UFRPE UAST Unidade Acadêmica de Serra Talhada - PE Desde 2006

Bacharelado em Sistemas de Informação

MPOO

Site: https://sites.google.com/site/profricodemery/mpoo

http://ava.ufrpe.br/

https://sigs.ufrpe.br/sigaa/ava/index.jsf

Disciplina: Modelagem e Programação Orientada a Objetos (MPOO)

Profº: Richarlyson D'Emery

LISTA DE EXERCÍCIOS IX

Leia atentamente as instruções gerais:

- No Eclipse crie um novo **projeto chamado br.edu.mpoo.listaIX.SeuNomeSobrenome**, o qual deverá ter **pastas de pacotes** para cada questão: questao1, questao2, e assim sucessivamente, contendo todas as respostas da lista.
- A lista envolve questões práticas e conceituais. Quando a questão envolver uma discussão teórica utilize um arquivo .txt (Menu File -> Submenu New -> Opção File), por exemplo, questao5_1.txt

Fique atento!

Prezado aluno, esta é a lista de exercícios relativa à Semana 12 em que vimos o conceito de "Tratamento de Eventos" e iniciamos a discussão sobre a organização de projeto com o *architectural pattern* MVC (do inglês, Model-View-Controller).

Relembrando!

Na aula de MPOO introduzimos o assunto de componentes gráficos em que aprendemos a criar interfaces gráficas para os nossos sistemas. Também vimos diversas possibilidades de codificação para a interação com essas telas (Tratamento de Eventos!):

- Tratamento de evento **em classe realizando uma interface**:

```
//View.java
public class View extends JFrame implements ActionListener {

    JButton button;

    public View(){
           //construção da GUI
           button = new JButton();
           //registro de listener
           button.addActionListener(this); //this -> indicação de tratamento na própria classe
    }

    //Método manipulador pertencente a classe
    @Override
    public void actionPerformed(ActionEvent event) {
           // método manipulador para tratar uma ação a ser executada por um componente gráfico, como,
por exemplo, button
    }
}
```

- Tratamento de evento **por classe interna anônima**

```
//View.java
public class View extends JFrame {

    JButton button;

    public View(){
           //construção da GUI
           button = new JButton();
           //registro de listener
           button.addActionListener(
                        new ActionListener() { //tratamento por classe interna anônima
                               @Override
                               public void actionPerformed(ActionEvent e) {
                                     // método manipulador para tratar a ação de button
                               }
                        });
    }
}
```

- Tratamento de evento **por classe interna**

```
//View.java
public class View extends JFrame{

    JButton button;
    ButtonHandler buttonHandler;

    public View(){
           //construção da GUI
           button = new JButton();
           buttonHandler = new ButtonHandler();//instância para classe interna
           //registro de listener
           button.addActionListener(buttonHandler); //indicação de tratamento na própria classe
    }

    private class ButtonHandler implements ActionListener{
           //Método manipulador pertence a classe interna
           @Override
           public void actionPerformed(ActionEvent arg0) {
                  // método manipulador para tratar uma ação a ser executada por um componente gráfico,
como, por exemplo, button
           }
    }
}
```

- Tratamento de evento **por classe outra classe que não interna**

```
//View.java
public class View extends JFrame{

    JButton button;
    ButtonHandler buttonHandler;

    public View(){
           //construção da gui
           button = new JButton();
           buttonHandler = new ButtonHandler(this); //passa esta view para a classe ButtonHandler que
tratará do evento ("comunicação entre classes!")

           //registro de listener
           button.addActionListener(buttonHandler); //indicação de tratamento na classe ButtonHandler
    }
}
```

(continua na próxima página...)

```java
//ButtonHandler.java
public class ButtonHandler implements ActionListener{

    View view;

    public ButtonHandler(View view) {
        this.view = view;
    }

    //Método manipulador pertencente a classe ButtonHandler
    @Override
    public void actionPerformed(ActionEvent arg0) {
        // método manipulador para tratar uma ação a ser executada por um componente gráfico de View
    }
}
```

Fique atento!

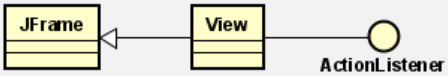
Observe que apenas a solução para o tratamento do evento por uma classe secundária não reconhece os componentes gráficos de uma *window* como variáveis globais, *claro*! Sendo necessária a passagem da view para a classe que tratará os eventos de seus componentes gráficos. Esta é uma problemática de comunicação entre classes.

Você Sabia?

Nas soluções apresentadas observamos o relacionamento entre classes e interfaces! Você saberia modelar cada uma delas? Vejamos as representações nos diagramas de classes abaixo:

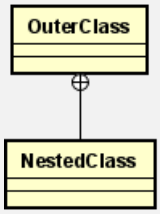
Classe com Herança e realização de interface:

```java
class View extends JFrame implements ActionListener{}
```



Nested Classe:

```java
class OuterClass {
    ...
    //além de default, o encapsulamento de NestedClass pode ser public, protected ou private
    class NestedClass {
        ...
    }
}
```

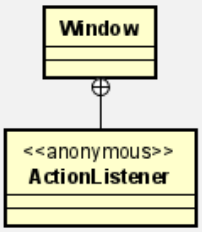


Atenção à terminologia: As classes aninhadas são divididas em duas categorias: não estáticas e estáticas. As classes aninhadas não estáticas são chamadas de classes internas (inner classes). As classes aninhadas que são declaradas estáticas são chamadas de classes aninhadas estáticas (static nested classes). Por exemplo:

```java
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

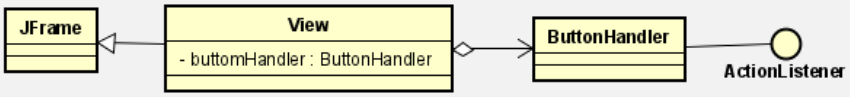
Anonymous inner classes:

```java
public class Window {
    public void tratamento(){
        ActionListener l = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // tratamento do evento
            }
        };
    }
}
```



Atenção à terminologia: *Não confundir "Anonymous Inner Class:" (classe interna anônima) com "Anonymous Bound Class" (classe vinculada anônima). Vejamos um exemplo de Anonymous Bound Class:*

Comunicação entre classes com realização de interfaces:



Mão na Massa!

1) Implemente em Java a GUI da calculadora ao lado (Fig. 1). Você deve utilizar componentes gráficos de javax.swing. Observe a disposição das opções da calculadora e escolha o devido layout. Realize os devidos tratamentos de eventos de maneira que tenha funcionalidade para pelo menos as quatro operações matemáticas. Lembre-se que as ações dos botões devem refletir a área de texto da calculadora, a qual não poderá ser editada pelo usuário. Adote a abordagem de "tratamento de evento por outra classe que não interna".

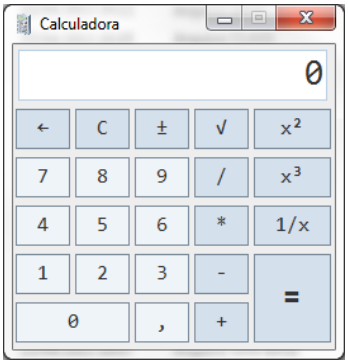
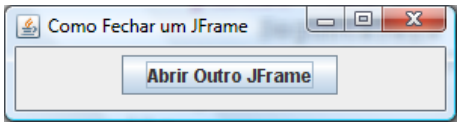
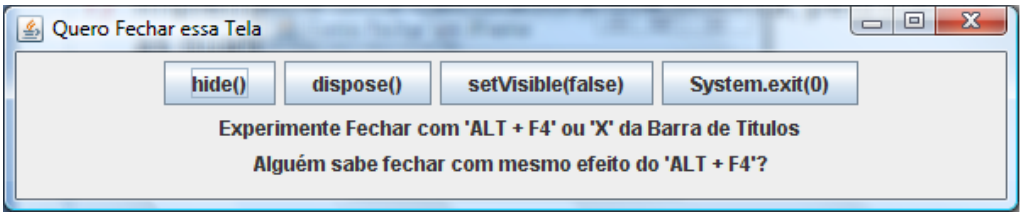


Fig. 1 *(260 x 255)*

2) A partir das quatro abordagens apresentadas na seção "Relembrando", implemente a seguinte aplicação Java:



Ao clicar no botão "Abrir Outro JFrame" (tratamento de evento por classe interna anônima) irá carregar outro JFrame:



Observe as funcionalidades para os botões:

- hide() – dispara o método hide() fechando o segundo JFrame. Utilize tratamento de evento em classe realizando uma interface.
- dispose () – dispara o método dispose() fechando o segundo JFrame. Utilize tratamento de evento por classe interna.
- setVisible(false) – dispara o método setVisible() fechando o segundo JFrame. Utilize tratamento de evento por outra classe que não interna.
- System.exit(0) – encerra a aplicação. Utilize tratamento de evento por classe interna anônima.

3) Uma Empresa solicitou a um de seus programadores (de codinome *mustela putórius furo* – "O Furão") que propusesse as GUI's para o sistema do "MPOO Market" em que são gerenciados clientes e produtos ***(Vide os problemas abordado na Lista de Exercícios VIII)***. Para o desenvolvimento, o Scrum Master definiu que a solução deve utilizar exclusivamente as bibliotecas disponíveis no JDK, ou seja, em `java` e `javax`, sem o auxílio de recursos de IDE's do tipo *drag-and-drop*, como, por exemplo, a palette de Design do Eclipse. Para ilustrar o sistema, foram definas duas telas: uma para abertura do sistema (Fig. 2A) e outra para gerenciar dados (Fig. 2B). Na barra de menu (JMenuBar), Fig. 2B, contém as opções para gerenciamento dos conceitos pretendidos. Mas, é possível observar que as opções estão incompletas, uma vez que se pode cadastrar, buscar, remover e atualizar os dados de um cliente ou um produto. **Realize o tratamento de evento em classe realizando uma interface para as opções do Menu, ou seja, atualiza as GUI's de acordo sua denominação.**

Fig. 2A

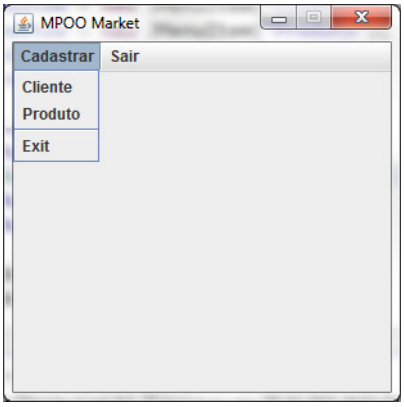
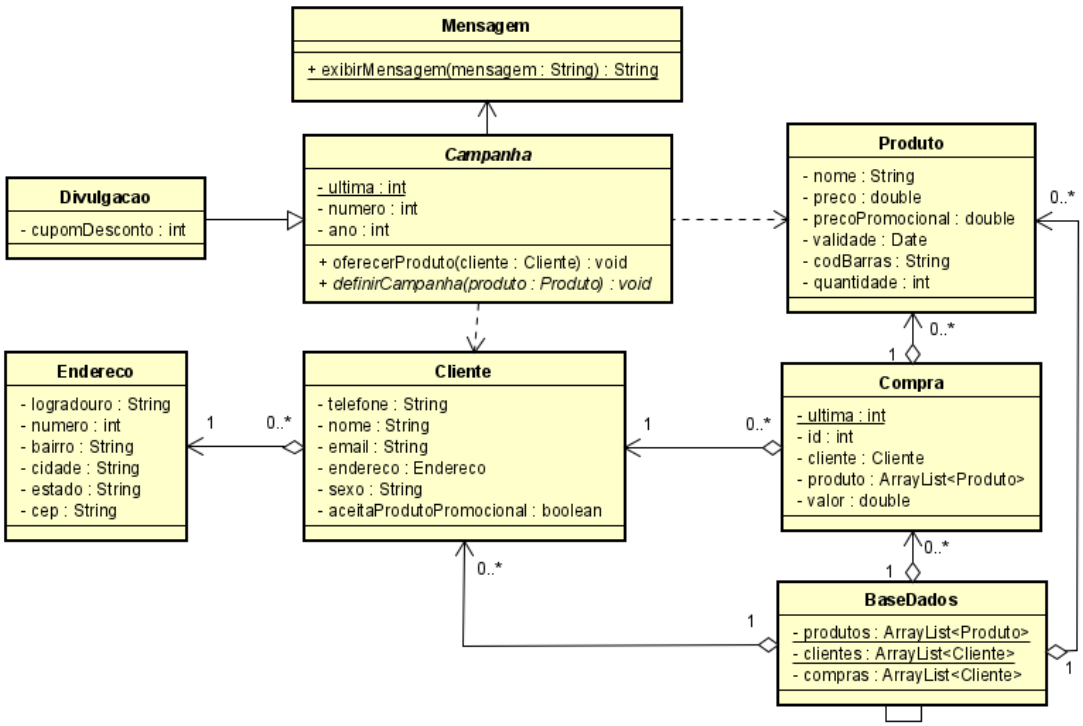


Fig. 2B

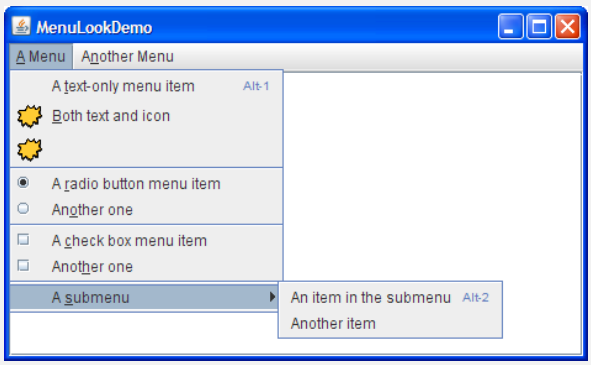
As opções do menu deverão atualizar o sistema com os campos para gerenciar a opção selecionada. Esses dados, pertencentes a Clientes e Produtos, devem estar de acordo o o diagrama de classe abaixo. Portanto, proponha GUI's para gerenciar esses dados.

Dica: Cada menu acionará JPanel com os campos desejados.



Saiba Mais!

Em um sistema é possível adicionar no menu opções que não são comumente usuais, mas que permitem a personalização amigável de uma interface, através da inclusão de: ImageIcon, JRadioButton, JCheckBox, JSeparator e indicação de tecla de atalho. Por exemplo:



Para saber mais como utilizar esses componentes e para seus respectivos tratamentos e eventos, acesse o tutorial de javase da Oracle disponível em:
https://docs.oracle.com/javase/tutorial/uiswing/components/menu.html

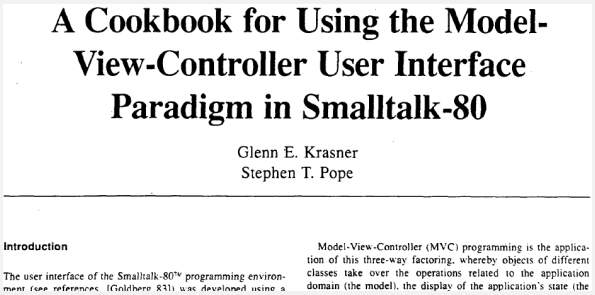
Desafio

(Opcional) Ainda no sistema do "MPOO Market" adicione ícones e teclas de atalho para algumas opções do menu.

Você Sabia?

Os padrões de design permitem que os desenvolvedores projetem partes específicas dos sistemas, como abstrair instanciações de objetos ou agregar classes a estruturas maiores. Os padrões de design também promovem o acoplamento fraco entre objetos. Padrões arquitetônicos promovem o acoplamento fraco entre subsistemas. Esses padrões especificam como os subsistemas interagem um com o outro. Um dos padrões arquitetônicos mais populares é o Model-View-Controller (MVC), o qual separa a codificação em camadas.

MVC foi desenvolvido para construir interfaces gráficas em Smalltalk por Trygve Reenskaug enquanto trabalhava na Xerox PARC e descrito inicialmente nos trabalhos de Krasner e Pope (1988) e Burberck (1992).



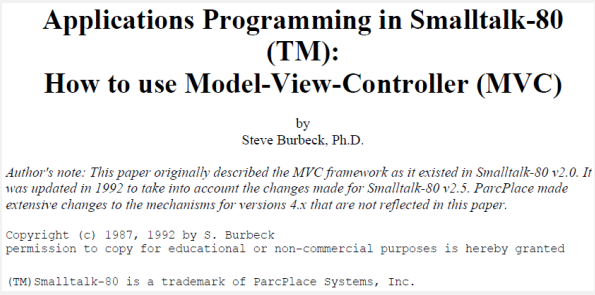
A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80

Glenn E. Krasner
Stephen T. Pope

Introduction

The user interface of the Smalltalk-80™ programming environment (see references, [Goldberg 83]) was developed using a

Model-View-Controller (MVC) programming is the application of this three-way factoring, whereby objects of different classes take over the operations related to the application domain (the model), the display of the application's state (the



Applications Programming in Smalltalk-80 (TM):
How to use Model-View-Controller (MVC)

by
Steve Burbeck, Ph.D.

Author's note: This paper originally described the MVC framework as it existed in Smalltalk-80 v2.0. It was updated in 1992 to take into account the changes made for Smalltalk-80 v2.5. ParcPlace made extensive changes to the mechanisms for versions 4.x that are not reflected in this paper.

Copyright (c) 1987, 1992 by S. Burbeck
permission to copy for educational or non-commercial purposes is hereby granted

(TM) Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

MVC é composto por três tipos de objetos: (i) o "*modelo*" é o objeto de aplicação, a "*vista*" é a apresentação na tela e o "*controlador*" define a maneira como a interface do usuário reage às entradas do mesmo. Antes do MVC, os projetos de interface para o usuário tendiam em agrupar esses objetos em um único módulo, consequentemente, havia muitas linhas de código dificultando o entendimento, a manutenção e a reutilização desse código. MVC para aumentar a flexibilidade e a reutilização (GAMMA et al. 2000, p. 20[1]). O MVC tem como principal objetivo: separar dados ou lógicos de negócios (Model) da interface do usuário (View) e o fluxo da aplicação (Controller), a ideia é permitir que uma mensagem da lógica de negócios pudesse ser acessada e visualizada através de várias interfaces. Na arquitetura MVC, á lógica de negócios, ou seja, nosso Model não sabe quantas nem quais as interfaces com o usuário esta exibindo seu estado, a view não se importa de onde esta recebendo os dados, mas ela tem que garantir que sua aparência reflita o estado do modelo, ou seja, sempre que os estados do modelo mudam, o modelo notifica as view para que as mesmas atualizem-se (*sendo esta última uma perspectiva atualmente adotada pelo padrão Observer – cenas dos próximos capítulos!*).

Não é recomendado quando o sistema não tenha uma interdependência de dados, ou seja, quando se trata de um sistema embarcado, único ou muito pequeno, devido ao seu custo de desenvolvimento (aumento da separabilidade de código em classes distintas).

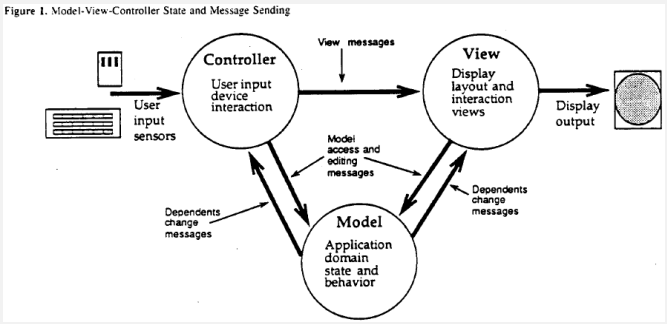
O MVC inicialmente foi desenvolvido no intuito de mapear o método tradicional de entrada, processamento e saída que os diversos programas baseados em GUI utilizavam. No padrão MVC, teríamos então o mapeamento de cada uma dessas três partes para o padrão MVC:

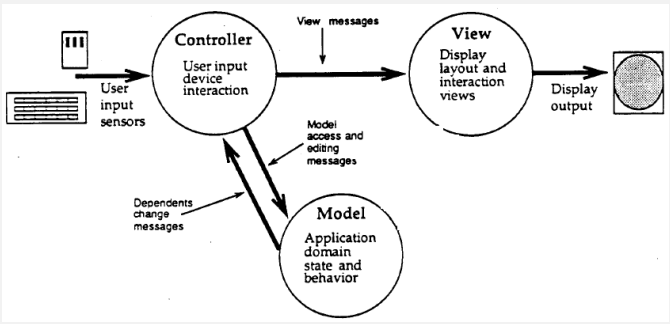
Input ➤ Processing ➤ Output
Controller ➤ Model ➤ View

Em suma: (i) model contém os dados da aplicação com a lógica da aplicação e as regras de negócios; (ii) view's são as interfaces gráficas em que o usuário interage; e (iii) controller intermedia o fluxo de informação entre view e model. Cada um com suas funcionalidades específicas.

Originalmente, Krasner e Pope (1988) descrevem o modelo (*esquerda*) em que todos os imputs são iniciados pelo Controller gerenciando a comunicação para as telas (views) e o domínio da aplicação (model). Entretanto, hoje em dia observa-se que na maioria das soluções que utilizam o MVC a remoção da comunicação entre View e Model, deixando essa comunicação por intermédio do Controlador (*direita*).

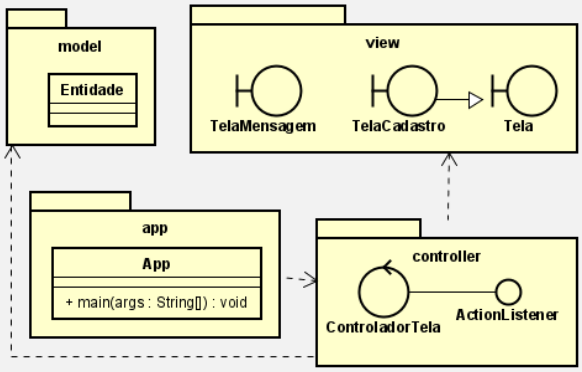
[1] GAMMA, E. et al. ***Padrões de Projeto:*** *Soluções reutilizáveis de software Orientado a Objetos.* Porto Alegre: Bookman, 2000.





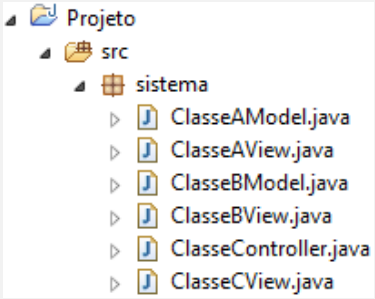
A representação MVC utiliza a mesma do padrão boundary-control-entity (BCE), em que: as entidades pertencem ao model, control ao controller e tela à boundary. Vale destacar que as diferenças entre os padrões arquiteturais se são principalmente quanto às regras de negócios do controle de BCE e controlador de MVC, uma vez que o primeiro encapsula também a lógica de negócios do caso de uso, enquanto o controlador MVC processa a entrada do usuário que seria de responsabilidade da fronteira (boundary) de BCE. O controle de BCE aumenta a separação de interesses na arquitetura ao encapsular a lógica de negócios que não está diretamente relacionada a uma entidade.

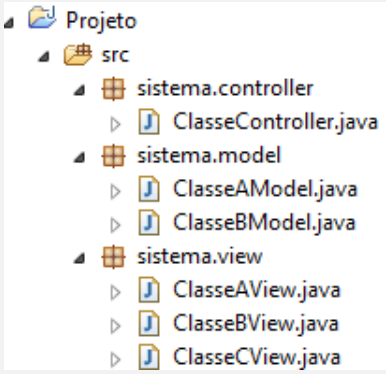
Vejamos uma exemplificação da modelagem MVC fazendo uso dos elementos BCE:



Fique atento!

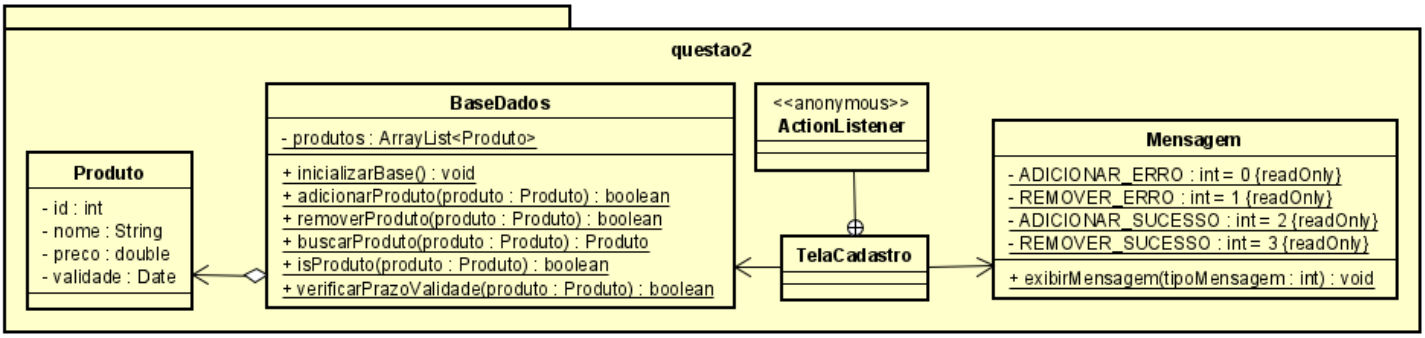
Não existe uma regra quanto à organização dos arquivos no projeto Java. Alguns desenvolvedores optam por separar as camadas apenas nomeando os arquivos ou os organizando em pacotes. Também se podem variar as nomenclaturas para as camadas por sinônimos que as representam, como, por exemplo, entidade para model, tela ou gui para view, e controle ou controlador(a) para controller.





Na disciplina MPOO, vamos adotar a padronização por separação de camadas em pacotes!

4) Uma Empresa solicitou a um de seus programadores (*de codinome mustela putórius furo – "O Furão"*) que resolvesse uma demanda de funcionalidades de seu sistema. Esta demanda envolve a manutenção de produtos. Entretanto, *O Furão* não havia estudado o padrão arquitetura *Model-View-Controller*, consequentemente, condensando diversas funcionalidades. A Tela Cadastro possui menus para cada comportamento disponível na base de dados, os quais permitem a atualização de telas conforme a funcionalidade pretendida. Projete as telas que *O Furão* possa ter pensado. Abaixo está o diagrama pensado por *O Furão* com as classes utilizadas e seus relacionamentos. Sendo assim, codifique em Java a solução pensada!



Desafio

5) Aproveite a oportunidade e mostre que você tem a capacidade de ocupar a vaga de "O Furão" na empresa. Observe que parte da codificação da questão 4 é condensada em um único arquivo (tela, tratamento de eventos e regras de negócios), isso porque *O Furão* não havia estudado o padrão arquitetural MVC. Após a leitura das seções "Você Sabia?" e "Fique Atento!" sobre MVC, responda as próximas perguntas para garantir sua vaga na empresa:

   5.1) Quais classes pensadas por *O Furão* condensam diversas funcionalidades que poderiam ser separadas? Explique os porquês?

   5.2) **(Opcional)** Apresente uma solução (diagrama de classes e codificação Java) adequada para o problema. *Dica: Lembre-se que você poderá aproveita a opção de copiar pacote e colar os códigos-fonte em um novo pacote (questao5_2)*