



Universidade Federal do Ceará
Centro de Ciências/Departamento de Computação
Código da Disciplina: CK0236
Professor: Ismayle de Sousa Santos

Aula 02

Técnica de Programação II

POO e Princípios S.O.L.I.D.



qpg4p5x



ismaylesantos@great.ufc.br



@IsmayleSantos

Agenda

- **POO**
 - classe, objeto, atributo e método
 - Encapsulamento
 - Herança,
 - Polimorfismo
 - **S.O.L.I.D**
 - Single Responsibility Principle
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle
-

Antes do tópico de hoje ...



Exemplo Real de
Boas Práticas -
Loggi

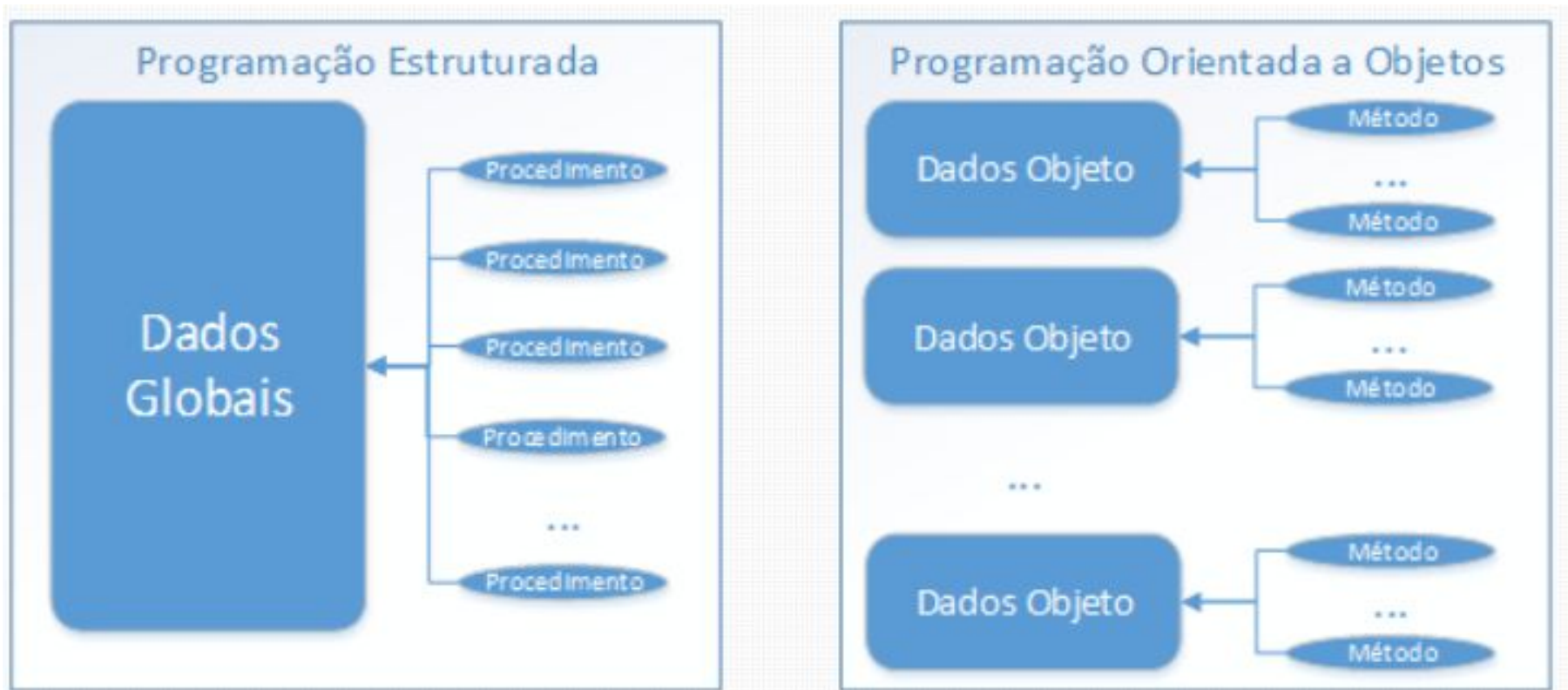
Techniques at a glance

- Fix real world first
- Code reviews
- Design docs
- Monorepo
- Continuous integration
- Feature switches
- Observability
- Dark launch & canary
- Iterate

POO

- **Programação Orientada a Objetos**
 - **Conjunto de objetos dizendo uns para os outros o que fazer através do envio de mensagens**
 - **Mensagens = chamadas a métodos de outro objetos**
 - **Modelagem no conceito de classe e nos seus relacionamentos**
-

Programação Estruturada vs POO



POO

- **Vantagens**
 - **Fácil de manter**
 - **Mais coeso**
 - **Código enxuto**
 - **Dados e funcionalidades encapsuladas**
 - **Independência (dos objetos)**
 - **Reuso**
-

P00

- **Suporte à P00**
 - **Java**
 - **C#**
 - **Python**



POO

- **Classe e Objeto**
 - **Classe**
 - Possui estrutura de dados e um conjunto de operações que atuam sobre estes dados
 - E.g.: Carro
 - **Objeto**
 - Instanciação de uma classe
 - E.g.: Fiat Uno, Pálio
-

POO

- Classe e Objeto - Exemplo de código

```
public class Aluno{  
  
    private int matricula;  
  
    private String endereço;  
  
    public int getMatricula() {  
        return matricula;  
    }  
  
    public void setMatricula(int matricula) {  
        this.matricula = matricula;  
    }  
  
    public String getEndereço() {  
        return endereço;  
    }  
  
    public void setEndereço(String endereço) {  
        this.endereço = endereço;  
    }  
}
```

POO

- **Herança**
 - **Evita duplicidade de código**
 - **Permite uma classe utilizar métodos e atributos de outras classes**
 - **Super-classe (classe base)**
 - **classe que é herdada**
 - **Sub-classe (classe derivada)**
 - **classe que herda**
 - **Sub-classe herda todas características da super-classe (comportamento e atributos)**
-

POO

- Herança - exemplo

```
public class Pessoa {  
    private String nome;  
    private String cpf;  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

```
public class Aluno extends Pessoa {  
    private int matricula;  
    private String endereço;
```

```
public class Main {  
    public static void Main() {  
        Aluno ana = new Aluno();  
        //função da super classe - Pessoa  
        ana.setNome("Ana Maria");  
        //função da classe Aluno  
        ana.setMatricula(234);  
    }  
}
```

POO

- **Classe Abstrata**
 - Classe que **não** pode ser instanciada
 - Ser para se utilizar herança
 - Pode conter um método abstrato
 - Método que deve ser sobrescrito
-

P OO

- Classe Abstrata - Exemplo

```
//Classe abstrata
public abstract class Funcionario {

    private String nome;

    private String identificar;

    //Método abstrato para ser implementado na classe base
    public abstract void imprimirSetor();

    public String getNome() {
        return nome;
    }

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

    public String getIdentificar() {
        return identificar;
    }

    public void setIdentificar(String identificar) {
        this.identificar = identificar;
    }
}

public class Analista extends Funcionario {

    @Override
    public void imprimirSetor() {
        System.out.println("Trabalha na TI ... ");
    }

}
```

POO

- Polimorfismo
 - Permite que o objetivo de uma subclasse seja usado instanciado como um objeto da superclasse
 - Um mesmo comando enviado para objetos diferentes pode gerar ações diferentes
-

POO

- Polimorfismo - Exemplo

```
Funcionario analista = new Analista();  
analista.setNome("Rafael");  
analista.imprimirSetor();
```

```
Funcionario professor = new Professor();  
professor.setNome("Lara");  
professor.imprimirSetor();
```

P OO

- **Interface**
 - Possui métodos que devem ser implementados pelas classes concretas que implementam essa interface
 - Uma classe pode implementar várias interfaces
 - Favorece o baixo acoplamento
-

POO

- Interface - Exemplo

```
public interface Beneficios {  
    boolean temValeAlimentação();  
    boolean temPlanoSaude();  
}
```

```
public class Professor extends Funcionario implements Beneficios{  
    @Override  
    public void imprimirSetor() {  
        System.out.println("Trabalha na docência ... ");  
    }  
    @Override  
    public boolean temValeAlimentação() {  
        return true;  
    }  
    @Override  
    public boolean temPlanoSaude() {  
        return true;  
    }  
}
```

POO

- **Encapsulamento**
 - É a capacidade de “esconder” parte do código e dos dados do restante do programa
 - Pode-se definir um grau de visibilidade aos métodos e atributos de cada Classe
-

P00

- Encapsulamento

	public	private
Atributos	Viola encapsulamento	Reforça encapsulamento
Métodos	Proporciona Serviços aos clientes	Suporta outros metodos na classe

P OO

- Boas Práticas
 - Código referente exclusivamente a uma classe deve ficar dentro desta classe
 - Uma classe deve ter o menor número possível de métodos públicos, mas deve ter pelo menos um método público
 - É uma boa prática evitar variáveis públicas. Prefira utilizar métodos (get, set) para acessar as variáveis
 - Use métodos construtores para inicializar objetos

POO

- **Boas Práticas**
 - **Classes** devem iniciar com letras maiúsculas, métodos, atributos e variáveis com letras minúsculas
 - **Métodos** devem ter um tipo de retorno ou “void”
 - **Métodos estáticos** não precisam de instâncias da classe
 - Construtores** são métodos especiais sem tipo de retorno

Princípios de Projeto

- Recomendações que desenvolvedores devem seguir para se garantir que um projeto atende a determinadas propriedades
- Princípios e Propriedades

Princípio de Projeto	Propriedade de Projeto
Responsabilidade Única	Coesão
Segregação de Interfaces	Coesão
Princípio de Inversão de dependência	Acoplamento
Prefira Composição a Herança	Acoplamento
Demeter	Ocultamento de Informação
Aberto/Fechado	Extensibilidade
Substituição de Liskov	Extensibilidade

Princípios SOLID

- Sigla cunhada por Robert Martin e Michael Feathers
 - **Single Responsibility Principle**
 - **Open Closed/Principle**
 - **Liskov Substitution Principle**
 - **Interface Segregation Principle**
 - **Dependency Inversion Principle**
-

Coesão

- **Toda classe deve implementar uma única funcionalidade ou serviço**
 - todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço
 - **Vantagens**
 - **Facilita implementação**
 - **Facilita reuso**
-

Acoplamento

- É a força da conexão entre duas classes
 - Quando mudanças em um elemento implicam em mudanças em outro elemento
 - Acoplamento pode ser indireto
 - Mudanças em A podem ser propagar para B, e então alcançar C
 - C está acoplado a A de forma indireta
 - O objetivo não é eliminar o acoplamento entre classes
-

Acoplamento

- Acoplamento **aceitável** de uma classe A para uma classe B
 - Classe A usa apenas métodos públicos da classe B
 - A interface provida por B é estável do ponto de vista sintático e semântico
 - são raras as mudanças em B que terão impacto em A

```
import java.util.Hashtable;

public class Estacionamento {

    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }
}
```

Ocultamento de Informação

- Classes devem esconder detalhes de implementação que estão sujeitas a mudanças

```
public class Estacionamento {  
  
    public Hashtable<String, String> veiculos;  
  
    public Estacionamento() {  
        veiculos = new Hashtable<String, String>();  
    }  
  
    public static void main(String[] args) {  
        Estacionamento e = new Estacionamento();  
        e.veiculos.put("TCP-7030", "Uno");  
        e.veiculos.put("BNF-4501", "Gol");  
        e.veiculos.put("JKL-3481", "Corsa");  
    }  
}
```

Ocultamento de Informação

- Exemplo

```
public class Estacionamento {  
  
    private Hashtable<String,String> veiculos;  
  
    public Estacionamento() {  
        veiculos = new Hashtable<String, String>();  
    }  
  
    public void estaciona(String placa, String veiculo) {  
        veiculos.put(placa, veiculo);  
    }  
  
    public static void main(String[] args) {  
        Estacionamento e = new Estacionamento();  
        e.estaciona("TCP-7030", "Uno");  
        e.estaciona("BNF-4501", "Gol");  
        e.estaciona("JKL-3481", "Corsa");  
    }  
}
```

Princípio da Responsabilidade Única

- Toda classe deve ter uma única responsabilidade
 - “Só deve haver um único motivo para modificar uma classe”
 - Uma implicação prática
 - Separar **apresentação** de **regras de negócio**
-

Princípio da Responsabilidade Única

Viola o SRP? Satisfaz o SRP?

```
class Disciplina {  
  
    void calculaIndiceDesistencia() {  
        indice = "calcula índice de desistência"  
        System.out.println(indice);  
    }  
  
}
```

Princípio da Responsabilidade Única

```
class Console {  
  
    void imprimeIndiceDesistencia(Disciplina disciplina) {  
        double indice = disciplina.calculaIndiceDesistencia();  
        System.out.println(indice);  
    }  
  
}  
  
class Disciplina {  
  
    double calculaIndiceDesistencia() {  
        double indice = "calcula índice de desistência"  
        return indice;  
    }  
  
}
```

Princípio da Segregação de Interfaces

- Interfaces tem que ser pequenas, coesas e específicas para cada tipo de cliente
 - É um caso particular de Responsabilidade Única com foco em interfaces
 - É melhor ter muitas interfaces específicas do que uma interface única
-

Princípio da Segregação de Interfaces

```
interface Funcionario {  
  
    double getSalario();  
  
    double getFGTS();// apenas funcionários CLT  
  
    int getSIAPE();// apenas funcionários públicos  
  
    ...  
}
```

Princípio da Segregação de Interfaces

```
interface Funcionario {  
    double getSalario();  
    ...  
}  
  
interface FuncionarioCLT extends Funcionario {  
    double getFGTS();  
    ...  
}  
  
interface FuncionarioPublico extends Funcionario {  
    int getSIAPE();  
    ...  
}
```

Princípio de Inversão de Dependências

- Recomenda que uma classe cliente deve estabelecer dependências prioritariamente com abstrações e não com implementações concretas
 - Abstrações (isto é, interfaces) são mais estáveis do que implementações concretas (isto é, classes).

Prefira Interfaces a Classes

Princípio de Inversão de Dependências

- Exemplo

```
void f() {  
    ...  
    ProjetorLG projetor = new ProjetorLG();  
    ...  
    g(projetor);  
}
```

```
void g(Projetor projetor) {  
    ...  
}
```

Prefira Composição a Herança

- Tipos de Herança
 - Herança de classe
 - classe A extends B
 - envolve reuso de código
 - Herança de interface
 - interface I implements J
 - não envolve reuso de código
 - Herança é um mecanismo de reuso caixa-branca
 - Subclasses têm acesso a implementação da classe base
-

Prefira Composição a Herança

- Composição entre classes A e B
 - Quando a classe A possui um atributo do tipo B
- Composição é um mecanismo de reuso caixa-preta



Prefira Composição a Herança

- Implementação de uma classe Stack

Solução via Herança:

```
class Stack extends ArrayList {  
    ...  
}
```

Solução via Composição:

```
class Stack {  
    private ArrayList elementos;  
    ...  
}
```

Princípio do Menor Privilégio (Demeter)

- A implementação de um método deve invocar apenas os seguintes outros métodos:
 - de sua própria classe
 - de objetos passados como parâmetros
 - de objetos criados pelo próprio método
 - de atributos da classe do método

Falar apenas com seus amigos

Princípio do Menor Privilégio (Demeter)

- Exemplo

```
class PrincipioDemeter {  
  
    T1 attr;  
  
    void f1() {  
        ...  
    }  
  
    void m1(T2 p) { // método que segue Demeter  
        f1();           // caso 1: própria classe  
        p.f2();          // caso 2: parâmetro  
        new T3().f3();    // caso 3: criado pelo método  
        attr.f4();        // caso 4: atributo da classe  
    }  
  
    void m2(T4 p) { // método que viola Demeter  
        p.getX().getY().getZ().doSomething();  
    }  
  
}
```

Princípio Aberto/Fechado

- A classe deve ser fechada para modificações e aberta para extensões
 - Novas regras devem ser obtidas por código novo sem alterar o existente
 - “Programe para interfaces”
-

Princípio Aberto/Fechado

Viola o OCP? Satisfaz o OCP?

```
double calcTotalBolsas(Aluno[] lista) {  
    double total = 0.0;  
    foreach (Aluno aluno in lista) {  
        if (aluno instanceof AlunoGrad) {  
            AlunoGrad grad = (AlunoGrad) aluno;  
            total += "código que calcula bolsa de grad";  
        }  
        else if (aluno instanceof AlunoMestrado) {  
            AlunoMestrado mestrando = (AlunoMestrado) aluno;  
            total += "código que calcula bolsa de mestrando";  
        }  
    }  
    return total;  
}
```

Princípio Aberto/Fechado

```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);  
  
System.out.println(nomes);  
// resultado: ["alexandre","joao","maria","ze"]
```

```
Comparator<String> comparador = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};  
Collections.sort(nomes, comparador);  
  
System.out.println(nomes);  
// resultado: [ze, joao, maria, alexandre]
```

Princípio de Substituição de Liskov

- Uma classe base deve poder ser substituída pela sua classe derivada sem que isso afete a execução correta do programa



Princípio de Substituição de Liskov

- Exemplo






```
class A {  
    int soma(int a, int b) {  
        return a+b;  
    }  
}
```

```
class B extends A {  
  
    int soma(int a, int b) {  
        String r = String.valueOf(a) + String.valueOf(b);  
        return Integer.parseInt(r);  
    }  
}
```

```
class Cliente {  
  
    void f(A a) {  
        ...  
        a.soma(1,2); // pode retornar 3 ou 12  
        ...  
    }  
}
```

Trabalho Prático - TP2

- TRABALHO PRÁTICO - TP2 (Aula Prática 07/12/2020)
 - Baixar código
 - Refatorar aplicando princípio S.O.L.I.D.

- >  solid.to.refactor.dip.ruim
- >  solid.to.refactor.isp.ruim
- >  solid.to.refactor.lsp.ruim
- >  solid.to.refactor.ocp.ruim
- ✓  solid.to.refactor.sip.ruim



Obrigado!

Por hoje é só pessoal...

Dúvidas?



qpg4p5x



ismaylesantos@great.ufc.br



@IsmayleSantos
