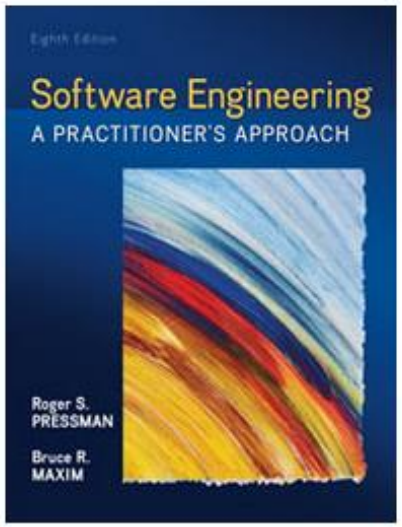
Unidade 25
Projeto de Componentes de Software

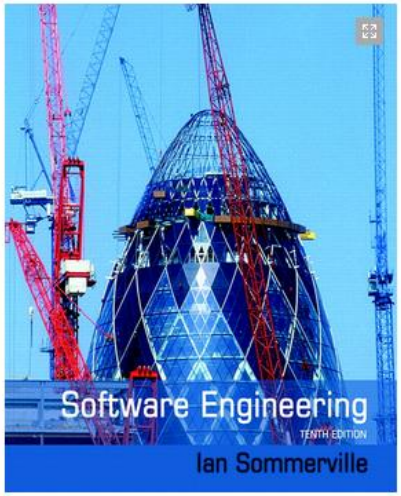
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

Bibliografia

- **Software Engineering – A Practitioner's Approach – Roger S. Pressman – Eight Edition – 2014**
- **Software Engineering – Ian Sommerville – 10th edition - 2015**
- Engenharia de Software – Uma abordagem profissional – Roger Pressman - McGraw Hill, Sétima Edição - 2011
- Engenharia de Software – Ian Sommerville – Nona Edição – Addison Wesley, 2007

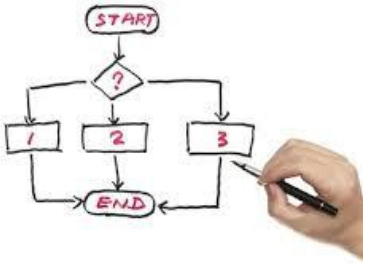


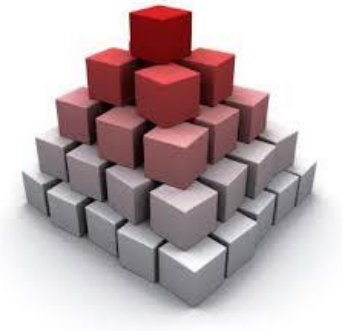
Software Engineering: A Practitioner's Approach, 8/e



Introdução

- O projeto de componentes ocorre após a **primeira iteração do projeto de arquitetura**;
- O objetivo é traduzir o modelo de projeto em **software operacional**;
- Pode-se representar o projeto de componentes por meio de uma linguagem de programação ou por alguma representação intermediária (**pseudocódigo**).





O que é um componente de Software?

Componente de Software

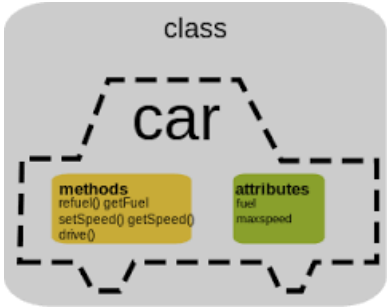
- "Uma parte **modular**, possível de ser **implementada** e **substituível** de um sistema que **encapsula** implementação e expõe um conjunto de **interfaces**". (OMG)

OMG
WE SET THE STANDARD®
http://www.omg.org/

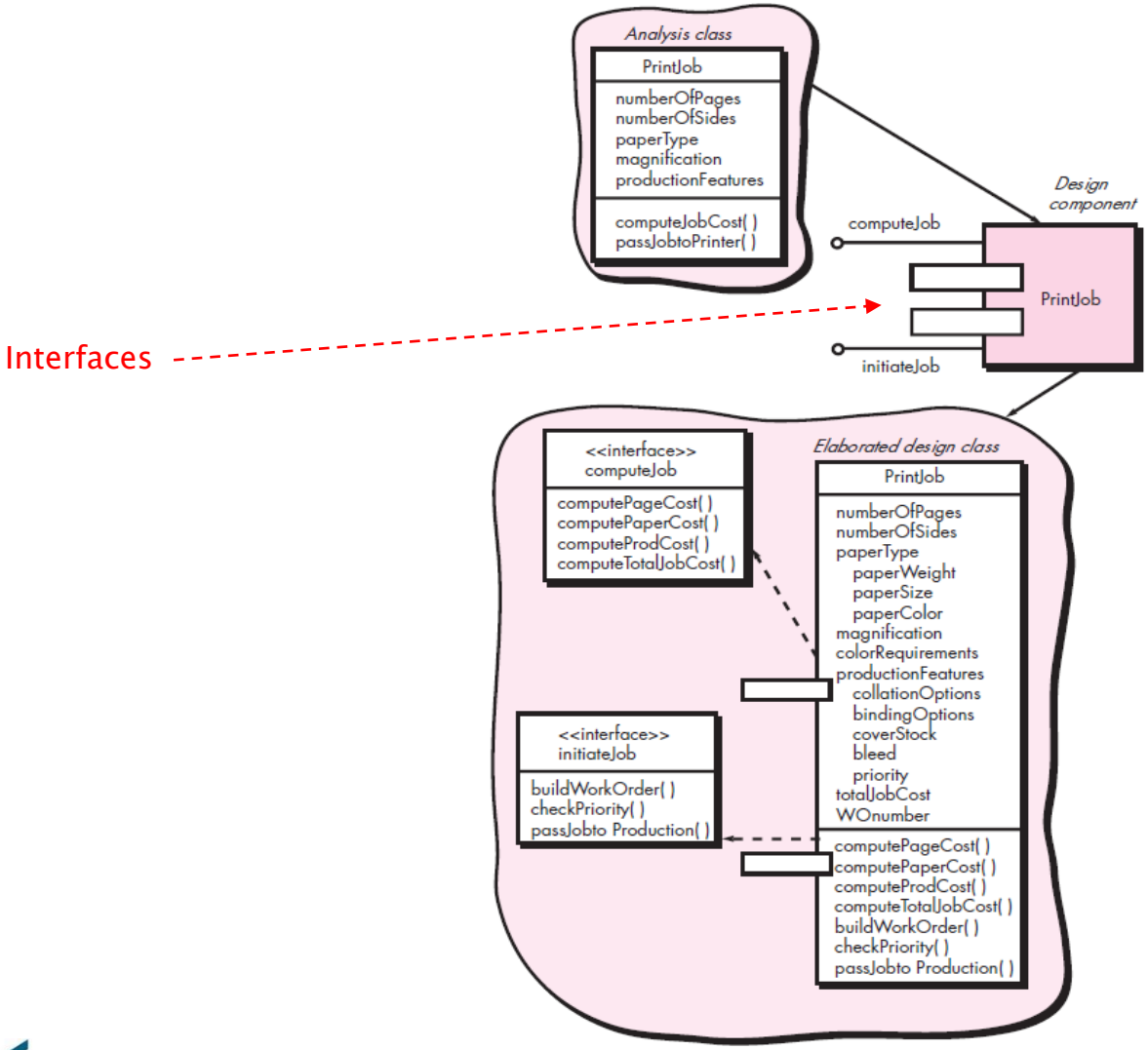
- Componentes preenchem a Arquitetura de Software e, como consequência, desempenham um papel para alcançar os objetivos e requisitos do software a ser construído.

Componente – Visão OOP

- No contexto de Engenharia de Software Orientada a Objetos, um componente contém um conjunto de **classes colaborativas**;
- Classes devem conter todos os atributos e operações relevantes à sua implementação;
- Adicionalmente, todas as **interfaces** também precisam ser definidas;
- Assim, a partir do modelo de requisitos elaboram-se as classes de domínio relacionadas ao problema (análise), bem como as classes de **infraestrutura** (para componentes que oferecem suporte a serviços para o domínio do problema.



Componente de Projeto



UML – Diagrama de Componente

Componente de projeto

Fonte: **Pressman**

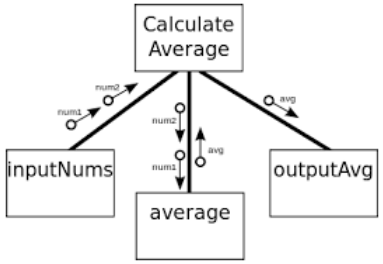
Componente – Visão Tradicional

- Um componente é o elemento **funcional** de um programa que incorpora a **lógica** de processamento, as **estruturas de dados** internas e uma **interface** que habilita o componente a ser chamado e que dados sejam passados a ele;

- Um componente tradicional também é chamado de **módulo**;

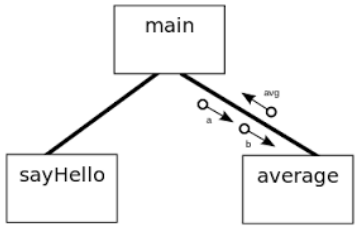
Componente – Visão Tradicional

- Um componente de software reside na arquitetura do software e presta três importantes papéis:
  - **Componente de Controle**: coordena a chamada de todos os demais componentes do domínio do problema.
  - **Componente de Domínio**: implementa a funcionalidade solicitada pelo cliente;
  - **Componente de Infraestrutura**: responsável por funções que darão suporte ao processamento necessário no domínio do problema.

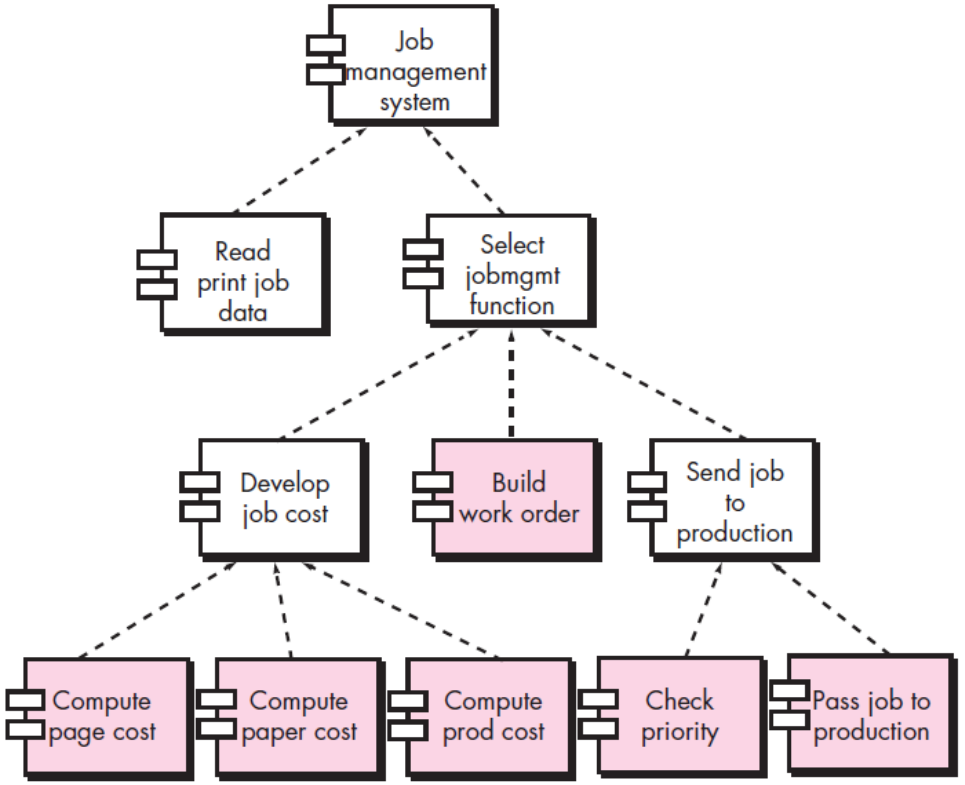


Componente – Visão Tradicional

- Componentes de software tradicionais são obtidos do **modelo de análise**, com base na modelagem de fluxo dos dados;
- Componentes (**módulos**) de controle residem próximo do **alto** da hierarquia (arquitetura de programas);
- Componentes de **domínio** do problema tendem a residir mais **próximos** da parte **inferior** da hierarquia.

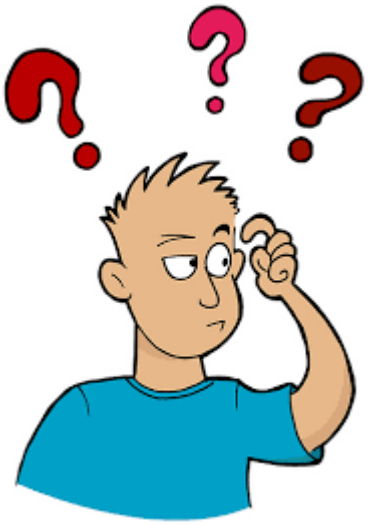


Componente – Visão Tradicional



Fonte: **Pressman**

Componentes sempre são desenvolvidos a partir do zero?



Fonte: **Pressman**

CBSE – Component-Based Software Enginnering

- A Engenharia de Software tem enfatizado a necessidade de se construir sistemas que façam uso de componentes de software ou padrões de projeto existentes;
- Desse modo, a arquitetura do software pode ser preenchida com componentes obtidos a partir de um **catálogo**, tendo-se em mente aspectos de **reusabilidade** de software.

Diretrizes – Projeto de Componentes baseados em Classes

- O projeto de componentes apoia-se nas informações desenvolvidas como parte do **modelo de requisitos** e representadas como parte do modelo da arquitetura de software;

- Ao se empregar a abordagem orientada a objetos, o projeto de componentes focaliza a elaboração de **classes específicas do domínio do problema** e a definição e o **refinamento de classes de infraestrutura** (suporte);

- Há alguns princípios de projeto aplicáveis ao projeto de componentes, cuja aplicação pode tornar o projeto **mais fácil de ser modificado**, **reduzindo-se**, assim, **a propagação de efeitos colaterais** na ocorrência de modificações.

Princípio do aberto-fechado (OCP)

- **OCP – Open-Closed Principle**
  - "Um módulo (componente) deve ser aberto para extensão, mas fechado para modificações" [Martin,2000]

Princípio do aberto-fechado (OCP)

- Esse princípio pode parecer uma contradição, mas **representa uma das características mais importantes de um bom projeto de componentes de software**;
- De acordo com o princípio, deve-se especificar o componente para permitir que ele seja **estendido** (em seu domínio funcional) **sem a necessidade de se fazer modificações internas** (em nível de código ou lógica) no próprio componente;
- Para tanto, devem ser criadas **abstrações** que servem como um buffer entre a funcionalidade que provavelmente será estendida e a classe de projeto em si.

Conceitos atrelados ao Princípio do aberto-fechado (OCP)

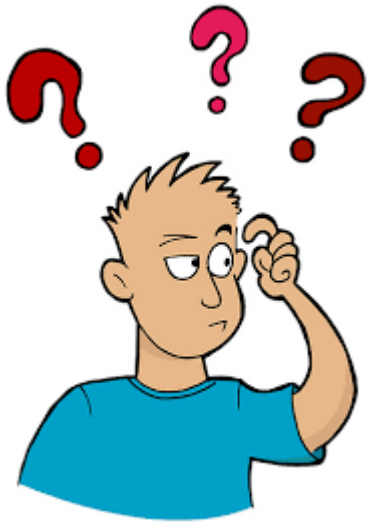
- **Extensibilidade**: É uma das chaves da orientação a objetos. Quando um novo comportamento ou funcionalidade precisar ser adicionada, espera-se que as existentes sejam **estendidas** e não **alteradas**, assim o código original permanece intacto e confiável enquanto as novas são implementadas através de extensibilidade. Criar código extensível é uma responsabilidade do desenvolvedor maduro, uma vez que se utiliza design duradouro para um software de boa qualidade e manutenibilidade.

- **Abstração**: Quando se aprende sobre orientação a objetos com certeza ouve-se sobre abstração. É ela que permite que o princípio OCP funcione. Se um software possui abstrações bem definidas, estará aberto para extensão.

Princípio do aberto-fechado (OCP) - Exemplo

```
package maua;

public enum TipoDebito {
        ContaCorrente, Poupanca;
}

public class Debito {

        public void Debitar(int valor, TipoDebito tipo) {

                if (tipo == TipoDebito.Poupanca) {

                        // Debita Poupanca
                }

                if (tipo == TipoDebito.ContaCorrente)  {

                        // Debita ContaCorrente
                }

        }
}
```

Como proceder para alterar a classe, caso tenha surgido um novo tipo de débito em conta (conta investimento)?



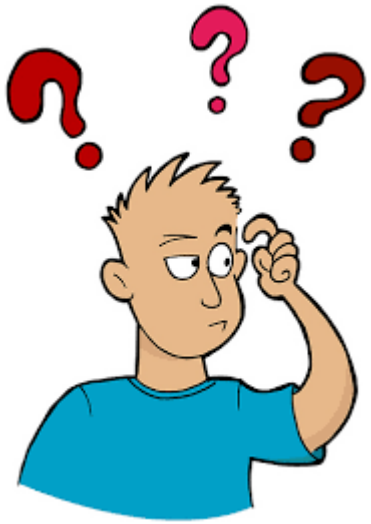
Fácil! Basta acrescentar um IF na classe !

Princípio do aberto-fechado (OCP) - Exemplo

```
public class Debito {

        public void Debitar(int valor, TipoDebito tipo) {

                if (tipo == TipoDebito.Poupanca) {
                        // Debita Poupanca
                }

                if (tipo == TipoDebito.ContaCorrente)  {
                        // Debita ContaCorrente
                }

                if (tipo == TipoDebito.Investimento)  {
                        // Debita Investimento
                }

        }
}
```

manutenção

Qual o problema de incluir mais um IF na classe?

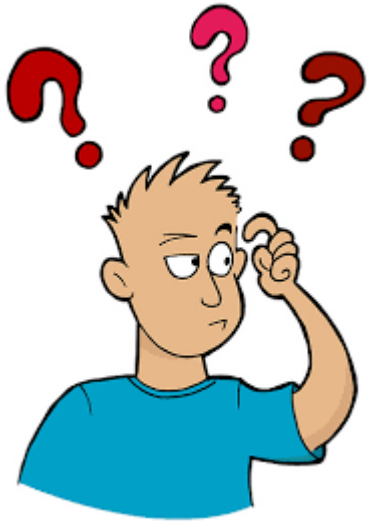


Princípio do aberto-fechado (OCP) - Exemplo

- Ao se modificar a classe colocando-se mais um IF de validação, além da **publicação da nova versão da classe**, corre-se o risco de se introduzir alguns **bugs** em uma classe **que já estava funcionando**.

- Além de ter que testar todos os tipos de débito em conta, um **bug** introduzido nesta modificação não afetaria apenas o débito em conta investimento, mas poderia **causar interrupção em todos os tipos de débito**.

Como então se deve aplicar o principio OCP?



Princípio do aberto-fechado (OCP) - Exemplo

- Deve-se implementar uma **abstração** bem definida, onde todas as extensões implementam **suas próprias regras de negócio** sem necessidade de modificar uma funcionalidade devido mudança ou inclusão de outra.

```
package maua;

public abstract class Debito {

        public abstract void Debitar(int valor);

}
```

Princípio do aberto-fechado (OCP) - Exemplo

Classes Inalteradas

```
public class DebitoContaCorrente extends Debito {
        public void  Debitar(int valor)    {
        // Debita Conta Corrente
  }
}
public class DebitoContaPoupanca extends Debito {
        public void  Debitar(int valor)    {
        // Debita Conta Poupanca
 }
}
```

Classe Inserida

```
public class DebitoContaInvestimento extends Debito {
        public void  Debitar(int valor)    {
        // Debita Conta Investimento
 }
}
```

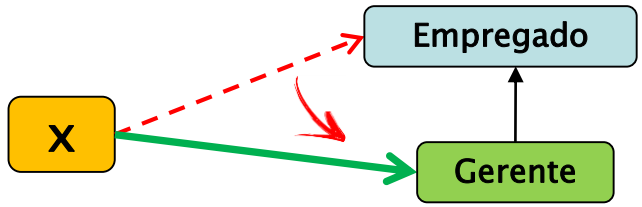
Princípio do aberto-fechado (OCP) – Considerações

- O tipo de débito em conta de investimento foi implementado **sem modificar classes existentes**, usando-se apenas a extensão.
- Além disso, o código está mais **legível** e mais fácil para se aplicar **cobertura de testes de unidade**.
- Este princípio nos atenta para um melhor design, tornando o software mais extensível e facilitando sua evolução sem afetar a **qualidade** do que já está desenvolvido.
- A implementação apresentada corresponde ao **Pattern Strategy**, no qual define-se novas operações sem alterar as classes dos elementos sobre os quais opera.

Princípio de substituição de Liskov (LSP)

LSP – Princípio de Substituição de Liskov

- "As subclasses devem ser **substitutas** de suas classes-base" [Martin,2000]

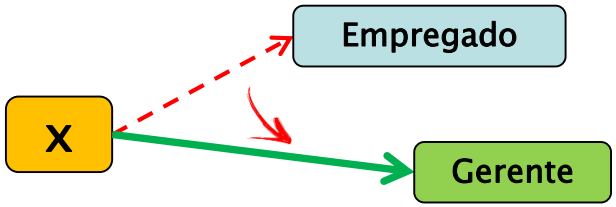


Princípio de substituição de Liskov (LSP)

- Esse princípio de projeto sugere que um componente que usa uma **classe-base deveria continuar a funcionar** apropriadamente caso uma classe derivada da classe-base fosse passada para o componente em seu lugar;
- O **LSP** exige que qualquer classe derivada de uma classe-base deva honrar qualquer **contrato** implícito entre a classe-base e os componentes que a usam;
- Nesse contexto, um "**contrato**" é uma pré-condição que deve ser verdadeira antes de o componente usar uma classe-base;
- Assim, ao se criar classes derivadas, deve-se certificar que **pré** e **pós-condições sejam atendidas**.

Princípio de substituição de Liskov (LSP)

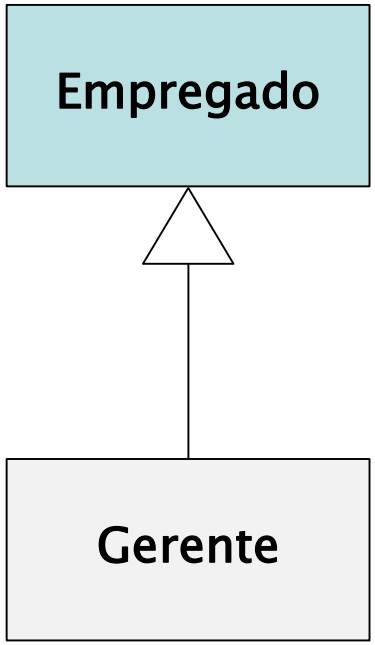
- Instâncias da **superclasse** podem ser substituídas por instâncias da **subclasse**;
- O princípio está fortemente relacionado ao conceito de Herança em Programação Orientada a Objetos;
- Herança permite que se obtenha reutilização de código;
- Porém, na prática, pode haver armadilhas criadas pela hierarquia de classes.



Fonte: **Pressman**

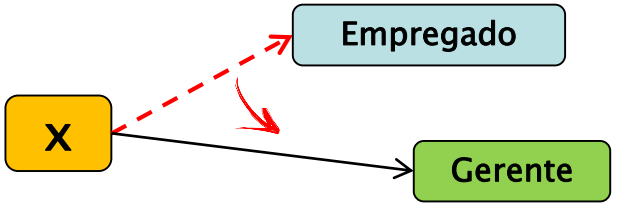
Conceitos de Herança

- A regra "**is-a**" estabelece que todo objeto da sub-classe é também um objeto da superclasse.
- Por exemplo, todo **gerente** também é um **empregado**.
- Naturalmente, o oposto não é verdade – nem todo empregado é gerente.



Princípio de substituição de Liskov (LSP)

- O **princípio da Substituição** estabelece que você pode usar um objeto de uma **subclasse** sempre que o programa espera um objeto da **superclasse**.
- Exemplo:



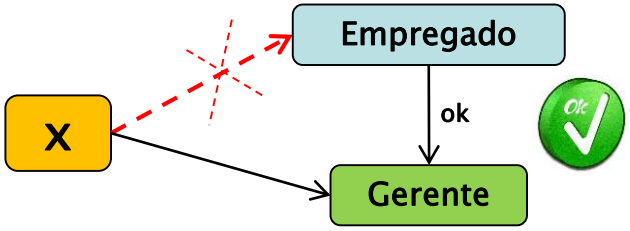

```
package maua;

public class LSP_01 {

        public static void main(String[] args) {

                Empregado x = new Empregado("Marcos", 2300.5,55150) ;

                x = new Gerente("Mike", 9300.5, 51234, 500);

                System.out.println(x.getDetalhes() ) ;
        }
}
```

Variáveis Polimórficas

- O **princípio da Substituição** estabelece que, sob herança, referências à objetos da superclasse são polimórficas, ou seja uma variável do tipo Empregado (**superclasse**) pode fazer referência à uma variável do tipo Gerente (**subclasse**).

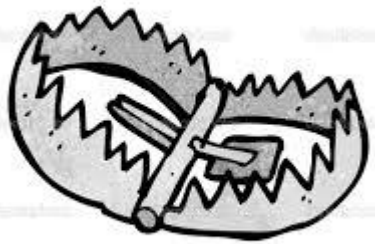
x é do tipo Empregado !

Todo Gerente também é Empregado !




```
Empregado x = new Empregado("Marcos", 2300.5,55150) ;

x = new Gerente("Mike", 9300.5, 51234, 500);
```

No entanto, herança pode causar armadilhas criadas por hierarquia de classes !



Princípio de substituição de Liskov (LSP)

- O **LSP** exige que qualquer classe derivada de uma classe-base deva honrar qualquer **contrato** implícito entre a classe-base e os componentes que a usam;
- Nesse contexto, um "**contrato**" é uma pré-condição que deve ser verdadeira antes de o componente usar uma classe-base;
- Assim, ao se criar classes derivadas, deve-se certificar que **pré** e **pós-condições** **sejam atendidas**.

Princípio de substituição de Liskov (LSP)

Exemplo – Violação da Regra

- Considere uma classe chamada **ContaComum** que, de forma simplificada, representa uma conta em um banco;
- A classe possui operações simples, como **deposita**() e **rende**().

Princípio de substituição de Liskov (LSP)

```
package maua;

public class ContaComum {

        protected double saldo;

        public ContaComum() {
           this.saldo=0;
        }

        public ContaComum(double saldo) {
           this.saldo=saldo;
        }

        public double getSaldo() {
           return saldo;
        }

        public void rende() {
           this.saldo*= 1.1;
        }
}
```

Princípio de substituição de Liskov (LSP)

Exemplo – Violação da Regra

- Suponha, como geralmente ocorre, que o **software precisa crescer**;
- Será necessário criar-se uma nova classe chamada **ContaEstudante**, que é exatamente igual a uma conta comum, porém com a diferença de que nunca é contabilizado rendimento para essa conta. Ou seja, o método **rende**() deve lançar uma exceção quando for executado por um objeto do tipo **ContaEstudante**.

Princípio de substituição de Liskov (LSP)

```
package maua;

public class ContaEstudante extends ContaComum {

        public ContaEstudante() {
          super();
        }


        public ContaEstudante (double saldo) {
          super(saldo);
        }

        public void rende() throws ContaNaoRendeException {

                throw new ContaNaoRendeException("Erro ...");
        }
}
```

Qual o problema com essa implementação?

Princípio de substituição de Liskov (LSP)

Exemplo – Violação da Regra

- É difícil visualizar-se o problema;
- Ocorreu uma quebra de contrato definida pela superclasse. O método **rende**() na superclasse **não** lança exceção, ou seja, um programa que trate de contas comuns e constas de estudante, pode **quebrar** em função de uma conta de estudante;
- Assim, uma pré-condição não foi atendida.

Princípio de substituição de Liskov (LSP)
Exemplo – Violação da Regra

```
package maua;

public class TesteContas {

	public static void main(String[] args) throws ContaNaoRendeException {

		ContaComum cc1 = new ContaComum(100.0);
		ContaEstudante ce1 = new ContaEstudante(300.0);

		ContaComum[] contas = new ContaComum[2];

		contas[0] = cc1;
		contas[1] = ce1;
		System.out.println("Saldo Anterior ao Rendimento: " + "\n");

		for (int i = 0; i < contas.length; i++) {
			System.out.println(contas[i].getSaldo());
		}

		System.out.println("\nSaldo com Rendimento: " + "\n");
		for (int i = 0; i < contas.length; i++) {
			contas[i].rende();
			System.out.println(contas[i].getSaldo());
		}
	}

}
```

Princípio de substituição de Liskov (LSP)
Exemplo – Violação da Regra

Saldo Anterior ao Rendimento:

100.0
300.0

Saldo com Rendimento:

110.0
Exception in thread "main" maua.ContaNaoRendeException: Erro ...
at maua.ContaEstudante.rende(ContaEstudante.java:16)
at maua.TesteContas.main(TesteContas.java:23)

Princípio da Inversão da Dependência (DIP)

- "Dependa de **abstrações**. Não dependa de **concretizações**" [Martin, 2000];
- **Abstração** é o lugar onde um projeto pode ser estendido **sem** grandes complicações;
- Quanto mais um componente depender de outros componentes (e não de abstrações como uma interface), **mais difícil será estendê-lo**.
- Projeto deve ter como meta baixo acoplamento e alta coesão de componentes;
- Módulos devem depender de abstrações;
- Abstrações não devem depender de detalhes;
- Detalhes devem depender de abstrações.

Quais as vantagens do Princípio da Inversão da Dependência (DIP)?

Princípio da Inversão da Dependência (DIP)
Vantagens

- Melhoria da **manutenibilidade**, pois o código fica mais flexível e reutilizável;
- Melhoria na **testabilidade**, pois não se necessita conhecer detalhes da dependência.

Princípio da Inversão da Dependência (DIP)
Exemplo

```
public class Botao   {

        private Lampada lampada;


        public  void  Acionar()    {

                if (condicao)

                    lampada.Ligar();

        }

}
```

Qual o problema com esta classe?

Um objeto Botao está fortemente acoplado a um objeto Lampada.

```
public class Botao   {

        private Lampada lampada;

        public  void  Acionar()    {

                if (condicao)

                    lampada.Ligar();

        }

}
```

A classe viola o princípio DIP

```
public class Botao   {

        private Lampada lampada;

        public  void  Acionar()    {

                if (condicao)

                    lampada.Ligar();

        }

}
```

- O design ao lado viola o **DIP**, uma vez que Botão depende de uma classe concreta Lampada.
- Ou seja, Botão conhece detalhes de implementação, ao invés de uma abstração para o design.
- Que abstração seria essa? Botão deve ser capaz de tratar alguma ação e ligar ou desligar algum dispositivo, seja ele qual for: uma lâmpada, um motor, um alarme, etc.

Aplicando DIP

```
public interface Dispositivo {

    void Ligar();

    void Desligar();

}
```

- A **abstração** está sendo implementada na Interface!

Aplicando DIP

```
public class Botao {

    private Dispositivo dispositivo;

    public void Acionar()    {

        if (condicao)
            dispositivo.Ligar();
    }
}
```

- A classe Botão agora está dependendo de uma abstração, implementada pela interface **Dispositivo**.

Aplicando DIP

```
public class Lampada implements Dispositivo {

    public void Ligar()    {

     // ligar lampada

    }

    public void Desligar()   {

     // desligar lampada

   }

}
```

- A classe Lampada implementa a interface **Dispositivo**.

Princípio da Segregação de Interfaces (ISP)

- **ISP – Princípio da Segregação de Interfaces**

  - "É melhor usar várias interfaces específicas de clientes do que uma única interface de propósito geral".

Princípio da Segregação de Interfaces (ISP)

- Este princípio afirma que uma interface não pode forçar uma classe a implementar métodos que não pertencem a ela.

```
public interface ITelefone {
    void tocar();
    void tirarFoto();
    void discar();
 }
 public  class  TelefoneComum implements ITelefone  {
    @Override
    public void tocar() {
    }
    @Override
    public void tirarFoto() {
    }
    @Override
    public void discar() {
    }
 }
```

Princípio da Segregação de Interfaces (ISP)

- Neste exemplo, a interface **ITelefone** contém métodos que não serão utilizados por todas as classes que a implementam, o que deixa a interface e suas implementações mais poluídas.

```
public interface ITelefone{
    void tocar();
    void tirarFoto();
    void discar();
 }
 public  class  TelefoneComum implements ITelefone {
    @Override
    public void tocar() {
    }
    @Override
    public void tirarFoto() {
    }
    @Override
    public void discar() {
    }
 }
```

Princípio da Segregação de Interfaces (ISP)

- O princípio da segregação estabelece que as interfaces devem ser divididas de forma a permitir que todas as implementações sejam usadas. No exemplo anterior, a criação da interface ITelefoneMultimidia, contendo o método tirarFoto(), é uma das soluções que resolvem o problema da poluição da interface

```
public interface ITelefone {
    void tocar();
    void discar();
}
public interface ITelefoneMultimidia {
    void tirarFoto();
}
```