Aplicações de Linguagem de Programação Orientada a Objeto

Design Patterns
MVC

Professora Sheila Cáceres

Padrões de Projeto (Design Patterns)

- Design Pattern: é uma solução geral reutilizável para um problema que ocorre frequentemente dentro de um contexto de projeto de software.
- São um conjunto de modelos de como resolver problemas comuns.
- São boas práticas de programação para resolver determinadas situações que costumam acontecer quando se desenvolve um sistema.

Design Patterns, historia

- Em 1987 Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto em ciência da computação e foram apresentados na conferência OOPSLA. Eram padrões para a construção de janelas na liguagem Smalltalk.
- O Livro “Design Patterns: Elements of Reusable Object-Oriented Software”, publicado em 1995, deu popularidade aos design patterns. Os autores são: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides e ficaram conhecidos como a "Gangue dos Quatro" (Gang of Four - GoF).

Design Patterns

- Segundo o livro mencionado, os padrões são organizados em 3 famílias:
  - Padrões de criação
    - Abstract Factory
    - Builder
    - Factory Method
    - Prototype
    - Singleton
  - Padrões estruturais
    - Adapter
    - Bridge
    - Composite
    - Decorator
    - Facade
    - Flyweight
    - Proxy
  - Padrões comportamentais ...

Design Patterns

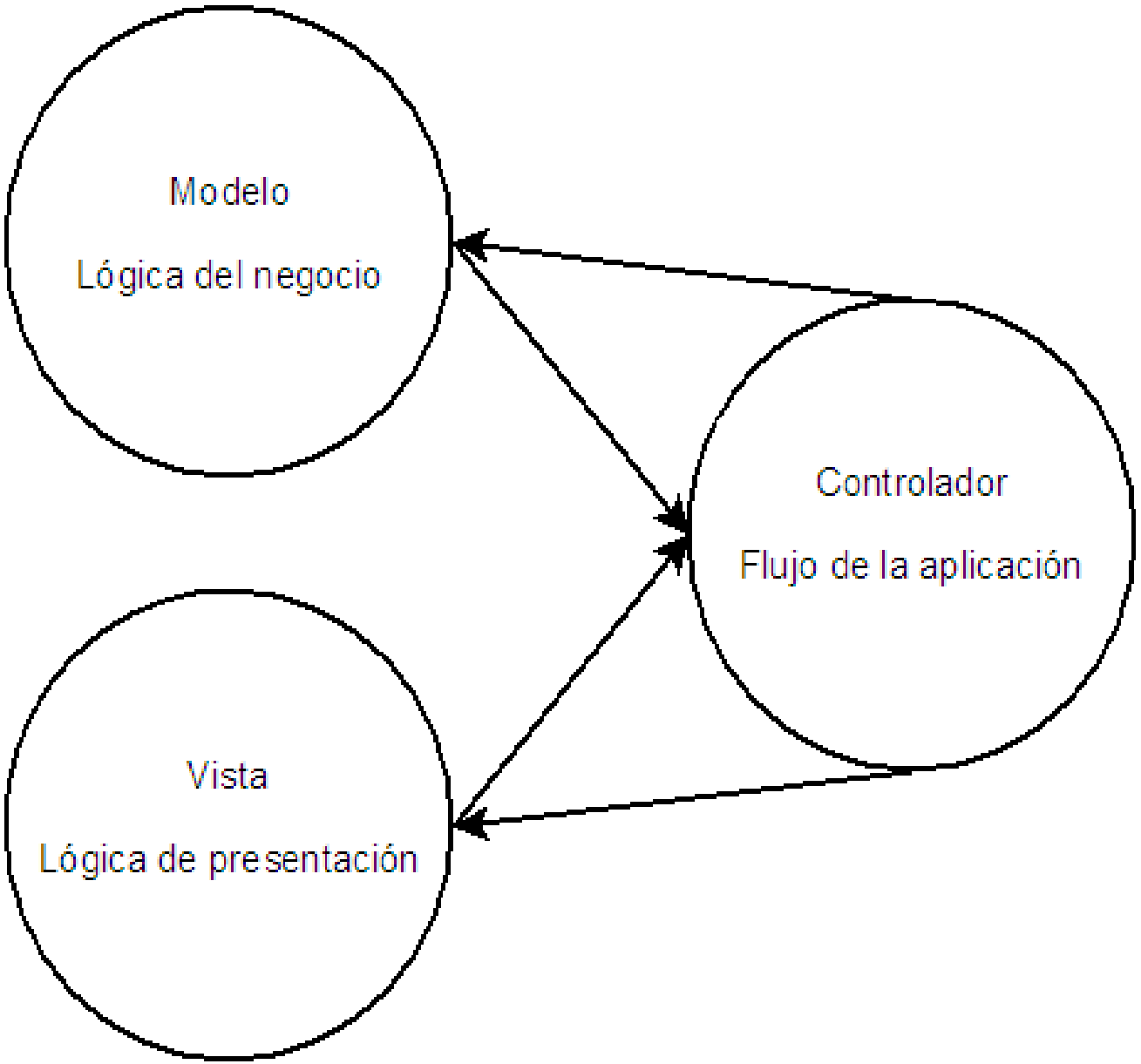
- Padrões comportamentais
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Modelo Vista Controlador
MVC

MVC

- MVC é um deign pattern orientado a objetos.
- Foi desenvolvido no Centro de Pesquisas Xerox Palo Alto a finais dos anos setenta.
- Originou-se na comunidade Smalltalk para implementar interfaces de usuario na qual as responsabilidades esivessem bem distribuidas entre distintas partes (componentes) do design.
- Assim, distinguiram-se tres responsabilidades distintas:
  - Lógica de negocio ➔ Modelo.
  - Gestão de eventos de usuario ➔ Controlador.
  - Presentação ➔ Vista.

Diagrama de MVC



O modelo

- Representa a lógica do negocio da aplicação.
- Chama-se modelo porque representa objetos e as suas iterações do mundo real.
- Muitas aplicações usam um mecanismo de armazenamento persistente como, por exemplo, um banco de dados para armazenar os dados fornecidos pelo usuário.
- MVC não cita especificamente a camada para acesso aos dados, porque subentende-se que estes métodos estariam encapsulados pelo Model.
- Exemplo:
  - aluno, professor e turma fazem parte do domínio de um sistema académico.

Vista

- Representa a lógica de apresentação.
- As vistas são as partes da aplicação MVC que presentam a saida ao usuário.
- "Renderiza" o model em uma forma especifica para a interação, geralmente uma interface de usuário.
- Separar o modelo e a vista permite a construção de interfaces con diferentes visuais.

Controlador

- Responsavel por receber eventos, determinar o processamento deles e finalmente provocar a geração de uma vista adequada.
- Generalmente cria instancias dos modelos e utiliza métodos deles para conseguir os dados que são apresentados aos usuários, enviándo-os a vista correspondente.
- Os controladores deven realizar:
  - Controle da segurança.
  - Identificação de eventos.
  - Preparar o modelo.
  - Procesar o evento.
  - Tratar os errores.
  - Provocar a generação de uma resposta.

Exemplo - Modelo

```
// O modelo faz o calculos necessarios.
// Não sabe nada em relação às vistas

public class CalculatorModel {

    private int calculationValue;

    // O modelo realiza as operações
    public void addTwoNumbers(int firstNumber, int secondNumber)
{
        calculationValue = firstNumber + secondNumber;
    }

    // O modelo mantem o resultado da soma dos números
ingressados na vista
    public int getCalculationValue(){
        return calculationValue;
    }
}
```

Exemplo Vista

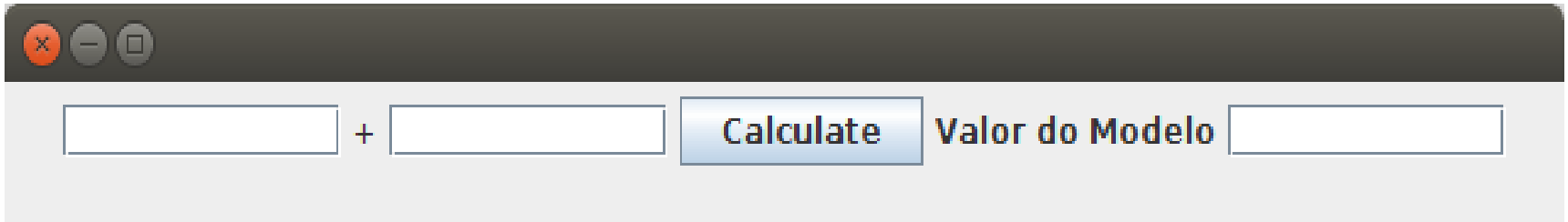
```
// A vista mostra a interface do usuário. Deve mostrar o estado do Modelo, Não realiza calculos
public class CalculatorView extends JFrame{
     private JTextField txtFirstNumber  = new JTextField(10);
     private JLabel lblAddition = new JLabel("+");
     private JTextField txtSecondNumber = new JTextField(10);
     private JButton calculateButton = new JButton("Calculate");
     private JLabel lblResult = new JLabel("Valor do Modelo");
     private JTextField txtCalcSolution = new JTextField(10);
     public CalculatorView(){
          JPanel calcPanel = new JPanel();
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          this.setSize(600, 200);
          calcPanel.add(txtFirstNumber);
          calcPanel.add(lblAddition);
          calcPanel.add(txtSecondNumber);
          calcPanel.add(calculateButton);
          calcPanel.add(lblResult);
          calcPanel.add(txtCalcSolution);
          this.add(calcPanel);
     }
     public int getFirstNumber(){     return Integer.parseInt(txtFirstNumber.getText());  }
     public int getSecondNumber(){  return Integer.parseInt(txtSecondNumber.getText()); }
     public int getCalcSolution(){ return Integer.parseInt(txtCalcSolution.getText());  }
     public void updateView(int solution){ txtCalcSolution.setText(Integer.toString(solution));}
     // Se o botão é clicado, o controlador trata esse evento
     void addCalculateListener(ActionListener listenForCalcButton){
          calculateButton.addActionListener(listenForCalcButton);
     }
     // Abre um popup que contem a mensagem de erro
     void displayErrorMessage(String errorMessage){
          JOptionPane.showMessageDialog(this, errorMessage);
     }
}
```

Controlador

```
// Coordena a interação entre a vista e o modelo
public class CalculatorController {
     private CalculatorView theView;
     private CalculatorModel theModel;

     public CalculatorController(CalculatorView theView, CalculatorModel theModel) {
          this.theView = theView;
          this.theModel = theModel;
     // Aloca à vista um listener que será tratado no metodo actionPerformed no Controlador
          this.theView.addCalculateListener(new CalculateListener());
     }

     class CalculateListener implements ActionListener{
          public void actionPerformed(ActionEvent e) {
               int firstNumber, secondNumber = 0;
               try{
                    firstNumber = theView.getFirstNumber();
                    secondNumber = theView.getSecondNumber();
                    theModel.addTwoNumbers(firstNumber, secondNumber);// modifica o modelo
                    theView.updateView(theModel.getCalculationValue());// atualiza as vistas
               }
               catch(NumberFormatException ex){
                    System.out.println(ex);
                    theView.displayErrorMessage("You Need to Enter 2 Integers");

               }
          }
     }
}
```

Rodando

```
public class MVCCalculator {
    public static void main(String[] args) {
        CalculatorView theView = new CalculatorView();
        CalculatorModel theModel = new CalculatorModel();
      CalculatorController theController=new
CalculatorController(theView,theModel);
      theView.setVisible(true);
      theView.setSize(650,80);
   }
}
```



Bibliografia

Alguns exemplos foram extraídos do material apresentado a seguir:

- Dessign Patterns, Eric Gamma et al.
- http://www.newthinktank.com/, Derek Banas, 2013.