

# Princípios de Projeto

Profa. Fabíola S. F. Pereira

[fabiola.pereira@ufu.br](mailto:fabiola.pereira@ufu.br)

**Na aula passada...**

# Propriedades de “bons projetos” de software


- Integridade Conceitual
- Ocultamento de Informação
- Coesão
- Acoplamento

# **Princípios de Projeto**

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



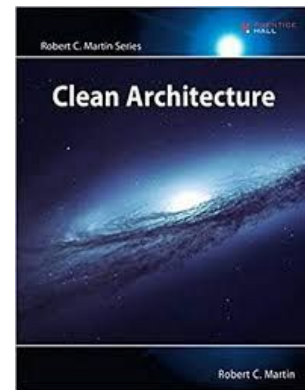
"Diretriz"



Consequência (o que  
vamos ganhar  
seguindo o princípio)

# Princípios SOLID

- **S**ingle Responsibility Principle
- **O**pen Closed/Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle



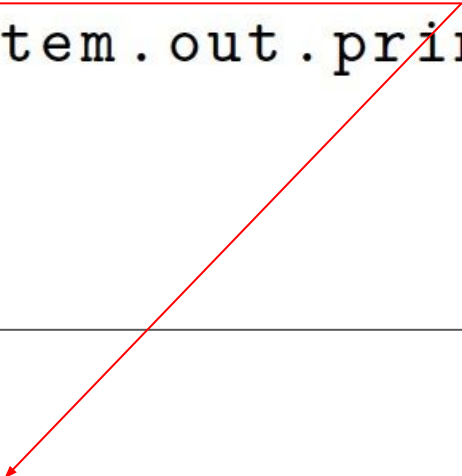
Robert Martin

# **(1) Princípio da Responsabilidade Única**

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

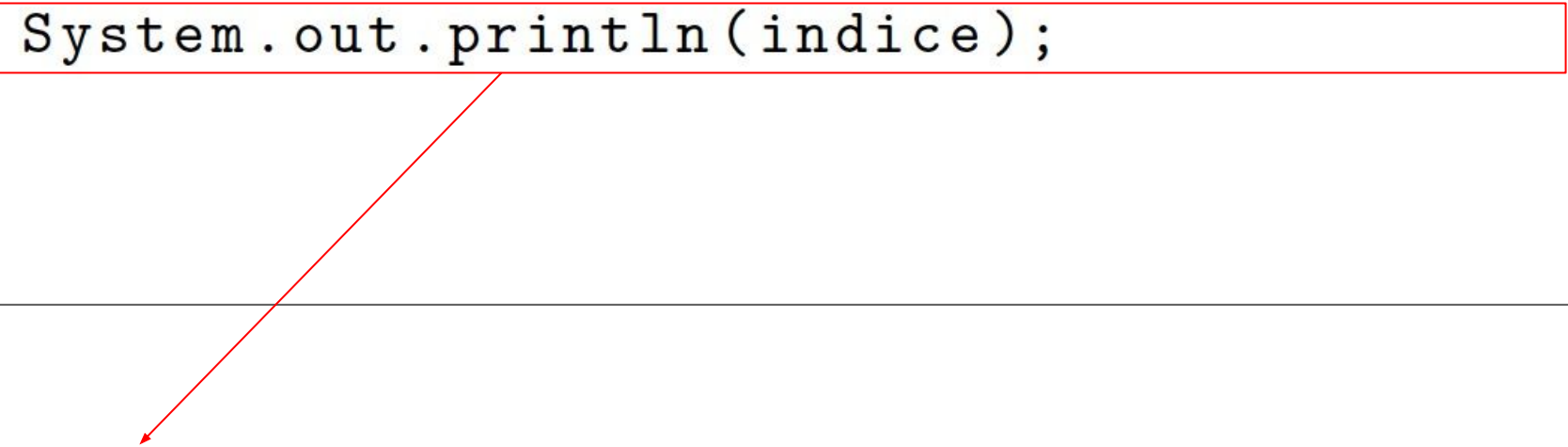


```
class Disciplina {  
  
    void calculaIndiceDesistencia() {  
        indice = "calcula índice de desistência"  
        System.out.println(indice);  
    }  
  
}
```



Responsabilidade #1: **calcular** índice de desistência

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



Responsabilidade #2: **imprimir** índice de desistência

Agora versão com **separação de responsabilidades**

```
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;  
    }  
  
}
```

```
class Console {  
  
    void imprimeIndiceDesistencia(Disciplina disciplina) {  
        double indice = disciplina.calculaIndiceDesistencia();  
        System.out.println(indice);  
    }  
  
    Uma única responsabilidade: interface com o usuário  
}
```

```
class Disciplina {  
  
    double calculaIndiceDesistencia() {  
        double indice = "calcula índice de desistência"  
        return indice;  
    }  
  
}
```

```
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;  
    }  
  
}
```

Uma única responsabilidade: "lógica ou regra de negócio"

# Vantagens

- Classe de negócio (`Disciplina`) pode ser usada por mais de uma classe de interface (`Console`, `WebApp`, `MobileApp` ...)
- Divisão de trabalho:
  - Classe de interface: frontend dev
  - Classe de negócio: backend dev

---

## Princípio de Projeto

---

Responsabilidade Única

---


## Propriedade de Projeto

---

Coesão

---

  
"Diretriz"

  
Consequência (o que  
vamos ganhar  
seguindo o princípio)



## **(2) Princípio da Segregação de Interfaces**

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

# Segregação de Interfaces

- Interfaces devem ser pequenas, coesas e específicas para cada tipo de cliente
- Caso particular do princípio anterior, mas voltado para interfaces

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

Interface genérica: trata de funcionários CLT e de funcionários públicos

O que "getSIAPE" retorna para funcionários CLT?

Agora versão que atende **segregação de interfaces**

```
interface Funcionario {  
    double getSalario();  
    ...  
}
```

```
interface FuncionarioCLT extends Funcionario {  
    double getFGTS();  
    ...  
}
```

```
interface FuncionarioPublico extends Funcionario {  
    int getSIAPE();  
    ...  
}
```

```
interface Funcionario {  
    double getSalario();  
    ...  
}
```

Comum para todos funcionários

```
interface FuncionarioCLT extends Funcionario {  
    double getFGTS();  
    ...  
}
```

```
interface FuncionarioPublico extends Funcionario {  
    int getSIAPE();  
    ...  
}
```

```
interface Funcionario {  
    double getSalario();  
    ...  
}
```

```
interface FuncionarioCLT extends Funcionario {  
    double getFGTS();  
    ...  
}
```

Específica para funcionários CLT

```
interface FuncionarioPublico extends Funcionario {  
    int getSIAPE();  
    ...  
}
```



```
interface Funcionario {  
    double getSalario();  
    ...  
}
```

```
interface FuncionarioCLT extends Funcionario {  
    double getFGTS();  
    ...  
}
```

```
interface FuncionarioPublico extends Funcionario {  
    int getSIAPE();  
    ...  
}
```

Específica para funcionários públicos

---

## Princípio de Projeto

---

---

## Propriedade de Projeto


---

Segregação de Interfaces

Coesão

---

  
"Diretriz"

  
Consequência (o que  
vamos ganhar  
seguindo o princípio)

### **(3) Princípio da Inversão de Dependências**

# Inversão de Dependências

- Na verdade, vamos chamar esse princípio de "**Prefira Interfaces a Classes**"
- Pois transmite melhor a sua ideia!

```
interface I { ... }  
  
class C1 implements I {  
    ...  
}  
  
class C2 implements I {  
    ...  
}
```

```
class Cliente {  
    I i;  
  
    Cliente (I i) {  
        this.i = i;  
        ...  
    }  
    ...  
}
```

Nos clientes, quando declarar  
variáveis ou parâmetros **prefira  
sempre uma interface**

Ou seja, use I em vez de C1 ou C2

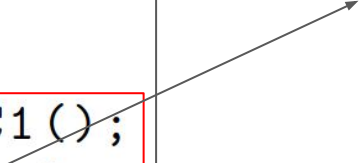
# Por que?

- Cliente funciona com qualquer classe que implementa I
- Isto é, com objetos das classes C1 e C2
- E também com uma nova classe (por exemplo, C3) que venha a ser criada

# Exemplo

```
class Main {  
  
    void main () {  
        C1 c1 = new C1();  
        new Cliente(c1);  
        ...  
        C2 c2 = new C2();  
        new Cliente(c2);  
        ...  
    }  
}
```

Cliente sendo instanciado  
com objeto da classe C1





# Exemplo

```
class Main {  
  
    void main () {  
        C1 c1 = new C1();  
        new Cliente(c1);  
        ...  
        C2 c2 = new C2();  
        new Cliente(c2);  
        ...  
    }  
}
```

Objeto da mesma classe (Cliente),  
mas instanciado com objeto do tipo C2

---

## Princípio de Projeto

---

---

## Propriedade de Projeto


---

Inversão de Dependências

Acoplamento

---

  
"Diretriz"

  
Consequência (o que  
vamos ganhar  
seguindo o princípio)

## **(4) Prefira Composição a Herança**

# Contexto Histórico

- Na década de 80, quando orientação a objetos tornou-se popular, as pessoas começaram a "abusar" de herança
- Achavam que herança iria ser uma bala de prata, promover reuso em larga escala, etc.

# Herança

- Relação "é-um"
- Exemplo: MotorGasolina **é-um** Motor
- No código:

```
class MotorGasolina extends Motor {  
  
    ... // herda atributos e métodos de motor  
  
}
```

# Composição

- Relação "possui"
- Exemplo: Painel **possui** ContaGiros
- No código:

```
class Painel {  
    ContaGiros cg; // possui um atributo  
    ...  
}
```

Prefira Composição a Herança  $\Rightarrow$  não force o  
uso de herança

# Uso "forçado" de herança

Herança

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



# Uso "forçado" de herança

Herança

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

em vez de:

Composição

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

---

## Princípio de Projeto

---

---

## Propriedade de Projeto

---


Prefira Composição a Herança

Acoplamento

---



"Diretriz"



Consequência (o que  
vamos ganhar  
seguindo o princípio)

## **(5) Princípio de Demeter**

# Demeter

- Demeter era o nome de um grupo de pesquisa de uma universidade norte-americana
- Evite longas "cadeias" de chamadas de métodos
- Exemplo:

```
obj.getA().getB().getC().getD().getOqueEuPreciso();
```



objetos de passagem

# Motivo

- Longas cadeias de chamadas quebram "encapsulamento"
- Não quero passar por A, B, C, D até obter que eu preciso
- Elos intermediários tornam a chamada frágil

```
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();
```

```
    }
```

Define quais chamadas de métodos são "permitidas" no corpo de um método

```
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();  
    }  
}
```

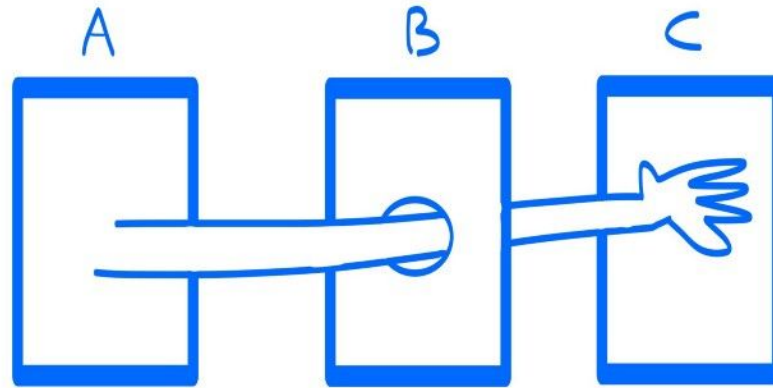
```
class PrincipioDemeter {  
  
    T1 attr;  
  
    void f1() {  
        ...  
    }  
  
    void m1(T2 p) { // método que segue Demeter  
        f1();           // caso 1: própria classe  
        p.f2();          // caso 2: parâmetro  
        new T3().f3();    // caso 3: criado pelo método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



```
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();  
    }  
}
```

```
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();  
    }  
}
```

Cenário **não recomendado** pelo Princípio de Demeter:



# Mas cuidado

- Demeter -- e demais princípios -- são recomendações
- Não devemos ser radicais e achar que cadeias de chamadas de métodos são totalmente proibidas
- Casos isolados podem existir e não justificar uma mudança no projeto

---

## Princípio de Projeto

---

---

## Propriedade de Projeto


---

Demeter

Ocultamento de Informação

---

  
"Diretriz"

  
Consequência (o que  
vamos ganhar  
seguindo o princípio)

## **(6) Princípio Aberto/Fechado**

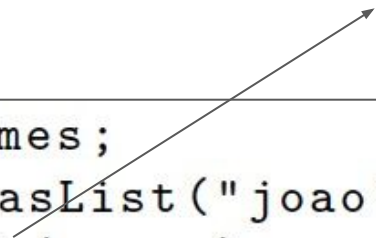
# Princípio Aberto/Fechado

- Proposto por Bertrand Meyer
- Ideia: uma classe deve estar **fechada** para modificações, mas **aberta** para extensões



# Exemplo

Ordena a lista passada com parâmetro



```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);
```



# Exemplo

Ordena uma lista passada com parâmetro



```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);
```

```
System.out.println(nomes);  
// resultado: ["alexandre","joao","maria","ze"]
```

Mas agora eu quero ordenar as strings da lista pelo seu tamanho, isto é, pelo número de chars

Será que o método sort está aberto (preparado) para permitir essa extensão?

Mas, mantendo o seu código fechado, isto é, sem ter que mexer no seu código

# Felizmente, sim!

```
Comparator<String> comparador = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};  
Collections.sort(nomes, comparador);
```

lista de  
strings

Objeto com um método **compare**, que  
vai comparar duas strings.  
Não existe almoço grátis, cliente tem  
que implementar esse método

# Explicando melhor

- Suponha que você vai implementar uma classe
- Usuários ou clientes vão querer usar a classe (óbvio!)
- Mas vão querer também customizar, parametrizar, configurar, flexibilizar e estender a classe!
- Você deve se antecipar e tornar possível tais extensões
- Mas sem que os clientes tenham que alterar o código da classe

Como tornar uma classe **aberta** a extensões, mas mantendo o seu código **fechado** para modificações?

- Parâmetros
- Funções de mais alta ordem
- Padrões de projeto
- Herança
- etc

Resumindo: ao implementar uma classe, pense em pontos de extensão!

---

## Princípio de Projeto

---

---

## Propriedade de Projeto

---


Aberto/Fechado

Extensibilidade

---



"Diretriz"



Consequência (o que  
vamos ganhar  
seguindo o princípio)



## **(7) Princípio de Substituição de Liskov**

# Princípio de Substituição de Liskov

- Nome é uma referência à Profa. Barbara Liskov
- Princípio define boas práticas para uso de herança
- Especificamente, boas práticas para redefinição de métodos em subclasses



Primeiro: vamos entender o termo "substituição"

```
void f(A a) {  
    ...  
    a.g(int n);  
    ...  
}
```

```
void f(A a) {  
    ...  
    a.g();  
    ...  
}
```

```
f(new B1()); // f pode receber objetos da subclasse B1  
...  
f(new B2()); // e de qualquer outra subclasse de A, como B2  
...  
f(new B3()); // e B3
```

```
void f(A a) {  
    ...  
    a.g();  
    ...  
}
```



- Tipo A pode ser substituído por B1, B2, B3,...
- Desde que eles sejam subclasses de A
- Em tempo de execução, método g chamado vai ser aquele de B1, B2, B3, etc.

# Princípio de Substituição de Liskov

- Redefinições de métodos em subclasses são possíveis
- Mas devem preservar o contrato do método da superclasse
- Preservar o contrato: tanto faz chamar `A.g` ou `B1.g` ou `B2.g` ou `B3.g`

Para concluir, um exemplo do dia-a-dia



- Suponha um Médico  $A$  plantonista em um hospital
- Em um fim de semana, ele não poderá fazer seu plantão
- Então, ele pede para um colega  $B1$  substituí-lo
- Quando a substituição vai funcionar?
  - Quando  $B1$  tiver pelo menos a mesma competência de  $A$
  - A substituição não vai afetar o funcionamento do hospital
  - Substituição de Liskov

- Quando a substituição **não** vai funcionar?
  - Exemplo: quando  $A$  for um Clínico Geral e  $B_1$  um Pediatra
  - Essa substituição vai prejudicar o funcionamento do hospital

---

Princípio de Projeto

---

---

Propriedade de Projeto

---


Substituição de Liskov

Extensibilidade

---



"Diretriz"



Consequência (o que  
vamos ganhar  
seguindo o princípio)

## Revisão dos conceitos



---

## Princípio de Projeto

---

Responsabilidade Única

Segregação de Interfaces

Inversão de Dependências

Prefira Composição a Herança

Demeter

Aberto/Fechado

Substituição de Liskov

---

## Propriedade de Projeto

---

Coesão

contra-exemplo: Disciplina: calcula e imprime índice de desistência

Coesão

Funcionario, FuncionarioPublico, FuncionarioCLT

Acoplamento

Prefira Interfaces a classes: class Cliente { I i; },  
Cliente (new C1()), Cliente( new C2() )

Acoplamento

contra-exemplo: class Stack extends ArrayList

Ocultamento de Informação

a.get().get().get()

Extensibilidade

Collections.sort , compare


Extensibilidade

subclasses, médico

---



"Diretriz"



Consequência (o que  
vamos ganhar  
seguindo o princípio)

# ATIVIDADE 1

```
void onclick() {  
  
    num1 = textfield1.value();  
    c1 = BD.getConta(num1)  
  
    num2 = textfield2.value();  
    c2 = BD.getConta(num2)  
  
    valor = textfield3.value();  
    beginTransaction();  
  
    try {  
        c1.retira(valor);  
        c2.deposita(valor);  
        commit();  
    }  
    catch() {  
        rollback();  
    }  
}
```

- Qual o princípio está sendo violado pelo seguinte código?

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter
- F. Aberto/Fechado
- G. Substituição de Liskov

# ATIVIDADE 2

- Qual o princípio está sendo violado pelo seguinte código?

```
void sendMail(ContaBancaria conta, String msg) {  
    Cliente cliente = conta.getCliente();  
    String mail = cliente.getMailAddress();  
    "Envia mail"  
}
```

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter
- F. Aberto/Fechado
- G. Substituição de Liskov

# ATIVIDADE 3

- Qual o princípio está sendo violado pelo seguinte código?

```
void imprimeDataContratacao(Funcionario func) {  
    Date data = func.getDataContratacao();  
    String msg = data.format();  
    System.out.println(msg);  
}
```

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter
- F. Aberto/Fechado
- G. Substituição de Liskov



# ATIVIDADE 4

```
class A {  
    int f(int x) { // pre: x > 0  
        ...  
        return exp;  
    } // pos: exp > 0  
    ...  
}  
  
class B extends A {  
    int f(int x) { // pre: x > 10  
        ...  
        return exp;  
    } // pos: exp > -50  
    ...  
}  
  
main() {  
    A a = new A();  
    a.f(5);  
    A b = new B();  
    b.f(5);  
}
```

- As pré-condições de um método são expressões booleanas envolvendo seus parâmetros (e, possivelmente, o estado de sua classe) que devem ser verdadeiras antes da sua execução. De forma semelhante, as pós-condições são expressões booleanas envolvendo o resultado do método.
- Qual o princípio está sendo violado pelo seguinte código?
  - A. Responsabilidade Única
  - B. Segregação de Interfaces
  - C. Inversão de Dependências
  - D. Prefira Composição a Herança
  - E. Demeter
  - F. Aberto/Fechado
  - G. Substituição de Liskov

# Créditos

- CC-BY: Slides adaptados de Marco Tulio Valente, ESM

FIM

# ATIVIDADE 1

```
void onclick() {  
  
    num1 = textfield1.value();  
    c1 = BD.getConta(num1)  
  
    num2 = textfield2.value();  
    c2 = BD.getConta(num2)  
  
    valor = textfield3.value();  
    beginTransaction();  
  
    try {  
        c1.retira(valor);  
        c2.deposita(valor);  
        commit();  
    }  
    catch() {  
        rollback();  
    }  
}
```

- Qual o princípio está sendo violado pelo seguinte código?

- A. **Responsabilidade Única**
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter
- F. Aberto/Fechado
- G. Substituição de Liskov

# ATIVIDADE 2

- Qual o princípio está sendo violado pelo seguinte código?

```
void sendMail(ContaBancaria conta, String msg) {  
    Cliente cliente = conta.getCliente();  
    String mail = cliente.getMailAddress();  
    "Envia mail"  
}
```

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter**
- F. Aberto/Fechado
- G. Substituição de Liskov

# ATIVIDADE 3

- Qual o princípio está sendo violado pelo seguinte código?

```
void imprimeDataContratacao(Funcionario func) {  
    Date data = func.getDataContratacao();  
    String msg = data.format();  
    System.out.println(msg);  
}
```

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter**
- F. Aberto/Fechado
- G. Substituição de Liskov

# ATIVIDADE 4

```
class A {  
    int f(int x) { // pre: x > 0  
        ...  
        return exp;  
    } // pos: exp > 0  
    ...  
}  
  
class B extends A {  
    int f(int x) { // pre: x > 10  
        ...  
        return exp;  
    } // pos: exp > -50  
    ...  
}  
  
main() {  
    A a = new A();  
    a.f(5);  
    A b = new B();  
    b.f(5);  
}
```

- As pré-condições de um método são expressões booleanas envolvendo seus parâmetros (e, possivelmente, o estado de sua classe) que devem ser verdadeiras antes da sua execução. De forma semelhante, as pós-condições são expressões booleanas envolvendo o resultado do método.
- Qual o princípio está sendo violado pelo seguinte código?
  - A. Responsabilidade Única
  - B. Segregação de Interfaces
  - C. Inversão de Dependências
  - D. Prefira Composição a Herança
  - E. Demeter
  - F. Aberto/Fechado
  - G. Substituição de Liskov**