Apresentando Model-View-Presenter, o MVC focado na visualização

Publicado por Tutoriais Admin em 04/10/2012 - 206.118 visualizações

comentários: 1

Daniel Fernandes Martins

dfmwork @ gmail.com

Neste artigo serão abordados os aspectos principais do padrão MVP, ou Model-View-Presenter. Serão explicados quais problemas motivaram a criação desse padrão, como ele resolve tais problemas e, principalmente, os as vantagens e desvantagens de se usar o MVP. Para demonstrar esses conceitos de forma prática, nós desenvolveremos uma aplicação de exemplo utilizando Swing.

O modelo de programação MVP é o que podemos chamar de uma derivação do modelo MVC, que surgiu com o Smalltalk. No Smalltalk, os componentes visuais, como as caixas de texto e botões, são regidos por três abstrações centrais: Model, View e Controller.

- Model: São informações que indicam o estado do componente, como, por exemplo, o texto de um TextField ou a indicação on-off de um CheckBox;
- View: Acessa os dados do Model e especifica como os dados do Model são mostrados ao usuário, como, por exemplo, um texto dentro de um TextBox ou um & # 61692; indicando que um CheckBox está marcado;
- Controller: Componente para mapear as ações do usuário na View (as quais ocorrem normalmente através de eventos) e fazem com que o Model seja modificado. Para citar um exemplo, quando um CheckBox? marcado? recebe um evento de click, o Controller mapeia essa ação do usuário e modifica o Model, indicando que este agora está desmarcado. O Model, por sua vez, notifica a View, indicando mudança em seu estado. A View recebe a notificação e renderiza o CheckBox desmarcado na tela.

A Figura 1 ilustra a interação entre o Model, a View e o Controller:

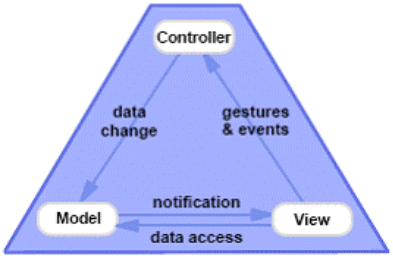

Figura 1? Modelo MVC

Este modelo de programação funciona muito bem no contexto de componentes individuais, como explicado anteriormente. Porém, durante o desenvolvimento, é necessário agrupar vários desses componentes sob contextos específicos que, juntos, representam o domínio do problema a ser solucionado pelo sistema.

Imagine uma janela que sirva para cadastrar clientes no sistema. É agrupado um conjunto de diversos tipos de controles visuais (View) que representam uma lista de objetos Cliente no sistema (Model). O resultado da interação do usuário com a janela é traduzido em eventos (Controller), que controlam o fluxo da aplicação e modificam o estado dos controles da janela, atualizando também os objetos de negócio.

Na maioria dos casos, o código responsável pelo tratamento dos eventos e controle de fluxo da Janela fica dentro de uma mesma classe. Temos então uma quebra de camadas: a View não deve conhecer nem o modelo que representa e nem a lógica de apresentação, já que temos alguns tipos de ação na lógica de apresentação que são regidas pelo domínio do problema em questão. Outro problema de manter tanto o código de montagem da tela quanto o tratamento de eventos é uma classe extremamente grande, difícil de manter, difícil de expandir e com código de impossível reutilização.

É aí que entra o MVP!

Motivações do MVP

A diferença entre o MVC e o MVP fica basicamente no conceito, já que as funções do? C? (Controller) do MVC são semelhantes ao? P? (Presenter) do MVP.

Imagine novamente o exemplo do cadastro de clientes. O Model é a coleção de objetos Cliente que são manipulados pela janela. A View é a janela propriamente dita. O Presenter é responsável por interceptar os eventos gerados na View, com a finalidade de controlar a lógica de apresentação.

Portanto, foco principal do MVP é separar a lógica de apresentação da apresentação em si. Com isso, conseguimos alternar entre diferentes apresentações facilmente, através da reutilização da lógica de apresentação. Além disso, conseguimos realizar testes na classe responsável pela lógica de apresentação sem precisar utilizar a View para isso. Ganhamos também no quesito manutenção, já que as responsabilidades foram divididas em mais classes especializadas e fáceis de entender.

MVP na Prática

Já que o? coração? do MVP é a separação da lógica de apresentação da apresentação em si, temos como resultado uma View que não contém muito código além

necessário para organizar os componentes na tela. Isso parece ser o caminho mais correto, pois a função da View deve ser somente oferecer uma forma de o usuário interagir com o sistema. Note aqui uma semelhança com o padrão MVC clássico.

Considere o seguinte diagrama de classes:

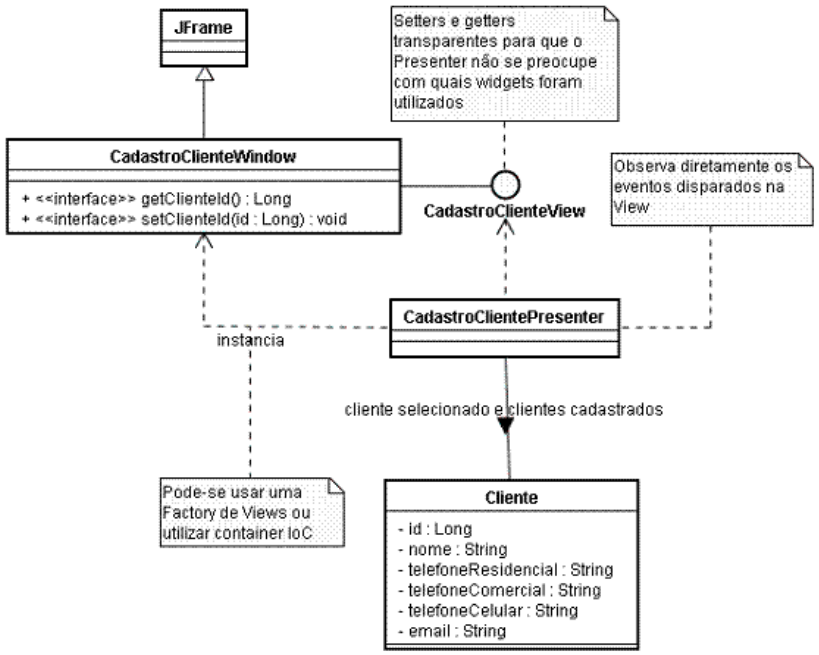

Figura 2? Estrutura de classes do exemplo, seguindo o MVP.

Para criar a nossa aplicação de exemplo, utilizaremos como base o diagrama de classes mostrado na Figura 2.

Todas as Views capazes de cadastrar clientes devem implementar a interface CadastroClienteView. Essa interface define métodos get () e set () para configurar os valores dos componentes, habilitar / desabilitar entrada de dados nos componentes visuais e quaisquer outras operações que sejam necessárias. Essas operações são utilizadas pelo Presenter, que no caso é a classe CadastroClientePresenter. Como mostrado na Figura 2, a classe CadastroClientePresenter se registra como observador dos eventos disparados na View, e os trata.

Para ilustrar melhor a interação entre as classes definidas na Figura 2, considere o seguinte diagrama de seqüência:

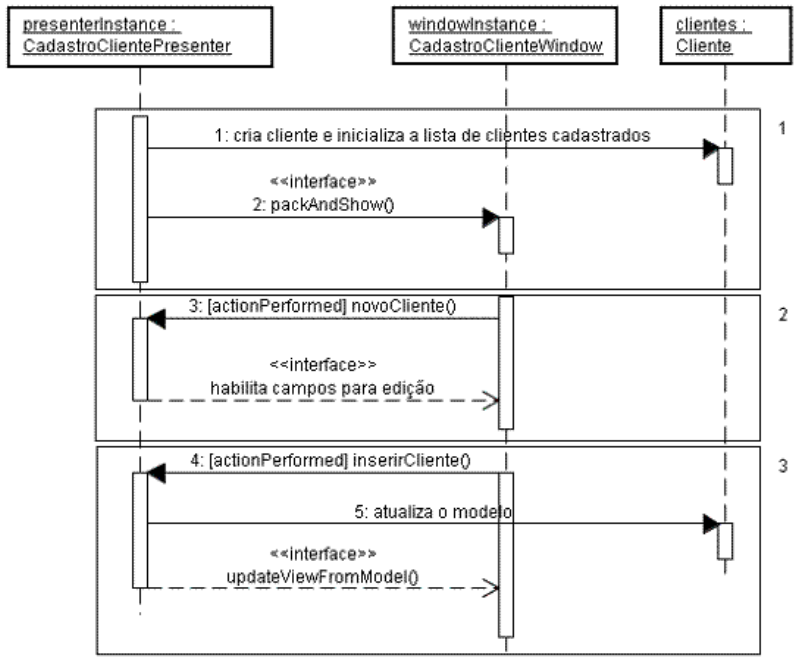

Figura 3? Procedimento de inserção de um novo cliente no sistema

Note que o diagrama está dividido em três partes. A primeira parte mostra como ocorre a instanciação da janela de cadastro de clientes: um objeto presenterInstance obtém o objeto windowInstance através de criação explícita (com o operador new) ou injetado via IoC. Em seguida, o objeto presenterInstance inicializa o Model, que no caso é composto por uma coleção de objetos Cliente. Finalmente, o objeto presenterInstance exibe a janela. (Parte 1)

A janela fica aguardando o usuário inserir um novo cliente. Quando o usuário clica no botão? Inserir?, o objeto presenterInstance executa o código responsável por liberar os campos para que o usuário digite os dados do novo cliente, através de chamadas aos métodos definidos na interface CadastroClienteView, que é implementada pelo objeto windowInstance. (Parte 2)

Para confirmar o cadastro, o usuário pressiona o botão? Confirmar?. Em seguida, o objeto presenterInstance cria uma nova instância da classe Cliente e preenche seus atributos, de acordo com os valores digitados na janela. Depois, o Presenter adiciona esse cliente na lista de clientes cadastrados e atualiza a View. (Parte 3)

Para aplicar os conceitos vistos até aqui, vamos criar uma aplicação de exemplo utilizando o pattern MVP.

Aplicativo de Demonstração

As imagens abaixo demonstram o nosso aplicativo em funcionamento:

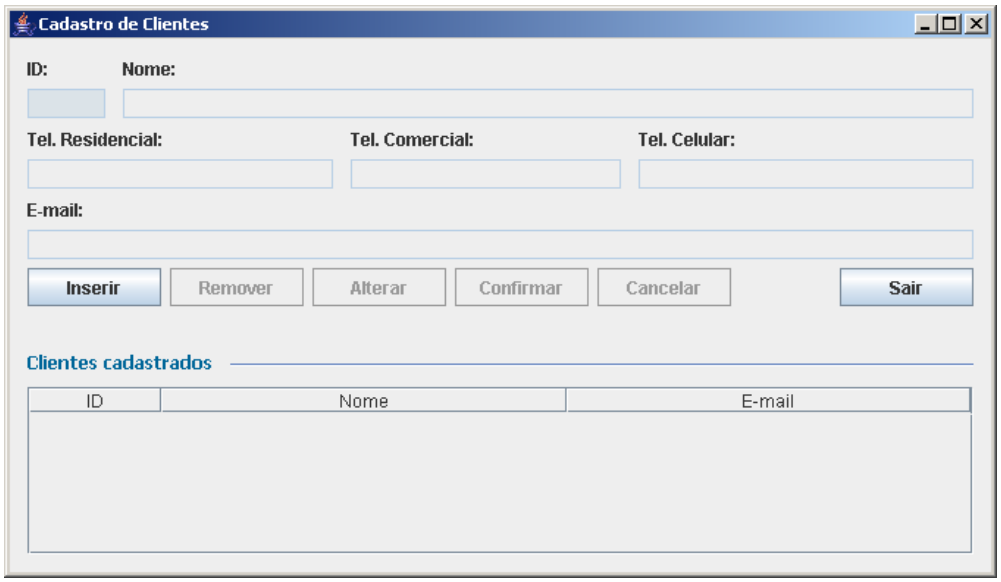

Figura 4? Janela de cadastro de clientes

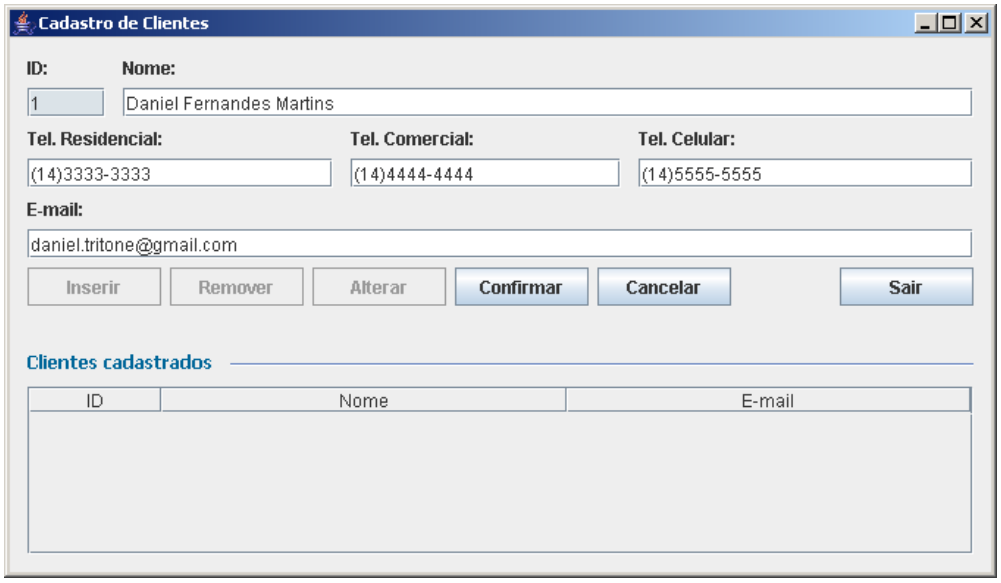

Figura 5? Inserindo um novo cliente

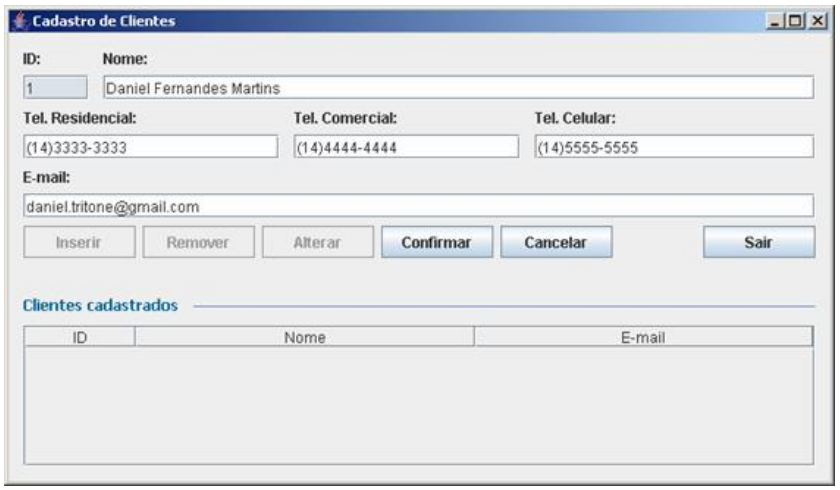

Figura 6? Novo cliente cadastrado no sistema

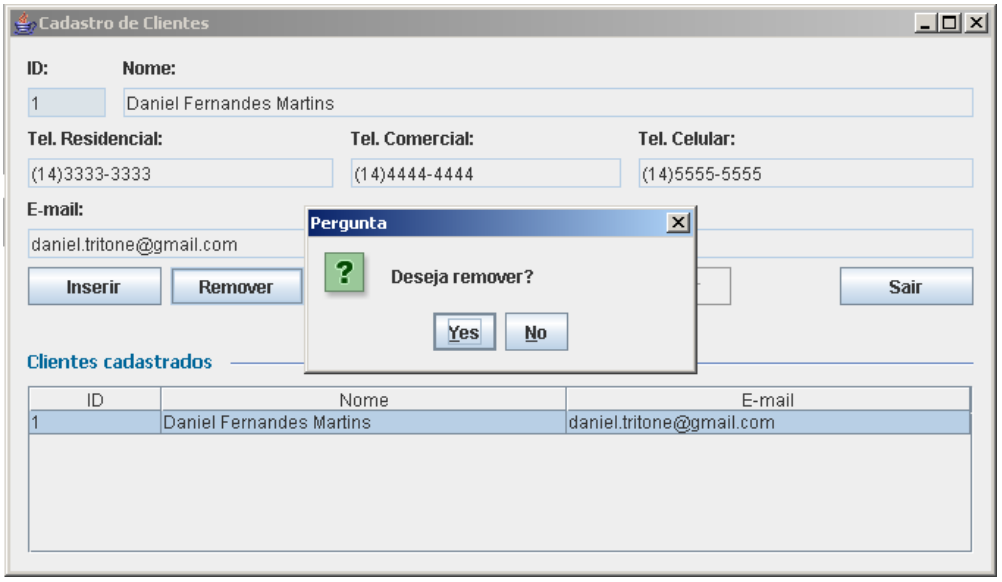

Figura 7? Removendo um cliente cadastrado

A aplicação de exemplo é bastante simples, mas mostra como aplicar o pattern e como utilizá-lo para tornar o desenvolvimento de aplicações rich-client menos traumático.

Ferramentas utilizadas

Para criação desse exemplo, aconselho a utilização da IDE NetBeans, devido a grande facilidade de criação de interfaces gráficas com Swing. Neste programa também usaremos o framework Spring para? colar? os componentes da aplicação sem que seja necessário escrever código para isso.

Criando o Model

Inicie um novo projeto no NetBeans com qualquer nome. Em seguida, crie uma classe chamada Cliente, no pacote org.javafree.mvp.model. Essa classe será o nosso Model.

Cliente.java

```
package org.javafree.mvp.model;

public class Cliente {

    private Long id;
    private String nome;
    private String telefoneResidencial;
    private String telefoneComercial;
    private String telefoneCelular;
    private String email;

    public Cliente() {}

    public Cliente(Long id, String nome, String telefoneResidencial, String telefoneComercial, String telefoneCelular, String email)
        setId(id);
        setNome(nome);
        setTelefoneResidencial(telefoneResidencial);
        setTelefoneComercial(telefoneComercial);
        setTelefoneCelular(telefoneCelular);
        setEmail(email);
    }

    // Métodos Get e Set


    public Object clone() {
        return new Cliente(this.id, this.nome, this.telefoneResidencial, this.telefoneComercial, this.telefoneCelular, this.email);
    }

    public boolean equals(Object obj) {
        if (obj instanceof Cliente) {
            Cliente c = (Cliente)obj;

            return c.getId().equals(id);
        }

        return false;
    }
}
```

Criando a View

Crie um novo JFrame chamado CadastroClienteWindow, no pacote org.javafree.mvp.gui.cadastro .

Crie um layout parecido com a figura abaixo:

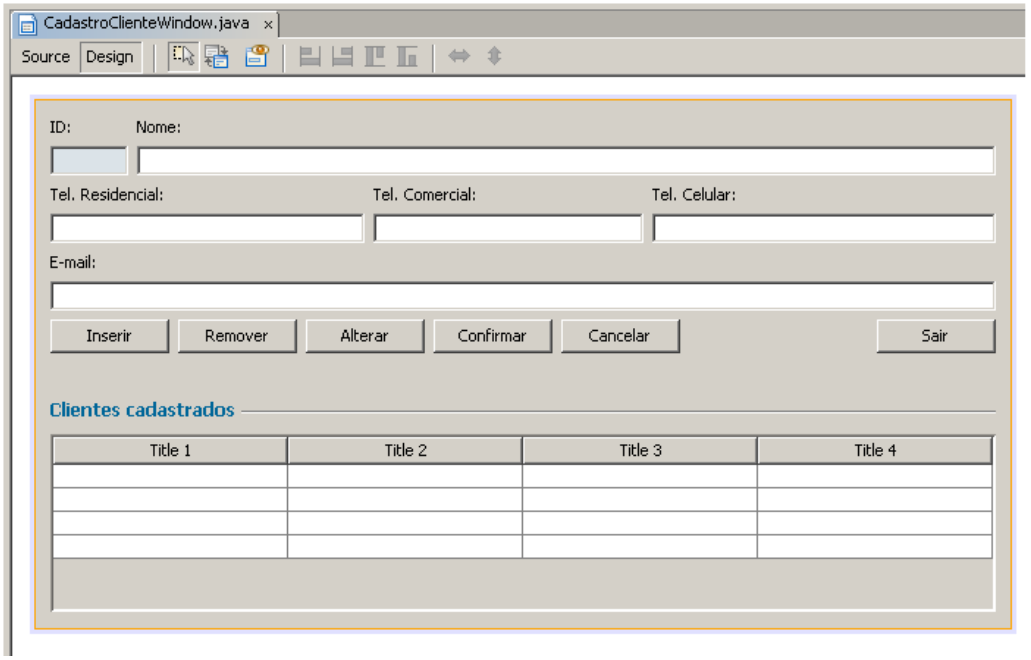

Figura 8 ? Layout da aplicação

Esta interface é composta por sete JLabel, seis JTextField, seis JButton, um JScrollPane e um JTable:

Classe	Objetos
JLabel	lbClienteId, lbClienteNome, lbClienteTelResidencial, lbClienteTelComercial, lbClienteTelCelular, lbClienteEmail, lbClientesCadastrados
JTextField	txtClienteId, txtClienteNome, txtClienteTelResidencial, txtClienteTelComercial, txtClienteTelCelular, txtClienteEmail
JButton	btnInserir, btnRemover, btnAlterar, btnConfirmar, btnCancelar, btnSair
JScrollPane	spClientesCadastrados
JTable	tableClientesCadastrados

Para permitir que o Presenter interaja com a View sem saber o seu tipo, precisamos criar uma interface, que deve ser implementada pela nossa View. Assim, nós não estaremos prendendo o Presenter a uma única View.

Ainda no pacote org.javafree.mvp.gui.cadastro, crie uma classe chamada CadastroClienteView.

CadastroClienteView.java

```
package org.javafree.mvp.gui.cadastro;

public interface CadastroClienteView {

    // Gets e Sets dos atributos que representam o model
    public void setId(Long id);
    public Long getId();

    public void setNome(String nome);
    public String getNome();

    // ...

    public void packAndShow();
    public void clearFields();

    public int linhaSelecionadaTableClientes();

    public void enableTxtClienteId(boolean arg);
    public void enableTxtClienteNome(boolean arg);

    // ...

    public void setClientesTableModel(ClientesTableModel model);
    public ClientesTableModel getClientesTableModel();

    public void refreshTableClientes();

    // Métodos para configurar os listeners que compõe a lógica de apresentação
    public void setInserirActionListener(ActionListener listener);
    public void setRemoverActionListener(ActionListener listener);
    public void setAlterarActionListener(ActionListener listener);

    // ...

}
```

Modifique a classe CadastroClienteWindow, fazendo que ela implemente a interface CadastroClienteView. Adicione o seguinte código na classe CadastroClienteWindow:

```
public class CadastroClienteWindow implements CadastroClienteView {

    // ...

    public void setId(Long id) {
        this.txtClienteId.setText(id.toString());
    }

    public Long getId() {
        return new Long(this.txtClienteId.getText());
    }

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

    public String getNome() {
        return this.txtClienteNome.getText();
    }

    // ...

    public void enableTxtClienteId(boolean arg) {
        this.txtClienteId.setEditable(arg);
    }

    public void enableTxtClienteNome(boolean arg) {
        this.txtClienteNome.setEditable(arg);
    }

    // ...

    public int linhaSelecionadaTableClientes() {
        return this.tableClientesCadastrados.getSelectedRow();
    }

    public void packAndShow() {
        this.pack();
        this.setVisible(true);
    }

    public void clearFields() {
        Component[] componentes = this.getContentPane().getComponents();

        for (int i = 0; i < componentes.length; i++) {
            if (componentes[i] instanceof JTextField) {
                JTextField field = (JTextField)componentes[i];
                field.setText("");
            }
        }
    }

    public void setClientesTableModel(ClientesTableModel model) {
        this.tableClientesCadastrados.setModel(model);

        // Configura as colunas da JTable
        TableColumnModel columnModel = this.tableClientesCadastrados.getColumnModel();

        columnModel.getColumn(0).setHeaderValue("ID");
        columnModel.getColumn(0).setPreferredWidth(10);

        columnModel.getColumn(1).setHeaderValue("Nome");
        columnModel.getColumn(1).setPreferredWidth(200);

        columnModel.getColumn(2).setHeaderValue("E-mail");
        columnModel.getColumn(2).setPreferredWidth(200);
    }

    public ClientesTableModel getClientesTableModel() {
        return (ClientesTableModel)this.tableClientesCadastrados.getModel();
    }

    public void refreshTableClientes() {
        this.tableClientesCadastrados.updateUI();
    }

    public void setInserirActionListener(ActionListener listener) {
        this.btnInserir.addActionListener(listener);
    }

    public void setRemoverActionListener(ActionListener listener) {
        this.btnRemover.addActionListener(listener);
    }

    public void setAlterarActionListener(ActionListener listener) {
        this.btnAlterar.addActionListener(listener);
    }

    // ...
}
```

Ainda no pacote org.javafree.mvp.gui.cadastro, crie uma nova classe chamada ClientesTableModel. Esta classe será necessária para renderizar o JTable com a

lista de todos os clientes cadastrados no sistema.

ClientesTableModel.java

```
package org.javafree.mvp.gui.cadastro;

public class ClientesTableModel extends AbstractTableModel {

    private List clientes;

    public ClientesTableModel(List clientes) {
        this.clientes = clientes;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        Cliente cliente = clientes.get(rowIndex);

        if (cliente != null) {

            switch (columnIndex) {
                case 0: return cliente.getId();
                case 1: return cliente.getNome();
                case 2: return cliente.getEmail();
            }
        }

        return null;
    }

    public int getRowCount() {
        if (clientes != null) {
            return this.clientes.size();
        }

        return 0;
    }

    public int getColumnCount() {
        // Id, Nome, E-mail
        return 3;
    }

    public Cliente getCliente(int row) {
        if (row >= 0) {
            return this.clientes.get(row);
        }

        return null;
    }
}
```

Criando o Presenter

Agora definiremos a classe Presenter, que será responsável pela lógica de apresentação do nosso aplicativo. Crie uma classe chamada CadastroClientePresenter, no pacote org.javafree.mvp.presenter.cadastro .

CadastroClientePresenter.java

```
 package org.javafree.mvp.presenter.cadastro;

public class CadastroClientePresenter {

    private Cliente cliente;
    private List model;


    private CadastroClienteView view;


    public void createView() {

        this.novoCliente();

        this.setUpViewListeners();
        this.habilitarEdicao(false);

        // ...

        view.packAndShow();
    }

    public void setUpViewListeners() {
        // Implementado posteriormente
    }

    public void updateModelFromView() {
        cliente.setId(view.getId());
        cliente.setNome(view.getNome());
```

```
        // ...
    }

    public void updateModelFromJTable() {
        ClientesTableModel tbModel = view.getClientesTableModel();

        cliente = tbModel.getCliente(view.linhaSelecionadaTableClientes());

        if (cliente != null) {
            this.updateViewFromModel();

            view.enableBtnAlterar(true);
            view.enableBtnRemover(true);
        }
        else {
            view.enableBtnAlterar(false);
            view.enableBtnRemover(false);
        }

        this.updateViewFromModel();
    }

    public void updateViewFromModel() {

        if (cliente != null) {
            view.setId(cliente.getId());
            view.setNome(cliente.getNome());
            view.setTelefoneResidencial(cliente.getTelefoneResidencial());
            view.setTelefoneComercial(cliente.getTelefoneComercial());
            view.setTelefoneCelular(cliente.getTelefoneCelular());
            view.setEmail(cliente.getEmail());
        }
        else {
            view.clearFields();
        }
    }

    public void novoCliente() {
        cliente = new Cliente();
    }

    public void adicionarCliente() {
        this.updateModelFromView();
        this.model.add((Cliente)cliente.clone());
        view.refreshTableClientes();
    }

    public void alterarCliente() {
        this.updateModelFromView();
        view.refreshTableClientes();
    }

    public boolean removerCliente() {
        this.updateModelFromView();
        boolean b = this.model.remove(cliente);
        view.refreshTableClientes();

        return b;
    }

    public void habilitarEdicao(boolean arg) {
        view.enableTxtClienteId(arg);
        view.enableTxtClienteNome(arg);
        view.enableTxtClienteTelResidencial(arg);
        view.enableTxtClienteTelComercial(arg);
        view.enableTxtClienteTelCelular(arg);
        view.enableTxtClienteEmail(arg);
    }

    public CadastroClienteView getView() {
        return view;
    }

    public void setView(CadastroClienteView view) {
        this.view = (CadastroClienteView)view;
    }

    public Object getModel() {
        return model;
    }

    public void setModel(Object model) {
        if (model instanceof List) {
            this.model = (List)model;
        }
    }
}
```

Para definir os eventos disparados pela View, criaremos duas classes no pacote org.javafree.mvp.presenter.cadastro: ClientesWindowActionListeners e ClientesWindowMouseListener.

A classe ClientesWindowActionListener contém todas as classes necessárias para tratar os eventos de ação na View, enquanto a classe ClientesWindowMouseListener trata os eventos de mouse. Segue abaixo o código das duas classes:

ClientesWindowActionListeners.java

```
package org.javafree.mvp.presenter.cadastro;

public class ClientesWindowActionListeners {

    static class AlterarActionListener implements ActionListener {
        private CadastroClientePresenter presenter;

        public AlterarActionListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }

        public void actionPerformed(ActionEvent evt) {

            presenter.habilitarEdicao(true);

            CadastroClienteView view = (CadastroClienteView)presenter.getView();

            view.enableBtnCancelar(true);
            view.enableBtnConfirmar(true);

            // ...
        }
    }

    static class CancelarActionListener implements ActionListener {
        private CadastroClientePresenter presenter;

        public CancelarActionListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }

        public void actionPerformed(ActionEvent evt) {

            CadastroClienteView view = (CadastroClienteView)presenter.getView();

            view.clearFields();

            presenter.habilitarEdicao(false);

            view.enableBtnCancelar(false);
            view.enableBtnConfirmar(false);

            // ...
        }
    }

    static class ConfirmarActionListener implements ActionListener {
        private CadastroClientePresenter presenter;

        public ConfirmarActionListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }

        public void actionPerformed(ActionEvent evt) {

            presenter.habilitarEdicao(false);

            CadastroClienteView view = (CadastroClienteView)presenter.getView();

            view.enableBtnCancelar(false);
            view.enableBtnConfirmar(false);

            // ...
        }
    }

    static class InserirActionListener implements ActionListener {
        private CadastroClientePresenter presenter;

        public InserirActionListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }

        public void actionPerformed(ActionEvent evt) {

            presenter.novoCliente();

            CadastroClienteView view = (CadastroClienteView)presenter.getView();

            presenter.habilitarEdicao(true);
            view.clearFields();

            view.enableBtnAlterar(false);
            view.enableBtnCancelar(true);

            // ...
        }
    }

    static class RemoverActionListener implements ActionListener {
        private CadastroClientePresenter presenter;

        public RemoverActionListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }
```

```
        public void actionPerformed(ActionEvent evt) {

            if (JOptionPane.showConfirmDialog((Component)presenter.getView(), "Deseja remover?", "Pergunta", JOptionPane.YES_NO_OPTI

                CadastroClienteView view = (CadastroClienteView)presenter.getView();

                presenter.removerCliente();
                view.clearFields();

                view.enableBtnAlterar(false);
                view.enableBtnRemover(false);
            }
        }
    }

    static class SairActionListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            System.exit(0);
        }
    }
}
```

ClientesWindowMouseListener.java

```
 package org.javafree.mvp.presenter.cadastro;

public class ClientesWindowMouseListener {

    static class TableClientesMouseListener extends MouseAdapter {
        private CadastroClientePresenter presenter;

        public TableClientesMouseListener(CadastroClientePresenter presenter) {
            this.presenter = presenter;
        }

        public void mouseClicked(MouseEvent e) {

            presenter.updateModelFromJTable();

            CadastroClienteView view = (CadastroClienteView)presenter.getView();

            view.enableBtnCancelar(false);
            view.enableBtnConfirmar(false);
            view.enableBtnInserir(true);
        }
    }
}
```

A criação da camada de lógica de apresentação está quase concluída. Agora que temos as classes que tratam os eventos da View, vamos voltar à classe CadastroClientePresenter e implementar o método setUpViewListeners () como mostrado abaixo:

```
 public void setUpViewListeners() {
        view.setInserirActionListener(new InserirActionListener(this));
        view.setRemoverActionListener(new RemoverActionListener(this));
        view.setAlterarActionListener(new AlterarActionListener(this));
        view.setCancelarActionListener(new CancelarActionListener(this));
        view.setConfirmarActionListener(new ConfirmarActionListener(this));
        view.setBtnSairActionListener(new SairActionListener());
        view.setTableClientesMouseListener(new TableClientesMouseListener(this));
    }
```

Para concluir o Presenter, nós precisamos definir o fluxo de operações do nosso aplicativo. Veja a imagem

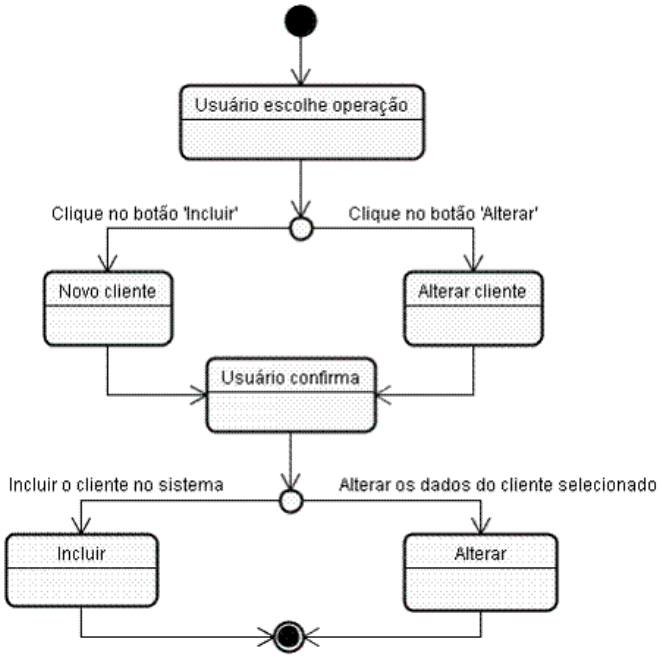

Figura 9 ? Fluxo dos comandos de Incluir e Alterar clientes

Como mostrado na imagem, a operação de confirmação (que é feita em um único local), deve saber se o usuário está inserindo um cliente no sistema ou se está modificando um cliente já cadastrado.

Para evitar testes do tipo

```
if (operacao == inserir) {
    // Faça uma coisa
}
else if (operação == alterar) {
    // Faça outra coisa
}
else if (etc...)
```

vamos implementar esta funcionalidade utilizando o padrão Strategy. Assim, quando o usuário pressionar o botão Inserir, a ? estratégia ? executada pelo botão Confirmar é inserir o cliente no sistema. Da mesma forma, se o usuário decidir alterar um cliente, a estratégia do botão Confirmar será alterar os dados do cliente.

Para conseguir esse efeito, vamos criar uma interface chamada Strategy, dentro do pacote org.javafree.mvp.presenter.cadastro.

Strategy.java

```
package org.javafree.mvp.presenter.cadastro;

public interface Strategy {
    public void execute();
}
```

Volte à classe CadastroClientePresenter e insira, no final do código da classe, o seguinte código:

```
     private class InsertStrategy implements Strategy {
        public void execute() {
            adicionarCliente();
        }
    }

    private class UpdateStrategy implements Strategy {
        public void execute() {
            alterarCliente();
        }
    }
```

Agora, vamos criar dois objetos que representam essas operações. Estes dois objetos devem estar visíveis para as classes tratadoras de eventos, permitindo a definição do fluxo da aplicação.

```
// Constantes de indicação de operação, para controlar o que o botão 'confirmar' faz
    public final Strategy INSERT_STRATEGY = new InsertStrategy();
    public final Strategy UPDATE_STRATEGY = new UpdateStrategy();

    private Strategy operacao = INSERT_STRATEGY;

    // ...

    public void setOperacao(Strategy operacao) {
        this.operacao = operacao;
    }

    public Strategy getOperacao() {
        return operacao;
    }
```

Estamos quase terminando o nosso Presenter. Abra novamente a classe ClientesWindowActionListeners e modifique o código, como mostrado no trecho abaixo:

```
    // ?
    static class AlterarActionListener implements ActionListener {
        // ...

        public void actionPerformed(ActionEvent evt) {
            presenter.setOperacao(presenter.UPDATE_STRATEGY);
   // ...
        }
    }

    // ...
    static class ConfirmarActionListener implements ActionListener {
        // ...
        public void actionPerformed(ActionEvent evt) {
            presenter.getOperacao().execute();
            // ...
        }
    }

    // ...
    static class InserirActionListener implements ActionListener {
        // ...
        public void actionPerformed(ActionEvent evt) {
            presenter.setOperacao(presenter.INSERT_STRATEGY);
            // ...
        }
    }
```

Juntando as partes

A nossa aplicação está praticamente finalizada. Só resta agora criar o arquivo de configuração do Spring. Crie um arquivo chamado applicationContext.xml no CLASSPATH da aplicação. Configure seu conteúdo para a listagem exibida abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <!-- Model da aplicação, que contém uma coleção de Clientes -->
    <bean id="model" class="java.util.ArrayList" singleton="true" />

    <!-- Classe que encapsula a lógica de apresentação -->
    <bean id="presenter" class="org.javafree.mvp.presenter.cadastro.CadastroClientePresenter" singleton="false">
        <property name="view">
            <ref local="view" />
        </property>
        <property name="model">
            <ref local="model" />
        </property>
    </bean>

    <!-- Classe responsável por renderizar os dados graficamente -->
    <bean id="view" class="org.javafree.mvp.gui.cadastro.CadastroClienteWindow" singleton="false">
        <property name="clientesTableModel">
            <ref local="clientesTableModel" />
        </property>
    </bean>

    <!-- Modelo que mostra dados de uma lista de clientes em um componente JTable -->
    <bean id="clientesTableModel" class="org.javafree.mvp.gui.cadastro.ClientesTableModel" singleton="false">
```

```
        <constructor-arg>
            <ref local="model" />
        </constructor-arg>
    </bean>
</beans>
```

Para finalizar a criação do nosso aplicativo, crie a classe Main, no pacote org.javafree.mvp.application.

Main.java

```
package org.javafree.mvp.application;

public class Main {

    public static void main(String[] args) {
        getPresenter().createView();
    }

    public static CadastroClientePresenter getPresenter() {
        BeanFactory factory = new ClassPathXmlApplicationContext("./applicationContext.xml");
        return (CadastroClientePresenter)factory.getBean("presenter");
    }
}
```

A aplicação está finalizada. Só não se esqueça de adicionar os arquivos spring.jar e commons-logging.jar nas Libraries do projeto.

Conclusão

Se formos comparar duas aplicações, uma delas seguindo o padrão MVP e outra não, é inevitável a comparação entre o código escrito: a aplicação que utiliza o MVP tem mais linhas de código e mais classes do que aplicação que não utiliza o MVP. Eu particularmente não considero isso uma desvantagem, pois justamente a quebra do código em várias classes torna a aplicação mais fácil de entender e de expandir, além de permitir maior re-utilização e melhorar a testabilidade do código.

Referências

Martin Fowler - http://www.martinfowler.com/eaaDev/ModelViewPresenter.html

IBM ? s Taligent - ftp://www6.software.ibm.com/software/developer/library/mvp.pdf

MVC - http://en.wikipedia.org/wiki/Model-view-controller

MVC Meets Swing - http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html

Links

NetBeans - http://www.netbeans.org

Spring Framework - http://www.springframework.org

O JavaFree.org é uma comunidade java formada pela coolaboração dos desenvolvedores da tecnologia java. A publicação de artigos além de ajudar a comunidade java, ajuda a dar maior visibilidade para o autor. Contribua conosco.