** que conterá as informações relacionadas à uma verificação pelo método de ordenação. A outra classe se chama **Ordenação** e contém os dados relativo ao resultado final do processo. Se não entendeu muito ainda, não tem problema, quando estiver implementado e funcionando você endentenderá.

```
01 package br.edu.qi.model.vo;
02
```

```
03 /**
04  *
05  * @author Gustavo Ferreira
06  * @see Classe que armazenas os dados relativo a cada passo da ordenacao
07  */
08 public class Passos {
09
10     private String numeroAnterior;
11     private String numeroResultante;
12     private String descricao;
13
14     /**
15      * Construtor que preenche todos os atributos do objeto
16      * @param numeroAnterior
17      * @param numeroResultante
18      * @param descricao
19      */
20     public Passos(String numeroAnterior, String numeroResultante, String descricao) {
21         this.numeroAnterior = numeroAnterior;
22         this.numeroResultante = numeroResultante;
23         this.descricao = descricao;
24     }
25
26     //Get and Set
27     public String getDescricao() {
28         return descricao;
29     }
```

```
30
31  public void setDescricao(String descricao) {
32      this.descricao = descricao;
33  }
34
35  public String getNumeroAnterior() {
36      return numeroAnterior;
37  }
38
39  public void setNumeroAnterior(String numeroAnterior) {
40      this.numeroAnterior = numeroAnterior;
41  }
42
43  public String getNumeroResultante() {
44      return numeroResultante;
45  }
46
47  public void setNumeroResultante(String numeroResultante) {
48      this.numeroResultante = numeroResultante;
49  }
50
51  //Sobrescrevendo o toString do objeto
52  @Override
53  public String toString() {
54      if (this.getNumeroAnterior() == null) {
55          return "\nDescricao: ".concat(this.descricao);
56      } else {
```

```
            return this.getNumeroAnterior().concat(" >>
57 ").concat(this.getNumeroResultante()).concat("\nDescricao:
   ").concat(this.descricao).concat("\n\n");

58        }

59    }

60 }
```

Segue a baixo a outra classe de valor

```
01 package br.edu.qi.model.vo;

02

03 /**

04  *

05  * @author Gustavo Ferreira

06  * @see Classe que cria objeto capaz de armazenar os dados relativo ao processo

07  * de ordenacao

08  */

09 public class Ordenacao {

10

11    private int numeroOriginal;

12    private String numeroOrdenado;

13    private int qtdeTrocas;

14

15    /**

16     * Construtor padrao que insere em todos os atributos do objeto

17     * @param numeroOriginal

18     * @param numeroOrdenado

19     * @param qtdeTrocas

20     */

21    public Ordenacao(int numeroOriginal, String numeroOrdenado, int qtdeTrocas) {
```

```
22        this.numeroOriginal = numeroOriginal;
23        this.numeroOrdenado = numeroOrdenado;
24        this.qtdeTrocas = qtdeTrocas;
25    }
26
27    //Get and Set
28    public String getNumeroOrdenado() {
29        return numeroOrdenado;
30    }
31
32    public void setNumeroOrdenado(String numeroOrdenado) {
33        this.numeroOrdenado = numeroOrdenado;
34    }
35
36    public int getNumeroOriginal() {
37        return numeroOriginal;
38    }
39
40    public void setNumeroOriginal(int numeroOriginal) {
41        this.numeroOriginal = numeroOriginal;
42    }
43
44    public int getQtdeTrocas() {
45        return qtdeTrocas;
46    }
47
48    public void setQtdeTrocas(int qtdeTrocas) {
```

```
49        this.qtdeTrocas = qtdeTrocas;
50    }
51
52    //Sobrescrevendo o toString do objeto
53    @Override
54    public String toString() {
55        return
56                String.valueOf(this.numeroOriginal)
57                .concat(" virou:\n")
58                .concat(String.valueOf(this.numeroOrdenado))
59                .concat("\nQtde de trocas: ")
60                .concat(String.valueOf(this.qtdeTrocas));
61    }
62 }
```

### Implementando a DAO

Também teremos duas classes DAO's, pois o sistema manipulará dois arquivos de texto. Se o sistema manipulasse dez arquivos, seria dez classes, uma para cada arquivo com os métodos que forem nescessários (excluir, salvar, editar, atualizar, etc.). A mesma regra valeria se fosse para manipular tabelas do banco de dados, uma DAO para cada tabela.

```
01    package br.edu.qi.model.dao;
02
03    import br.edu.qi.model.vo.Passos;
04    import java.io.FileNotFoundException
;
05    import java.io.PrintWriter;
06    import java.util.List;
07
08    /**
09     *
```

```
10     * @author Gustavo Ferreira
11     * @see Classe que executa as operacoes de IO (entrada e saida) do sistema com relacao
12     * aos dados resultantes do passo a passo
13     */
14    public class PassosDAO {
15
16        /**
17         * Metodo que recebe todos os passos (lista) e salva todos em um arquivo
18         * @param passos
19         * @throws FileNotFoundException
20         */
21        public void salvarPassos(List<passos> passos) throws FileNotFoundException{
22            PrintWriter pw = new PrintWriter("passos.txt");
23            for (Passos p : passos){
24                pw.print(p);
25            }
26            pw.flush();
27            pw.close();
28        }
29    }
30 </passos>
```

Segue abaixo a outra classe DAO

```
01 package br.edu.qi.model.dao;
02
03 import br.edu.qi.model.vo.Ordenacao;
04 import java.io.FileNotFoundException
;
```

```
05 import java.io.PrintWriter;
06
07 /**
08  *
09  * @author Gustavo Ferreira
10  * @see Classe que executa as operacoes de IO (entrada e saida) do sistema com relacao
11  * aos dados resultantes da ordenacao
12  */
13 public class OrdenacaoDAO {
14
15     /**
16      * Metodo que salva em um arquivo de texto os dados do objeto de ordenacao
17      * @param Ordenacao ordenacao
18      * @throws FileNotFoundException
19      */
20     public void salvar(Ordenacao ordenacao) throws FileNotFoundException{
21         PrintWriter pw = new PrintWriter("ordenacao.txt");
22         pw.print(ordenacao);
23         pw.flush();
24         pw.close();
25     }
26 }
```

### Implementando a BO

Nosso sistema terá apenas uma BO com apenas um método, o de ordenação pelo algoritmo "Bolha". Sistemas comerciais geralmente tem várias BO's, uma para cada responsabilidade. Por exemplo, essa BO que vamos criar é responsável por ordenar números e poderia conter vários métodos, cada um utilizando um algoritmo diferente de ordenação.

Todo acesso à classes DAO's é feito através das classes BO's. É como se a BO dissesse: **"Ninguém vai à DAO senão por mim!"**

Imagine que você tem um sistema com três DAO's, cada uma acessa uma tabela do banco de dados. A sua aplicação precisa inserir dados em uma tabela e logo em seguida inserir nas outras duas mas se der erro na inserção da última tabela, a ação toda precisa ser desfeita para garantir que **"Ou insere em todas ou em nenhuma"**. Para assegurar isso, será preciso usar transações que deverão ser manipuladas pela BO. Ou seja, a BO solicita as ações para as três DAO's e ela mesmo gerencia para poder desfazer se for nescessário. Por isso sempre chame DAO's a partir da BO!

A BO a seguir pode confundir bastante, principalmente se você ficar batendo cabeça para entender como funciona a lógica de ordenação. Se isso acontecer, pense o seguinte: É uma classe que contém um metodo responsável em receber um número (532135), ordena-lo (123355) e em seguida solicitar às duas DAO's que salvem o passo-a-passo e o resultado, só isso.

```
01  package br.edu.qi.model.bo;
02
03  import br.edu.qi.model.dao.OrdenacaoDAO;
04  import br.edu.qi.model.dao.PassosDAO;
05  import br.edu.qi.model.vo.Ordenacao;
06  import br.edu.qi.model.vo.Passos;
07  import java.util.ArrayList;
08  import java.util.List;
09
10  /**
11   *
12   * @author Gustavo Ferreira
13   * @see Classe que contem o(s) metodo(s) de ordenacao e eh capaz de processa-los
14   */
15  public class OrdenacaoBO {
16
17      /**
18       * Metodo responsavel em fazer a ordenacao pelo algorito BubbleSort (metodo bolha)
19       * @param Int numero
20       * @return Ordenacao
```

```
21        */
22       public Ordenacao bubbleSort(int numero) {
23          try {
24             //Transforma em String para fazer as trocas consideranco caracter por caracter
25             //Converto o numero do tipo Int para String e depois gero um array de chars.
26             char[] digitos = String.valueOf(numero).toCharArray();
27             //Nosso 'balde' intermediario entre as trocas, variavel auxiliar.
28             char aux;
29             //Outra auxiliar que serve para armazenar o numero antes da modificacao para
30             //se criar o 'Passo'
31             char[] antes;
32             //Variavel que sera incrementada a cada troca para contar quantas trocas houve
33             int qtdeTrocas = 0;
34             //Vetor de passos para descrever todo o processo
35             List<passos> passos = new ArrayList<passos>();
36             //Variavel que marca determina se houve trocas, usada para
37             //interromper o processo quando ja nao houver mais numeros a serem
38             //ordenados
39             boolean continua=true;
40
41             //Sera percorrido todos os numeros de acordo com o tamanho da sequencia
42             for (int i = 0; i < digitos.length; i++) {
43                if (!continua){ //Verificando se foram feitas trocas no ultimo ciclo, se nao
foram, indica que ja esta ordenado
44                   break; //Interrompe o algoritmo
45                }
46                //Descrevendo o passo
```

```
47                  passos.add(new Passos(null, null, "Inicio da verificacao numero
".concat(String.valueOf(i)).concat("\n----------\n")));
48                  continua=false;
49                  //Percorrendo cada numero com o seu
proximo
50                  for (int j = 0; j < digitos.length - 1; j++) {
51                      if (digitos[j] > digitos[j + 1]) {
52                          //Esse numero eh maior que o proximo, troca!
53                          antes = new String(digitos).toCharArray();
54                          aux = digitos[j];
55                          digitos[j] = digitos[j + 1];
56                          digitos[j + 1] = aux;
57                          //Incrementando a quantidade de trocas
58                          qtdeTrocas++;
59                           //Descrevendo o passo
                            passos.add(new Passos(new String(antes), newString(digitos), "Trocou-se
60 o digito ".concat(String.valueOf(digitos[j+1])).concat(" pelo
   ").concat(String.valueOf(digitos[j]))));
61                          continua=true;
62                      } else {
                            passos.add(new Passos(new String(digitos), newString(digitos), "Nao
63 houve troca pois o numero ".concat(String.valueOf(digitos[j])).concat(" ja eh menor/igual
   que ").concat(String.valueOf(digitos[j + 1]))));
64                       }
65                  }
66              }
67
68              //Persiste os resultados
69              Ordenacao ordenacao = new Ordenacao(numero, new String(digitos),
qtdeTrocas);
```

```
70                 new OrdenacaoDAO().salvar(ordenacao);
71                 new PassosDAO().salvarPassos(passos);
72                 //Retorno um objeto da classe Ordenacao informando os resultados.
73                 return ordenacao;
74
75             } catch (Exception ex)
    {
76                 ex.printStackTrace();
77                 return  null;
78             }
79         }
80     }
81 </passos></passos>
```

Pronto, sua aplicação já estará funcionando, mesmo que sem nenhuma interação com o usuário e nem interface gráfica, essa é a essência da cama ***model***, ser o **motor da aplicação!**

## Testando a model

Vamos testar nossa aplicação agora, para isso coloque a seguinte classe no pacote principal da aplicação (br.edu.qi):

```
01 package br.edu.qi;
02
03 import br.edu.qi.model.bo.OrdenacaoBO;
04 import br.edu.qi.model.vo.Ordenacao;
05
06 /**
07  *
08  * @author Gustavo Ferreira
09  */
10 public class MVC {
11
```

```
12     public static void main(String[] args) {
13         Ordenacao ordenacao = new OrdenacaoBO().bubbleSort(532135);
14         System.out.println(ordenacao); //Invoco o toString() da classe Ordenacao
15     }
16 }
```

Essa classe contém o método *main* que é o método principal e será nescessário daqui pra frente. Agora você já pode testar o sistema, embora não tenha ainda interface gráfica. Para interagir com a camada model, será necessário fazer alterações no código do método main.

### Interface gráfica

Nessa parte do projeto, iremos desenvolver a interface gráfica, mas apenas no que se refere aos componentes, nenhum tipo de comportamento ou interação com o usuário será implementado.

O sistema terá duas telas, uma principal onde será exibido o resumo do processo de ordenação e outra onde será configurado o número a ser ordenado. As duas telas irão se interagir em alguns pontos (vocês verão a importância da *controller* quando isso ocorrer). Segue abaixo o código da tela principal. Observando que, como o foco do artigo não é interface gráfica, as telas foram projetadas de forma simplificada para facilitar o entendimento.

```
001 package br.edu.qi.view;
002
003 import java.awt.Color;
004 import javax.swing.JButton;
005 import javax.swing.JFrame;
006 import javax.swing.JLabel;
007 import javax.swing.JScrollPane;
008 import javax.swing.JTextField;
009 import javax.swing.JTextPane;
010
011 /**
012  *
013  * @author Gustavo Ferreira
014  */
```

```
015 public class FramePrincipal extends JFrame {
016
017     private JLabel lbNumeroMaximo;
018     private JLabel lbNumeroGerado;
019     private JLabel lbNumeroOrdenado;
020     private JLabel lbQtdeTrocas;
021     private JTextField tfNumeroMaximo;
022     private JTextField tfNumeroGerado;
023     private JTextField tfNumeroOrdenado;
024     private JTextField tfQtdeTrocas;
025     private JTextPane tpInformacao;
026     private JScrollPane spRolagemPassos;
027     private JButton btGerarNumero;
028     private JButton btOrdernarNumero;
029
030     //Get and Set
031     public JButton getBtGerarNumero() {
032         return btGerarNumero;
033     }
034     public JButton getBtOrdernarNumero() {
035         return btOrdernarNumero;
036     }
037     public JTextField getTfNumeroGerado() {
038         return tfNumeroGerado;
039     }
040     public JTextField getTfNumeroMaximo() {
041         return tfNumeroMaximo;
```

```
042     }
043     public JTextField getTfNumeroOrdenado() {
044         return tfNumeroOrdenado;
045     }
046     public JTextField getTfQtdeTrocas() {
047         return tfQtdeTrocas;
048     }
049     public JTextPane getTpInformacao() {
050         return tpInformacao;
051     }
052     public JScrollPane getSpRolagemPassos() {
053         return spRolagemPassos;
054     }
055
056     public FramePrincipal() {
057
058         this.setTitle("MVC");
059         this.setDefaultCloseOperation(EXIT_ON_CLOSE);
060         this.setResizable(false);
061         this.setSize(380, 320);
062         this.setLocationRelativeTo(null);
063         this.setLayout(null);
064
065         this.lbNumeroMaximo = new JLabel("Numero maximo");
066         this.lbNumeroGerado = new JLabel("Numero gerado");
067         this.lbNumeroOrdenado = new JLabel("Numero ordenado");
068         this.lbQtdeTrocas = new JLabel("Qtde de trocas");
```

```
069
070         this.tfNumeroMaximo = new JTextField("1");
071         this.tfNumeroMaximo.setEnabled(false);
072         this.tfNumeroGerado = new JTextField();
073         this.tfNumeroGerado.setEnabled(false);
074         this.tfNumeroOrdenado = new JTextField();
075         this.tfNumeroOrdenado.setEnabled(false);
076         this.tfQtdeTrocas = new JTextField();
077         this.tfQtdeTrocas.setEnabled(false);
078
079         this.btGerarNumero = new JButton("Gerar numero randomico");
080         this.btOrdernarNumero = new JButton("Ordenar numero gerado");
081         this.btOrdernarNumero.setEnabled(false);
082
083         this.tpInformacao = new JTextPane();
          this.tpInformacao.setText("Esse sistema ordena uma sequencia numerica atravez do
084 algoritmo bolha (buble sort).\n\nEsse algoritmo realiza trocas entre um numero e o seu
      imediato caso este seja menor que o proximo.");
085         //Evita que o texto do TextPane seja selecionavel
086         this.tpInformacao.setEnabled(false);
087         //Define a cor do texto
088         this.tpInformacao.setDisabledTextColor(Color.BLACK);
089         //Tira o fundo branco padrao do TextPane
090         this.tpInformacao.setOpaque(false);
091
092         this.spRolagemPassos = new JScrollPane();
093         this.spRolagemPassos.setBorder(null);
094         this.spRolagemPassos.setViewportView(this.tpInformacao);
```

```
095
096         this.lbNumeroMaximo.setBounds(20, 20, 150, 20);
097         this.lbNumeroGerado.setBounds(20, 85, 150, 20);
098         this.lbNumeroOrdenado.setBounds(20, 150, 150, 20);
099         this.lbQtdeTrocas.setBounds(20, 215, 150, 20);
100
101         this.tfNumeroMaximo.setBounds(20, 50, 100, 25)
;
102         this.tfNumeroGerado.setBounds(20, 115, 100, 25);
103         this.tfNumeroOrdenado.setBounds(20, 180, 100, 25);
104         this.tfQtdeTrocas.setBounds(20, 245, 100, 25);
105
106         this.btGerarNumero.setBounds(150, 20, 200, 25);
107         this.btOrdernarNumero.setBounds(150,55, 200, 25);
108
109         this.spRolagemPassos.setBounds(150, 90, 200, 180);
110
111         this.add(lbNumeroMaximo);
112         this.add(lbNumeroGerado);
113         this.add(lbNumeroOrdenado);
114         this.add(lbQtdeTrocas);
115
116         this.add(tfNumeroMaximo);
117         this.add(tfNumeroGerado);
118         this.add(tfNumeroOrdenado);
119         this.add(tfQtdeTrocas);
120
121         this.add(btGerarNumero);
```

```
122         this.add(btOrdernarNumero);
123
124         this.add(spRolagemPassos);
125     }
126 }
```

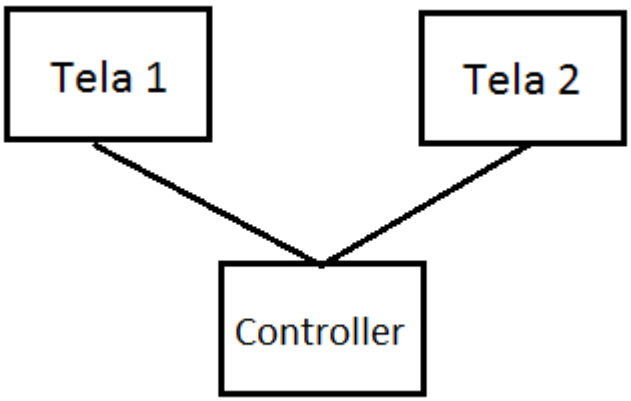
A outra tela, como dito anteriormente, será onde se configura o número a ser ordenado, segue abaixo.

```
01 package br.edu.qi.view;
02
03 import javax.swing.JButton;
04 import javax.swing.JDialog;
05 import javax.swing.JLabel
   ;
06 import javax.swing.JSlider;
07
08 /**
09  *
10  * @author Gustavo Ferreira
11  */
12 public class DialogGerarNumero extends JDialog{
13
14     private JLabel lbNumeroMaximo;
15     private JButton btGerar;
16     private JButton btGerarOrdenar;
17     private JSlider slNumeroMaximo;
18
19     //Get and Set
20     public JButton getBtGerar() {
```

```
21      return btGerar;
22  }
23  public JButton getBtGerarOrdenar() {
24      return btGerarOrdenar;
25  }
26  public JSlider getSlNumeroMaximo() {
27      return slNumeroMaximo;
28  }
29
30  public DialogGerarNumero() {
31      this.setSize(300, 205);
32      this.setLocationRelativeTo(null);
33      this.setTitle("Gerar numero randomico");
34      this.setModal(true);
35      this.setLayout(null);
36      this.setResizable(false);
37      this.setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
38
39      this.lbNumeroMaximo = new JLabel("Numero maximo da sequencia");
40
41      this.slNumeroMaximo = new JSlider(1, 99999);
42      this.slNumeroMaximo.setMajorTickSpacing(9999);
43      this.slNumeroMaximo.setPaintTicks(true);
44
45      this.btGerar = new JButton("Gerar numero");
46      this.btGerarOrdenar = new JButton("Gerar e ordenar");
47
```

```
48        this.lbNumeroMaximo.setBounds(20, 20, 200, 20);
49        this.slNumeroMaximo.setBounds(20, 50, 245, 30);
50        this.btGerar.setBounds(70, 90, 150, 25);
51        this.btGerarOrdenar.setBounds(70, 125, 150, 25);
52
53        this.add(lbNumeroMaximo);
54        this.add(slNumeroMaximo);
55        this.add(btGerar);
56        this.add(btGerarOrdenar);
57    }
58 }
```

## Controller




Dependência da controller

Agora é hora de implmentar a ***controller***. Se você copiou as classes da interface gráfica e colou ao seu projeto e configurou o método *main*, deve ter notado que só a tela principal abriu e ainda sim não foi possível interarir por falta de uma classe controladora de eventos, a camada ***controller***.
Como as duas telas irão se interagir e ambas copartilharão de alguns métodos, o correto é fazer com que elas usem a mesma ***controller***, pois assim a interação de uma interferirá na outra.

```
001 package br.edu.qi.controller;
002
003 import br.edu.qi.model.bo.OrdenacaoBO;
004 import br.edu.qi.model.vo.Ordenacao;
```

```
005 import br.edu.qi.view.DialogGerarNumero;
006 import br.edu.qi.view.FramePrincipal;
007 import java.awt.event.ActionEvent;
008 import java.awt.event.ActionListener;
009 import javax.swing.event.ChangeEvent;
010 import javax.swing.event.ChangeListener;
011
012 /**
013  * @author Gustavo Ferreira
014  * @see Classe que cria objeto de controle entre a camada Model e View
015  */
016 public class ControllerPrincipal implements ActionListener, ChangeListener {
017
018     private FramePrincipal framePrincipal;
019     private DialogGerarNumero dialogGerarNumero;
020
021     /**
022      * Construtor<br>Recebe o objeto da FramePrincipal para 'observer' seu
    comportamento,
023      * tratar os eventos e redirecionar para a model.
024      * @param framePrincipal
025      */
026     public ControllerPrincipal(FramePrincipal framePrincipal) {
027         this.framePrincipal = framePrincipal;
028         //Definindo os listeners para os botoes dessa view.
029          this.framePrincipal.getBtGerarNumero().addActionListener(this);
030         this.framePrincipal.getBtOrdernarNumero().addActionListener(this);
031     }
```

```
032
033     //Evento de acao, pressionar um botao ou um [Enter] em inputs
034     @Override
035     public void actionPerformed(ActionEvent e) {
036         /**
037          * Se for o pressionar do botao 'Gerar Numero' da FrameMain, instancia uma
038          * DialogGerarNumero
039          */
040         if (e.getSource() == this.framePrincipal.getBtGerarNumero()) {
041
042             //Instanciando a DialogGerarNumero
043             this.dialogGerarNumero = new DialogGerarNumero();
044             this.dialogGerarNumero.getSlNumeroMaximo().setValue(
    Integer.parseInt(this.framePrincipal.getTfNumeroMaximo().getText()));
045
046             //Registrando os listeners do Dialog
047             this.dialogGerarNumero.getBtGerar().addActionListener(this);
048             this.dialogGerarNumero.getBtGerarOrdenar().addActionListener(this);
049             this.dialogGerarNumero.getSlNumeroMaximo().addChangeListener(this);
050             this.dialogGerarNumero.setVisible(true);
051             //Destruo o Dialog
052             this.dialogGerarNumero = null;
053
054         } else if (this.dialogGerarNumero != null) {
055             //Eventos do DialogGerarNumero
056             if (e.getSource() == this.dialogGerarNumero.getBtGerar()) {
057                 gerarNumero();
058             }
```

```
059             else if (e.getSource() == this.dialogGerarNumero.getBtGerarOrdenar()){
060                 gerarNumero();
061                 ordenarNumero();
062             }
063         }
064         else if (e.getSource() == this.framePrincipal.getBtOrdernarNumero()){
065             ordenarNumero();
066         }
067     }
068 
069     @Override
070     public void stateChanged(ChangeEvent e) {
071         if (this.dialogGerarNumero != null) {
072             /**
073              * A medida que o Slider eh arrastado, o campo equivalente na FramePrincipal
074              * tem seu valor alterado
075              */
076             if (e.getSource() == this.dialogGerarNumero.getSlNumeroMaximo()) {
077                 this.framePrincipal.getTfNumeroMaximo().setText(
String.valueOf(this.dialogGerarNumero.getSlNumeroMaximo().getValue()));
078             }
079         }
080     }
081 
082     /**
083      * Metodo responsavel em controlar a acao de ordenacao. Redireciona para a Model
084      */
085     private void ordenarNumero(){
```

```
086
087         //Manda ordenar e recebe uma Ordenacao como resultado do processo.
088         Ordenacao ordenacao = new OrdenacaoBO().bubbleSort(
Integer.parseInt(this.framePrincipal.getTfNumeroGerado().getText()));
089
090         //Atualiza a view com o resultado
091         this.framePrincipal.getTfNumeroOrdenado().setText(
String.valueOf(ordenacao.getNumeroOrdenado()));
092         this.framePrincipal.getTfQtdeTrocas().setText(
String.valueOf(ordenacao.getQtdeTrocas()));
093     }
094     /*
*
095      * Metodo que limpa os campos que contem valores resultados de uma ordenacao
096      */
097     private void limparDadosOrdenacaoAnterior() {
098         this.framePrincipal.getTfNumeroOrdenado().setText(null);
099         this.framePrincipal.getTfQtdeTrocas().setText(null);
100     }
101
102     /**
103      * Metodo que gera um numero randomico e atualiza a view.<br>
104      * Executa logo apos o pressionar do botao 'Gerar' do DialogGerarNumero
105      */
106     private void gerarNumero() {
107         limparDadosOrdenacaoAnterior();
108
109         //Fecha o DialogGerarNumero
110         this.dialogGerarNumero.setVisible(false);
```

```
111
112         //Atualiza o numero na FramePrincipal
113         this.framePrincipal.getTfNumeroGerado().setText(String.valueOf((int)
(Math.random() * this.dialogGerarNumero.getSlNumeroMaximo().getValue())));
114
115         //Destroi a DialogGerarNumero
116         this.dialogGerarNumero = null;
117
118         //Habilita o botao 'Ordenar' da FramePrincipal
119         this.framePrincipal.getBtOrdernarNumero().setEnabled(true);
120     }
121 }
```

Você deve ter percebido que quando implementamos somente a camada ***model*** não havia erros de compilação e o sistema até funcionava, mostrando que a ***model*** não dependia essencialmente de mais nada para funcionar. Também deve ter percebido que quando implementou a ***view***, não havia erros de compilação mostrando que a camada ***view*** desconhecia todo o resto e poderia ser encaixado em qualquer sistema que precisasse de uma ***view*** como aquela. Já no caso da ***controller***, esta sim está amarrado com tudo. Sozinha a ***controller*** não é nada e dá erros de compilação na falta do resto pois, ela tem o papel de "juntar as partes". Outra coisa interessante da controller é que é ela a responsável em tratar os erros de execução, nenhuma classe da model trata os tais erros.
Segue o link para baixar o código fonte e o sistema já compilado, fiquem a vontade para comentar abaixo.



**Código fonte**

**Executável**