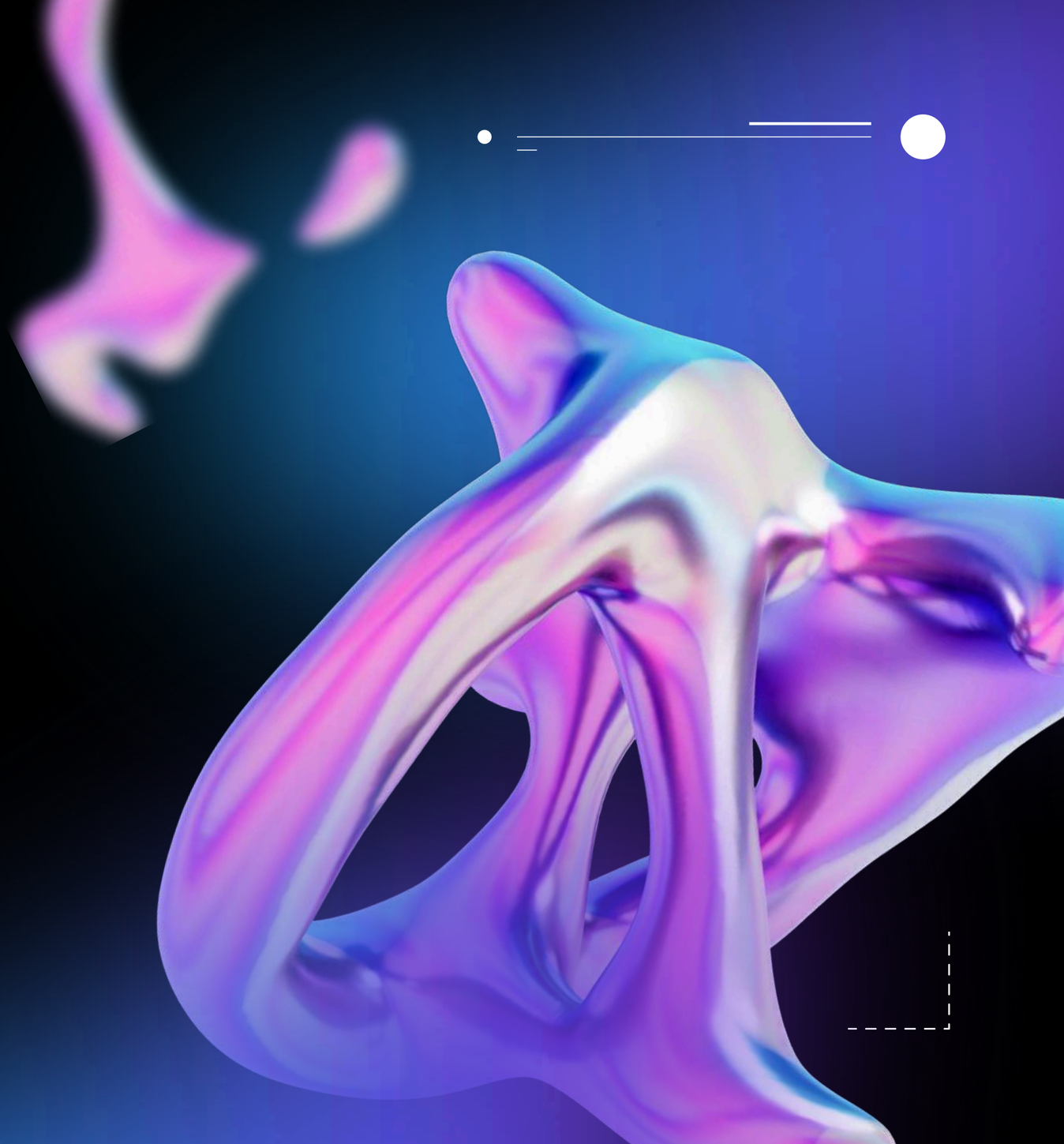
PADRÕES DE PROJETO GRASP

Variações Protegidas & Controlador.



IFRS - Campus Farroupilha
Alunos: Mariana Rossatto e Samuel Balbinot
Professor: Rogério Xavier de Azambuja

Controller.

- Qual é o primeiro objeto, além da camada de interface, que recebe e coordena (“controla”) as operações do sistema quando um evento ocorre?

- Esses eventos vêm da camada de interface do usuário, mas, para organizar melhor o software, queremos que essa responsabilidade de controle seja atribuída a outra camada – e não diretamente à interface.

- A solução é atribuir a responsabilidade de controlar o evento a uma classe **Controller**.

Características.

Centralização da lógica de controle:

Responsável por lidar com os eventos de entrada do sistema e direcioná-los para os objetos apropriados.

Desacoplamento:

Ao centralizar a lógica de controle, o Controlador promove um desacoplamento entre a interface do usuário e a lógica de negócio.

Delegação de responsabilidades:

Delega tarefas específicas a outras classes, mantendo o princípio da separação de responsabilidades.

Exemplo.


Biblioteca
Título do Livro:
Adicionar Livro
Excluir Livro
Lista de Livros:
- The Hobbit
- Harry Potter

Classe Livro.

```java
public class Livro {
    private String titulo;

    public Livro(String titulo) {
        this.titulo = titulo;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    @Override
    public String toString() {
        return titulo;
    }
}
```

Classe Service.

```java
import java.util.ArrayList;
import java.util.List;

public class BibliotecaService {
    private List<Livro> livros = new ArrayList<>();

    public void adicionarLivro(Livro livro) {
        livros.add(livro);
    }

    public void removerLivro(String titulo) {
        livros.removeIf(livro -> livro.getTitulo().equals(titulo));
    }

    public List<Livro> listarLivros() {
        return new ArrayList<>(livros);
    }
}
```

BibliotecaController.

```java
import java.util.List;

public class BibliotecaController {
    private BibliotecaService bibliotecaService;

    public BibliotecaController() {
        this.bibliotecaService = new BibliotecaService();
    }

    public void adicionarLivro(String titulo) {
        Livro livro = new Livro(titulo);
        bibliotecaService.adicionarLivro(livro);
    }

    public void removerLivro(String titulo) {
        bibliotecaService.removerLivro(titulo);
    }

    public List<Livro> listarLivros() {
        return bibliotecaService.listarLivros();
    }
}
```

BibliotecaUI.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BibliotecaUI {
    private BibliotecaController controller;
    private JFrame frame;
    private JTextField campoTitulo;
    private JTextArea areaDeTexto;

    public BibliotecaUI() {
        controller = new BibliotecaController();
        frame = new JFrame(title:"Biblioteca");
        campoTitulo = new JTextField(columns:20);
        areaDeTexto = new JTextArea(rows:10, columns:30);
        areaDeTexto.setEditable(b:false);
```

BibliotecaUI.

```
19
20      JButton botaoAdicionar = new JButton(text:"Adicionar Livro");
21      JButton botaoExcluir = new JButton(text:"Excluir Livro");
22
23      botaoAdicionar.addActionListener(new ActionListener() {
24          public void actionPerformed(ActionEvent e) {
25              String titulo = campoTitulo.getText();
26              controller.adicionarLivro(titulo);
27              atualizarListaDeLivros();
28              campoTitulo.setText(t:"");
29          }
30      });
31
32      botaoExcluir.addActionListener(new ActionListener() {
33          public void actionPerformed(ActionEvent e) {
34              String titulo = campoTitulo.getText();
35              controller.removerLivro(titulo);
36              atualizarListaDeLivros();
37              campoTitulo.setText(t:"");
38          }
39      });
```

BibliotecaUI.

```
40          JPanel panel = new JPanel();
41          panel.setLayout(new FlowLayout());
42          panel.add(new JLabel(text:"Título do Livro:"));
43          panel.add(campoTitulo);
44          panel.add(botaoAdicionar);
45          panel.add(botaoExcluir);
46
47          frame.add(panel, BorderLayout.NORTH);
48          frame.add(new JScrollPane(areaDeTexto), BorderLayout.CENTER);
49          frame.pack();
50          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51          frame.setVisible(b:true);
52
53          atualizarListaDeLivros();
54      }
```

```
55
56      private void atualizarListaDeLivros() {
57          areaDeTexto.setText(controller.listarLivros());
58      }
59
        Run | Debug
60      public static void main(String[] args) {
61          SwingUtilities.invokeLater(new Runnable() {
62              public void run() {
63                  new BibliotecaUI();
64              }
65          });
66      }
67  }
```

Variações Protegidas.

- Protected Variations é um princípio que visa proteger partes do sistema de mudanças nos elementos externos (como outros objetos, componentes ou sistemas).

- Esse princípio se concentra em definir interfaces e encapsular detalhes que podem variar, minimizando o impacto de mudanças no sistema.

Características.

- Encapsulamento de Pontos de Variação: Isola elementos do sistema que podem ser afetados por mudanças externas.
- Definição de Interfaces Estáveis: Cria interfaces que permanecem consistentes, mesmo que as implementações mudem.
- Isolamento de Dependências: Minimiza o impacto de mudanças externas, tornando o sistema mais robusto.
- Flexibilidade e Adaptabilidade: Facilita adaptações e extensões futuras no sistema.

Exemplo.

Plataforma de e-commerce que precisa integrar diferentes provedores de pagamento (como PayPal, Stripe e MercadoPago)

- Em vez de vincular diretamente o sistema a um provedor específico, é possível usar o princípio de Protected Variations para proteger o sistema de mudanças futuras, permitindo que você altere ou adicione provedores de pagamento sem afetar o sistema principal

Relação com Padrões de Projeto.

Erich Gamma e colaboradores destacam como alguns padrões de projeto aplicam PV.

Como por exemplo, o padrão Adapter, cria uma interface que adapta uma classe a outra interface, protegendo o sistema das mudanças na classe adaptada.

Vantagens.

- Reduz o impacto de mudanças.
- Facilita a testabilidade, uma vez que partes do sistema podem ser substituídas ou simuladas.
- Promove o reuso de código ao permitir a troca de implementações.

Desvantagens.

- Aumento de Complexidade: Mais interfaces e classes podem tornar o sistema mais complexo.
- Sobrecarga de Desenvolvimento: Requer mais tempo e estorço, especialmente em sistemas menores.
- Dificuldade na Identificação de Pontos de Variação: Pode ser difícil prever com precisão quais partes realmente precisarão de encapsulamento.

Referências

LARMAN, Craig. **Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e desenvolvimento iterativo**. Porto Alegre: Bookman, 2011.

SILVA, Ramon Ferreira. **Controller – Padrões GRASP.** 2019. Disponível em: https://www.ramonsilva.net/post/controller-padr%C3%B5es-grasp

Obrigado!