

# AULA 5 – PRINCÍPIOS E PADRÕES DE PROJETO

GS1020 – Programação Orientada a Objetos II

Prof. Dr. Murillo G. Carneiro  
*mgcarneiro@ufu.br*



*Material baseado nos slides cedidos pelo Prof. Rafael D. Araújo (FACOM/UFU)*

# Objetivo da aula

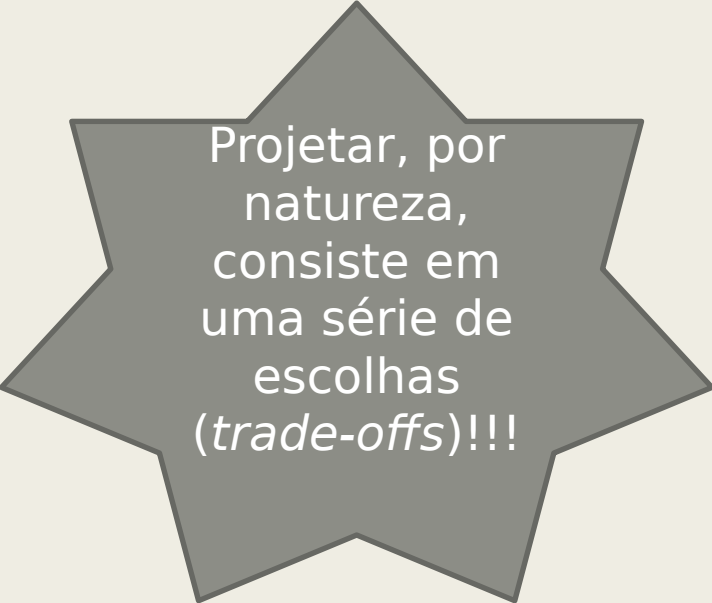
- Obter uma visão geral sobre os princípios de um bom projeto de software Orientado a Objetos e a importância da utilização de padrões de projeto.

# O que significa projetar um software?

- Do ponto de vista de processo: atividade do ciclo de vida na qual os requisitos de software são analisados para **produzir uma descrição da estrutura interna do software** que servirá de base para a sua construção.
- Do ponto de vista de resultado: descreve como um sistema é **decomposto e organizado em componentes e descreve as interfaces** entre esses componentes.

# Importância do projeto de software

- Detecção de problemas
- Tornar o software mais robusto
- Reduzir retrabalho
- Facilitar manutenção



Projetar, por natureza, consiste em uma série de escolhas (*trade-offs*)!!!

# Características de projetos ruins

*Design smells  
ou bad smells*

- **Rigidez** – difícil de mudar porque cada mudança altera muitas outras partes.
- **Fragilidade** – quando você faz uma mudança, partes inesperadas deixam de funcionar.
- **Imobilidade** – é difícil de reusar em outras aplicações porque não pode ser desacoplada da aplicação atual.
- **Viscosidade** – mais fácil fazer a coisa errada do que a certa.
- **Complexidade desnecessária** – elementos excessivos (e nunca utilizados) que não adicionam benefícios diretos.
- **Repetição desnecessária** – estruturas repetidas (copiar-e-colar) que dificultam correções.
- **Opacidade** – dificuldade de compreensão de trechos de código.

# Princípios SOLID

Cinco princípios para  
o bom projeto de  
software Orientado a  
Objetos

- ( S ) SRP: Single Responsibility Principle
- ( O ) OCP: Open/Close Principle
- ( L ) LSP: Liskov Substitution Principle
- ( I ) ISP: Interface Segregation Principle
- ( D ) DIP: Dependency Inversion Principle

# SRP: Princípio da Responsabilidade Única

coesão

- Uma classe deve possuir **apenas uma responsabilidade** totalmente **encapsulada** na respectiva classe
- “Uma classe deve ter somente uma razão para mudar”

Quando houver alteração relacionada à estrutura ou comportamento do conceito que a classe representa

# SRP: Exemplo de violação

```
public class Pedido {  
    public void AdicionarProduto(Produto produto, int quantidade) { }  
    public Float CalcularTotal() { }  
    public void GerarPlanilhaExcel() { }  
}
```

Relacionado à exibição de dados em um formato específico. Faz mais sentido estar em uma classe relacionada com UI.



# SRP: Exemplo de violação

```
public class Cliente {  
    public Float CalcularDescontoPara(Venda venda) {  
        if (venda.FormaDePagamento == FormaDePagamento.AVista) {  
            if (venda.Total > 2000)  
                return venda.Total * 0.2;  
            return venda.Total * 0.1;  
        }  
        return 0;  
    }  
}
```

Esse método não manipula nenhum dado do Cliente. Duas razões para essa classe ser alterada: uma quando houver alteração de Cliente e outra quando houver alguma alteração na lógica de uma Venda. Ou seja, essa classe “está fazendo coisas demais”.

# OCP: Princípio da Abertura-Fechamento



abstração

- O código deve ser aberto para extensão, mas fechado para alteração
- Quando for preciso estender o comportamento de um código, é melhor criar um novo código novo ao invés de alterar o código existente

# OCP: Exemplo de violação

Necessidade de um novo requisito: suporte para arquivos TXT. O que fazer?

```
public class Arquivo { }

public class ArquivoWord extends Arquivo {
    public void GerarDocX() {
        // codigo para geracao do arquivo DOCX
    }
}

public class ArquivoPdf extends Arquivo {
    public void GerarPdf() {
        // codigo para geracao do arquivo PDF
    }
}
```

```
public class GeradorDeArquivos {
    public void GerarArquivos(ICollection<Arquivo> arquivos)
    {
        for (Arquivo arquivo : arquivos) {
            if (arquivo instanceof ArquivoWord)
                ((ArquivoWord)arquivo).GerarDocX();
            else if (arquivo instanceof ArquivoPdf)
                ((ArquivoPdf)arquivo).GerarPdf();
        }
    }
}
```

**Esse método não está  
fechado para mudanças!**

# OCP: Exemplo de violação

Possível solução

```
public interface Arquivo {  
    public abstract void Gerar();  
}  
  
public class ArquivoWord implements Arquivo  
{  
    @Override  
    public void Gerar() { ... }  
}  
  
public class ArquivoPdf implements Arquivo {  
    @Override  
    public void Gerar() { ... }  
}
```

```
public class GeradorDeArquivos {  
    public void GerarArquivos(ICollection<Arquivo> arquivos)  
    {  
        for (Arquivo arquivo : arquivos) {  
            arquivo.Gerar();  
        }  
    }  
}
```

Agora esse método  
está fechado para  
mudanças!

# LSP: Princípio da Substituição de Liskov

Barbara  
Liskov

- Primeira mulher dos EUA a obter o grau de doutorado (PhD) em Ciência da Computação.
- Uma das inventoras do Tipo Abstrato de Dados (TAD).
- Atualmente é professora do MIT.
- Recebeu o prêmio Turing da ACM em 2008 por seus trabalhos na concepção da POO.



Fonte: [https://pt.wikipedia.org/wiki/Barbara\\_Liskov](https://pt.wikipedia.org/wiki/Barbara_Liskov)

# LSP: Princípio da Substituição de Liskov

- Objetos devem ser substituíveis com instâncias de seus tipos base sem alterar o bom funcionamento do software
- Exemplo de violação
  - *Considere as mesmas classes do exemplo anterior (Arquivo, ArquivoWord e ArquivoPdf)*
  - *Um objeto ArquivoWord e um objeto ArquivoPdf podem ser substituídos por um objeto Arquivo?*

# LSP: Exemplo de violação

```
public class Retangulo {  
    private float altura;  
    private float comprimento;  
}  
  
public class Quadrado extends Retangulo {  
    @Override  
    public void setAltura(float alt) {  
        this.altura = alt;  
        this.comprimento = alt;  
    }  
    @Override  
    public void setCompr(float compr) {  
        this.altura = compr;  
        this.comprimento = compr;  
    }  
}
```

Com  
getter e  
setter

```
public void redimensionar(Retangulo ret) {  
    ret.setAltura(ret.getAltura() * 2);  
    ret.setCompr(ret.getCompr() * 4);  
    //imprime área = altura * comprimento  
}
```

...

```
Retangulo ret = new Retangulo();  
ret.setAltura(3);  
ret.setComprimento(5);  
redimensionar(ret);
```

Qual seria o resultado  
da área após o  
redimensionamento?

```
Quadrado qua = new Quadrado();  
qua.setAltura(3);  
redimensionar(qua);
```

Qual seria o resultado  
da área após o  
redimensionamento?

# ISP: Princípio da Segregação de Interface



Coesão de interfaces

- “Clientes não devem ser forçados a depender de interfaces que eles não usam”.
- Várias interfaces específicas são melhores que uma única interface genérica.
- Diminui o acoplamento.



# ISP: Exemplo de violação

```
public abstract class Funcionario {  
    public float getSalarioBase() {  
        return this.salarioBase;  
    }  
    public float getSalario();  
    public float getComissao();  
}
```

Valor fixo

Salário base + comissão

Valor da comissão

Imagine as classes que herdam Funcionario:

Vendedor => ganha salário + comissão de 1%  
Representante => ganha apenas comissão de 10%  
AtendenteCaixa => ganha apenas salário

O que acontece com os métodos em cada classe?

# ISP: Exemplo de violação

A gray speech bubble with a tail pointing towards the code block, containing the text "Possível solução".

Possível solução

```
public interface Assalariado {  
    float getSalario();  
}  
  
public interface Comissionavel() {  
    float getComissao();  
}
```

- Funcionário implementa Assalariado
- Vendedor herda Funcionário e implementa Comissionavel
- Representante implementa Comissionavel

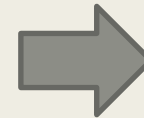
# DIP: Princípio da Inversão de Dependência

- Uma implementação deve depender de abstrações (interfaces), não de classes concretas.
- Componentes de alto nível não devem depender de componentes de níveis mais baixos. Ambos devem depender de abstrações.
  - *Ou seja, interfaces não devem depender de detalhes*

# DIP: Exemplo de violação

- Imagine um cenário onde toda chamada de método gere um log.

```
public class MinhaClasse {  
    public void MeuMetodo() {  
        //lógica para escrever log  
        //lógica do método em si  
    }  
}
```



```
Log l = new Log();  
l.EscreveLog();
```

MeuMetodo depende da implementação de Log

E se cada classe precisar log em um formato diferente?

# DIP: Exemplo de violação

Possível solução

- O correto seria o Log ser instanciado em primeiro lugar e esta instância ser passada para a classe (no construtor).
- Além disso, a classe deve depender de uma interface de Log e não da implementação de Log.

```
interface ILog {  
    void EscreveLog();  
}  
  
class LogTxt implements ILog {  
    public void EscreveLog() {  
        //método para escrever o log em TXT  
    }  
}
```

```
public class MinhaClasse {  
    private ILog log;  
    public MinhaClasse (ILog lg) {  
        this.log = lg;  
    }  
    public void MeuMetodo() {  
        log.EscreveLog();  
        //lógica do método em si  
    }  
}
```

# Vantagens de aplicar os princípios SOLID

## ■ Software:

- *mais extensível*
- *de fácil manutenção*
- *mais coeso*
- *com baixo acoplamento*

# O que é um padrão (no geral)?

- É uma forma de fazer alguma coisa (cozinhar, construir uma casa, montar um carro, fazer cerveja, desenvolver um software, etc.)
- Um método habitual e efetivo para se atingir um objetivo

# O que é um padrão de projeto de software?

- Um padrão de projeto (*design pattern*) é uma solução geral reutilizável para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software.
- Um padrão de projeto não é um projeto finalizado que pode ser diretamente transformado em código fonte ou de máquina, ele é uma descrição ou modelo (*template*) de como resolver um problema que pode ser usado em muitas situações diferentes.



# O que é um padrão de projeto?

- Padrões são boas práticas formalizadas que o programador pode usar para resolver problemas comuns quando projetar uma aplicação ou sistema.
- Padrões de projeto podem ser vistos como uma solução que já foi testada para um problema específico.
  - ***Alto nível de reuso***
  - ***Maior qualidade de software***

# Padrões de projeto

- Facilitam a **reutilização** de soluções de projeto do software.
- Estabelecem um vocabulário comum de projeto, facilitando **comunicação, documentação e aprendizado** dos sistemas de software.
- Consequentemente, facilitam a manutenção do software.
  - *Corretiva*
  - *Evolutiva*

# Importância do reuso

- Produtividade
- Confiabilidade
- Qualidade
- Redução de custo, esforço e risco

**“Quanto mais  
padrões eu utilizar,  
melhor vai ficar o  
meu código?”**

  
“Quanto mais  
padrões eu utilizar,  
melhor vai ficar o  
meu código?”

# Padrões de projeto

## ■ GRASP: General Responsibility and Assignment Software Patterns

– *Craig Larman, “Applying UML and Patterns”*

## ■ GoF: Gang of Four

Existem outros catálogos (famílias) de padrões, mas durante o curso vamos concentrar no GoF.

– *Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm, “Design Patterns”*

# Gang of Four (GoF)

- Descreve 23 padrões de projeto
- Soluções genéricas para os problemas mais comuns
- São documentação de soluções obtidas através da experiência
- Vocabulário comum para conversar sobre projetos de software

# Formato geral de um padrão de projeto

- Todo padrão inclui:
  - *seu **nome**,*
  - *o **problema**,*
  - *quando aplicar esta **solução** e*
  - *suas **consequências/forças**.*



# Características

■ Um padrão de projeto OO deve ter as seguintes características:

- *Encapsulamento: um padrão encapsula um problema e uma solução bem definida.*
- *Generalidade: todo padrão deve permitir a construção de outras realizações a partir deste padrão (reutilizável).*
- *Abstração: os padrões representam abstrações da experiência ou do conhecimento cotidiano.*

# Classificação GoF

- Os padrões são organizados em 3 propósitos:
  - **Padrões de criação:** *relacionados à criação de objetos*
  - **Padrões estruturais:** *tratam da composição de classes ou de objetos*
  - **Padrões comportamentais:** *tratam das interações e divisões de responsabilidades entre as classes ou objetos.*

# Classificação GoF

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	<ul style="list-style-type: none"> <li>• Factory Method</li> </ul>	<ul style="list-style-type: none"> <li>• (Class) Adapter</li> </ul>	<ul style="list-style-type: none"> <li>• Interpreter</li> <li>• Template Method</li> </ul>
	Objeto	<ul style="list-style-type: none"> <li>• Abstract Factory</li> <li>• Builder</li> <li>• Prototype</li> <li>• Singleton</li> </ul>	<ul style="list-style-type: none"> <li>• (Object) Adapter</li> <li>• Bridge</li> <li>• Composite</li> <li>• Decorator</li> <li>• Facade</li> <li>• Flyweight</li> <li>• Proxy</li> </ul>	<ul style="list-style-type: none"> <li>• Chain of Responsibility</li> <li>• Command</li> <li>• Iterator</li> <li>• Mediator</li> <li>• Memento</li> <li>• Observer</li> <li>• State</li> <li>• Strategy</li> <li>• Visitor</li> </ul>

# Referências

- GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995. Capítulo 1.
- MARTIN, R. C. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002. Capítulos 7 a 12.