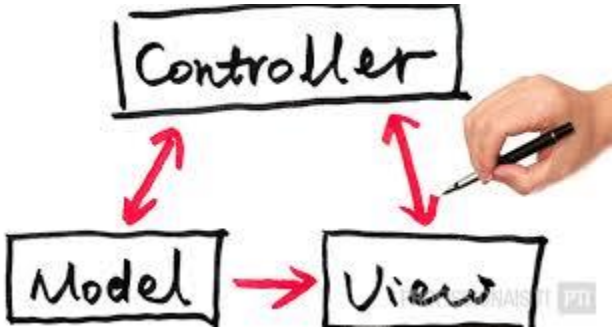
Unidade 17

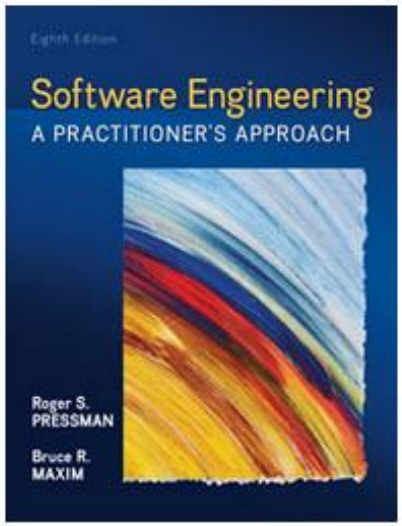
Conceitos de Arquitetura de Software
Padrões SOLID



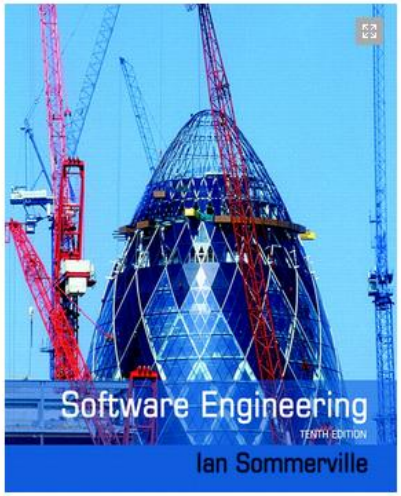
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas#@gmail.com

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

- A atividade de **projeto de software** é crítica para o êxito em Engenharia de Software;
- Por meio do projeto de software, finca-se o pé em dois mundos: O mundo da **tecnologia** e o mundo das **pessoas** e dos propósitos do ser humano. Tenta-se em um bom projeto unir-se esses dois mundos;
- Software de boa qualidade deve ter:
  - ✓ **Solidez**: um programa não deve apresentar bugs que impeçam sua operação;
  - ✓ **Comodidade**: um programa deve atender aos propósitos para os quais foi planejado;
  - ✓ **Deleite**: A experiência de se usar o programa deve ser prazerosa.

Projeto no contexto de Engenharia de Software

- O projeto de software reside no **núcleo técnico** da Engenharia de Software e é independente do modelo de processo de software adotado;
- O projeto de software inicia-se logo após o levantamento e modelagem dos requisitos e é a última ação da atividade de modelagem de software;
- Prepara a cena para a construção do software (geração de código e testes);
- É composto por **Projeto de dados/classes**, **projeto de arquitetura**, **projeto de interfaces** e **projeto de componentes**;

Diretrizes de Projeto de Software

- Arquitetura exibindo **padrões** arquiteturais reconhecíveis;
- Arquitetura composta por componentes que apresentem boas características de projeto;
- Arquitetura que possa ser implementada de forma **evolucionária**;

- Projeto deve ser **modular**, para ser mais fácil de ser testado e mantido;
- Deve conter representações distintas de **dados**, **arquitetura**, **interfaces** e **componentes**;
- Componentes devem ter **baixo** **acoplamento** (características funcionais independentes);
- Interfaces devem reduzir a complexidade das conexões entre componentes e o ambiente externo (**encapsulamento**);
- **Notação** de projeto deve efetivamente **comunicar** seu significado.

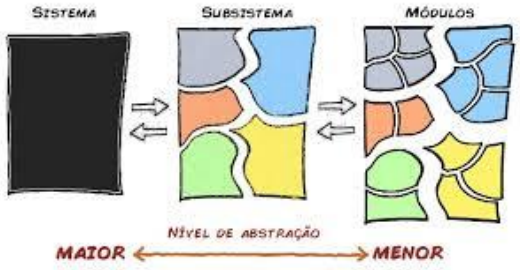
Evolução de um projeto de Software

- Projeto de software tem evoluído nas últimas **seis** décadas;
- Trabalhos iniciais concentravam-se na criação de programas **modulares** e de métodos para refinamento das estruturas de software de forma topdown;
- Aspectos procedurais da definição de um projeto evoluíram para uma filosofia denominada **Programação Estruturada**;
- Abordagens de projeto mais recentes, propuseram uma abordagem **orientada a objetos** para a derivação do projeto;

Conceitos de Projeto de Software

- Abstração
- Arquitetura
- Padrões
- Modularidade
- Encapsulamento de Informações
- Independência Funcional

Abstração



- Ao se considerar uma solução modular para qualquer problema, muitos níveis de **abstração** pode se apresentar;
- No nível de **abstração** mais alto, uma solução é expressa em termos abrangentes usando-se uma linguagem do domínio do problema;
- Em níveis de **abstração** mais baixos, uma descrição detalhada da solução é fornecida;
- Por fim, no nível de **abstração** mais baixo, a solução técnica do software é expressa de maneira que possa ser diretamente implementada.

```
Algoritmo Exemplo
     var x : numerico
Inicio
     Escreva "Escrevendo divisíveis por 2"
     x <- 0
     Enquanto x < 10 faça
          Se x%2 = 0
               Então
                    Escreva x
               Senão x <- x + 1
          Fim_Se
     Fim_Enquanto
Fim_Algoritmo
```

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s"' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Arquitetura

- Refere-se à **organização geral do software** e os modos pelas quais disponibiliza integridade conceitual para um sistema;

- Em sua forma mais simples, a arquitetura é a **estrutura ou a organização de componentes de programa** (módulos), a maneira através da qual esses componentes interagem e a estrutura de dados empregada pelos componentes;

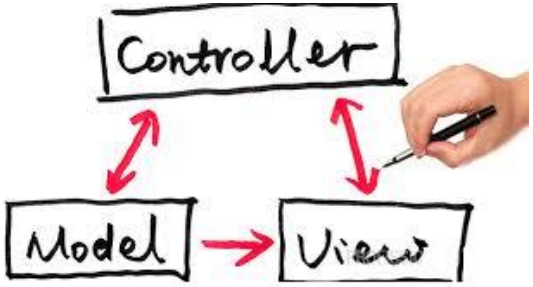
- Um conjunto de **padrões** de arquitetura permite a um Engenheiro de Software **reusar** soluções-padrão para problemas similares;

- O projeto de Arquitetura do Software inclui modelos **estruturais** (arquitetura como um conjunto organizado de componentes de programas), modelos **dinâmicos** (comportamento dos programas) e modelos **funcionais** (hierarquia funcional de um sistema).

Padrões de Projeto

- Um **padrão de projeto** descreve uma estrutura de projeto que resolve uma particular categoria de problemas de projeto;

- Os padrões de projeto fornecem um mecanismo codificado para resolver problemas e suas soluções de modo que permite à comunidade de Engenharia de Software **capturar conhecimento** de projeto para que possa ser **reutilizado**.



Modularidade

- O princípio "**divide-and-conquer"** estabelece que é mais fácil resolver um problema complexo quando o dividimos em partes gerenciáveis;
- Isso tem implicações importantes em relação à **modularidade** do software;
- Com a modularidade, o software é dividido em componentes especificados e endereçáveis, algumas vezes denominados **módulos**, que são integrados para satisfazer os requisitos de um problema;
- Acredita-se que "**modularidade**" é o único atributo de software que possibilita que um programa seja intelectualmente **gerenciável**".

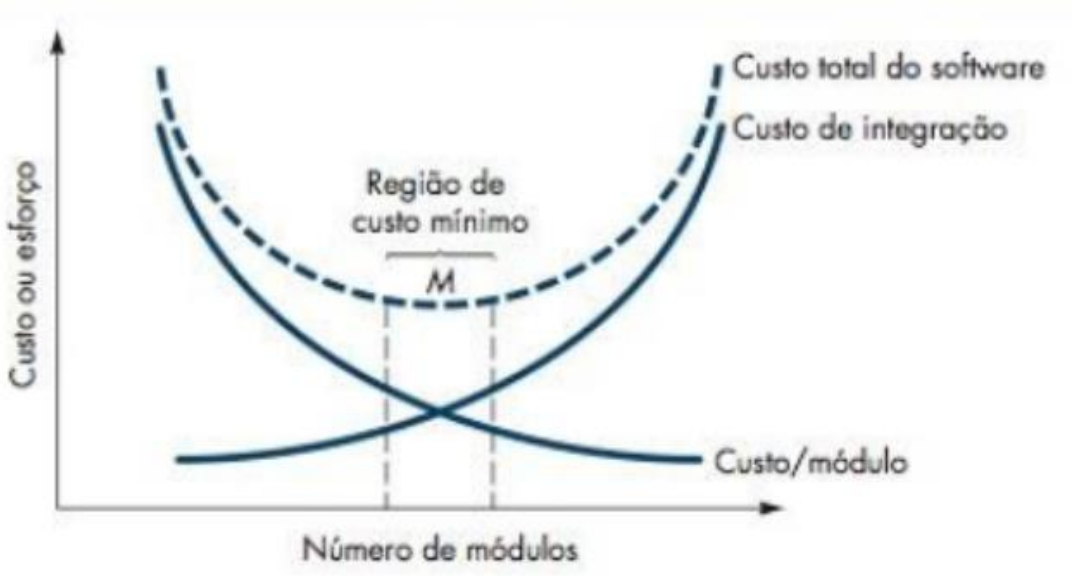
Modularidade – Considerações

- **Software monolítico** (composto de um único módulo) **não** pode ser facilmente compreendido por um Engenheiro de Software;
- O **número de caminhos de controle**, **variáveis** e **complexidade** geral tornaria o entendimento próximo do impossível;
- Assim, em quase todos os casos, **deve-se dividir o projeto em vários módulos**, para **facilitar** a compreensão e, consequentemente, **reduzir os custos** necessários para a construção do software.

Modularidade – Observações

- Infelizmente, quando se subdivide o software indefinidamente, à medida em que o número de módulos aumenta, o **esforço** (**custo**) associado à integração dos módulos também cresce;

- Deve-se modularizar, mas tomar cuidado para permanecer nas vizinhanças de **M**. Os conceitos vistos neste capítulo auxiliarão a determinar as vizinhanças de **M**.



Fonte: Pressman

Independência Funcional

- Resultado direto da aplicação dos conceitos de **modularidade**, **abstração** e **encapsulamento de informações**;
- Independência funcional é atingida desenvolvendo-se módulos com **função** "**única**" e com "**aversão**" à interação excessiva com outros módulos;
- Assim, deve-se projetar software de forma que **cada módulo atenda a um subconjunto específico de requisitos** e tenha uma **interface simples** quando vista de outras partes da estrutura do programa.

Importância da Independência Funcional

- ✓ Software com efetiva **modularidade**, isto é, módulos independentes é mais fácil de ser desenvolvido;
- ✓ Módulos **independentes** são mais fáceis de serem mantidos e testados, pois efeitos colaterais provocados por modificações no código são limitados, a propagação de erros é reduzida e módulos reutilizáveis são possíveis;
- ✓ **Independência funcional** é a chave para um bom projeto (chave para a qualidade de um software).

Independência Funcional – Avaliação

- ✓ A independência funcional é avaliada por meio de dois critérios qualitativos: **coesão** e **acoplamento**;

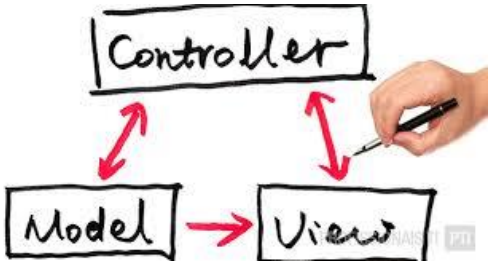
- ✓ **Coesão** é uma extensão natural do conceito de encapsulamento de informações. Um módulo coeso realiza uma única tarefa, exigindo pouca interação com outros componentes em outras partes de um programa. De forma simples, um módulo coeso deve (idealmente) fazer apenas uma coisa. Um módulo com alta **coesão** indica que este módulo possui uma funcionalidade ou responsabilidade bem definida no sistema, o que facilita a manutenção e reutilização;

- ✓ Módulos "esquizofrênicos" (módulos que realizam muitas funções não relacionadas) devem ser evitados caso se queira um bom projeto;

- ✓ O **acoplamento** é uma indicação da interconexão entre os módulos em uma estrutura de software e depende da complexidade da interface entre os módulos. Módulos com **acoplamento** baixo entre si indicam que a interdependência é fraca, o que diminui o risco de que uma falha em um módulo afete outro módulo no sistema;

- ✓ Em um bom projeto de software, deve-se esforçar para se obter o menor grau de **acoplamento** possível.

Arquitetura de Software



Introdução

- A Arquitetura de Software é composta por alguns **princípios**;
- Embora haja ferramentas que auxiliem na concepção da arquitetura, sem o emprego desses **princípios** não se desenhará com **qualidade** a arquitetura do software;

O que se pode pensar dessa imagem?

O que se pode pensar dessa imagem?

- Quarto desorganizado!
- Quarto desarrumado!
- Muitos objetos espalhados pelo quarto!

Algumas questões

- Você acharia alguma roupa no quarto ?
- Você acharia facilmente algum objeto no quarto?
- É fácil mudar o layout do quarto?
- Você se sentiria bem nesse quarto?

Uma outra visão

- Quarto organizado;
- Objetos em seus lugares corretos;
- Ambiente confortável.

Qual a diferença dos dois ambientes?

Caos x Ambiente Perfeito !

Ambiente Organizado

- Você acharia alguma roupa no quarto ?
- Você acharia facilmente algum objeto no quarto?
- É fácil mudar o layout do quarto?
- Você se sentiria bem nesse quarto?

Outra questão

- Se num ambiente físico tão simples (um quarto) pode-se facilmente se instalar um **caos**, imagine como seria num ambiente de desenvolvimento de software?

No desenvolvimento de software, essa situação pode ocorrer (caos), porém as consequências são muito mais drásticas ! ! !

Será que é fácil efetuar manutenção em softwares desorganizados?

Que problemas usualmente se encontra ao se fazer manutenção em um sistema desorganizado?

Software desorganizado
Observações frequentes

- Dificuldade para se **localizar** código;
- Partes internas difíceis de serem **alteradas**;
- Alterando-se alguma parte, **quebram-se** outras;
- Desenvolvedor inseguro e com **estresse**;
- Atribui-se falta de **qualidade** ao sistema de software;
- Atribui-se **incompetência** do desenvolvedor;
- Cliente passa a ter **desconfiança** da equipe de software;
- **Credibilidade** do desenvolvedor é afetada;
- **Custos** podem afetar o projeto;
- **Prazos** podem afetar o projeto;
- Mudanças de negócio não absorvidas pelo software;
- Há probabilidade de se **redesenvolver** o software.

Como desenvolver softwares organizados?

Diretriz Básica

- "**A única constante de um software é a sua permanente mudança**"; (Martin Folwer)
- O software deve ser organizado, estruturado, para **absorver** **mudanças**;
- Software não deve ser desenhado para permanecer **estático** ao longo do tempo;
- Desenvolver experiente emprega essa diretriz no desenvolvimento de software;
- **Software foi feito para ser alterado**;
- Software deve fornecer valor agregado ao cliente para o negócio atual e deve se **adequar** às futuras mudanças tecnológicas e corporativas.

Software para absorver mudanças

- Mudança de paradigma: Focar em **Qualidade** de Software;
- A Engenharia de Software trata a Qualidade focando conceitos de **Organização**, **Práticas** e **Diretrizes** de desenvolvimento de software;
- Esses conceitos são tratados na área de **Arquitetura de Software**;

O que é Arquitetura de Software?

Arquitetura de Software

- Corresponde à **estrutura** que abrange os **componentes** do software, as propriedades externamente visíveis desses componentes e as relações entre eles;
- Um **componente** pode ser um módulo de programa, uma classe orientada a objetos ou pode ainda ser estendido para abranger um banco de dados, etc;
- As **propriedades** dos componentes são características necessárias para o entendimento de como eles interagem com outros componentes;
- As **relações** podem ser, por exemplo, uma chamada de módulo, uma conexão a um banco de dados, etc.

Estilos de Arquitetura

- Arquitetos da Construção Civil usam diversos estilos arquitetônicos para a construção de um casa;
- Por exemplo, estilo Colonial, Estilo Contemporâneo, Estilo Clássico, etc

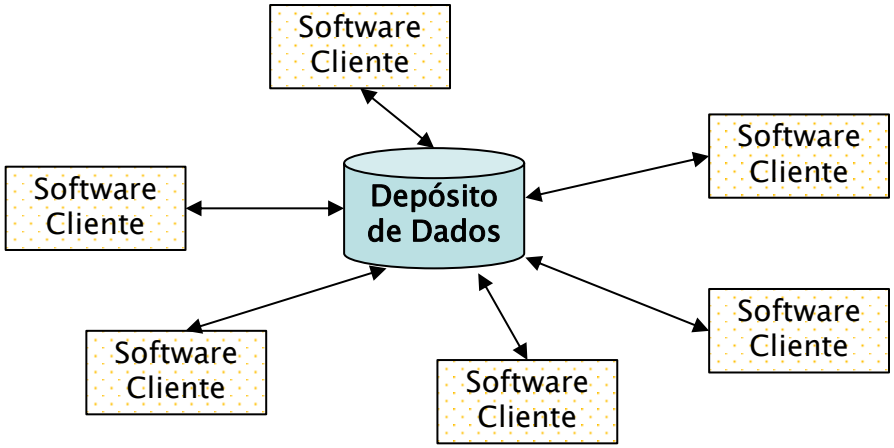
Estilos de Arquitetura de Software

- Arquiteturas Centralizadas em Dados;
- Arquiteturas de Fluxo de Dados;
- Arquitetura de Chamadas e Retornos;
- Arquiteturas Orientadas a Objetos;
- Arquitetura em Camadas;
- Arquitetura Orientada a Serviços.

Arquiteturas Centralizadas em Dados

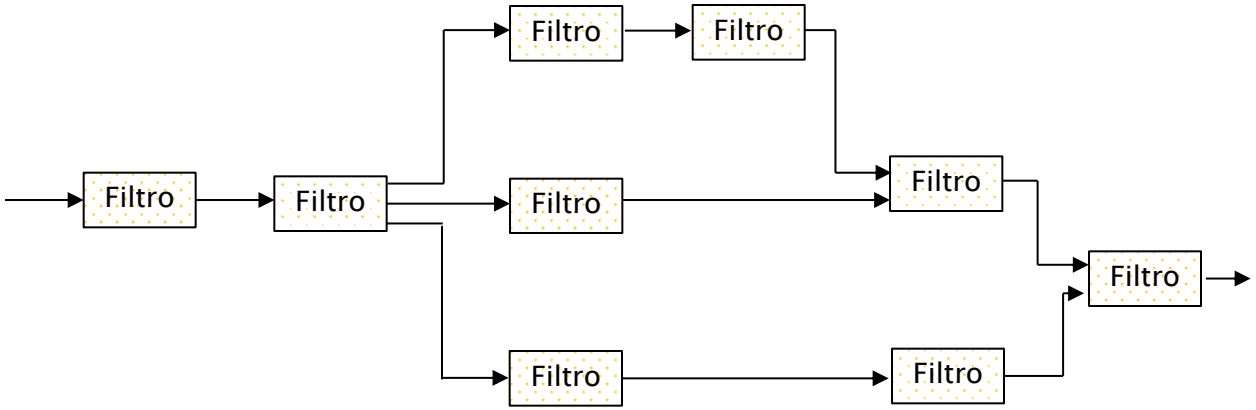
- Um **repositório** de dados (banco de dados) reside no centro dessa arquitetura e é em geral acessado por outros componentes que atualizam, acrescentam, eliminam ou modificam os dados contidos nesse repositório;

- O software-cliente acessa um repositório central e executam processos de forma **independente**.



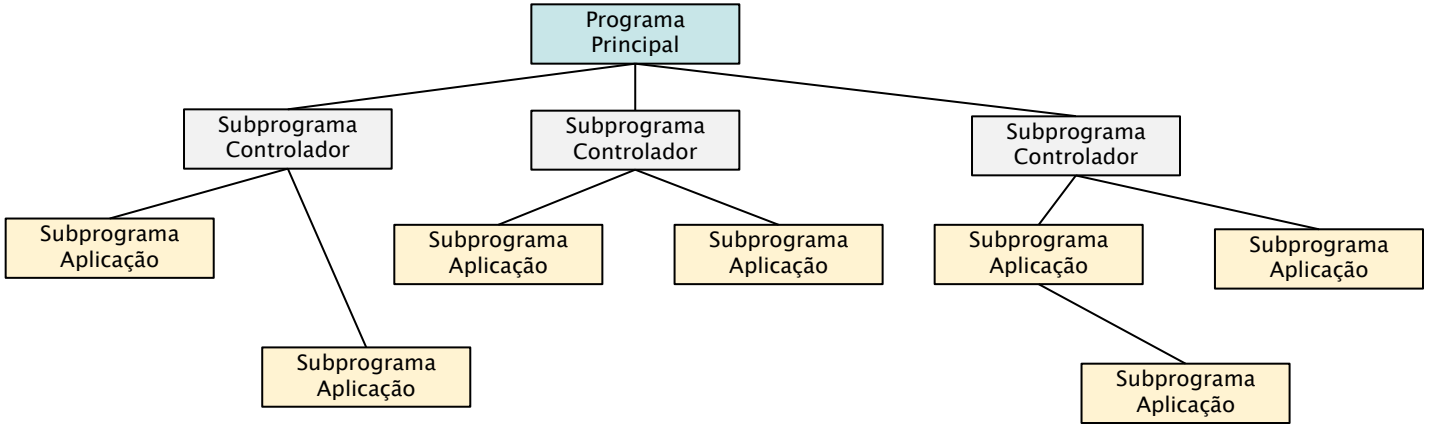
Arquiteturas de Fluxo de Dados

- Estilo no qual dados de entrada devem ser **transformados** por meio de uma série de componentes computacionais ou de manipulação de dados de saída;
- Exemplo: Processo **batch** de compilação de linguagens de programação.



Arquiteturas de Chamadas e Retornos

- Estrutura clássica de programação no qual decompõe-se a função em uma **hierarquia** de **controle** e um "**programa principal**" invoca uma série de componentes de programa que, por sua vez, podem invocar outros. A chamada pode ser feita por procedimentos remotos em um ambiente distribuído.



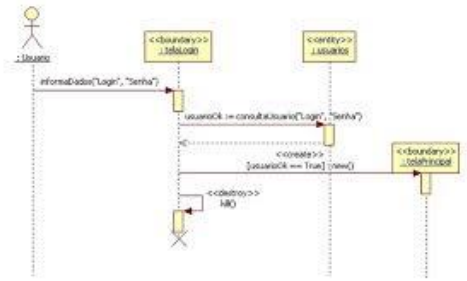
Arquiteturas Orientadas a Objeto

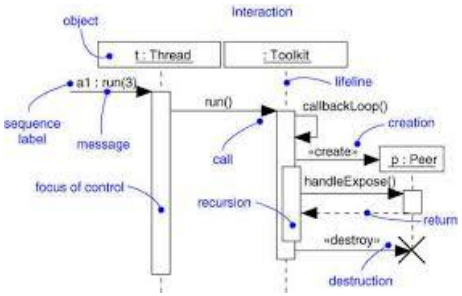
- Essa estrutura combina dados com funções numa única entidade (**objeto**), facilitando a decomposição do problema, manutenção e reuso;

- É comum utilizar a arquitetura orientada a objetos em sistemas de informação como sistemas de consulta e empréstimos online de bibliotecas de instituições de ensino que dispõem de componentes de cadastro de usuários e componentes de autenticação de usuários.

- Note que **componentes similares** existem em outros sistemas de informações, tais como sites de conteúdos, que exigem cadastro e autenticação de qualquer usuário antes de disponibilizar o conteúdo;

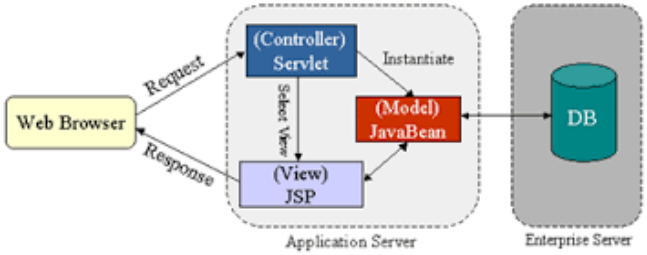
- Os componentes encapsulam dados e as operações que devem ser aplicadas para manipular dados. A comunicação e a coordenação entre componentes é feita por meio de passagem de **mensagens**.



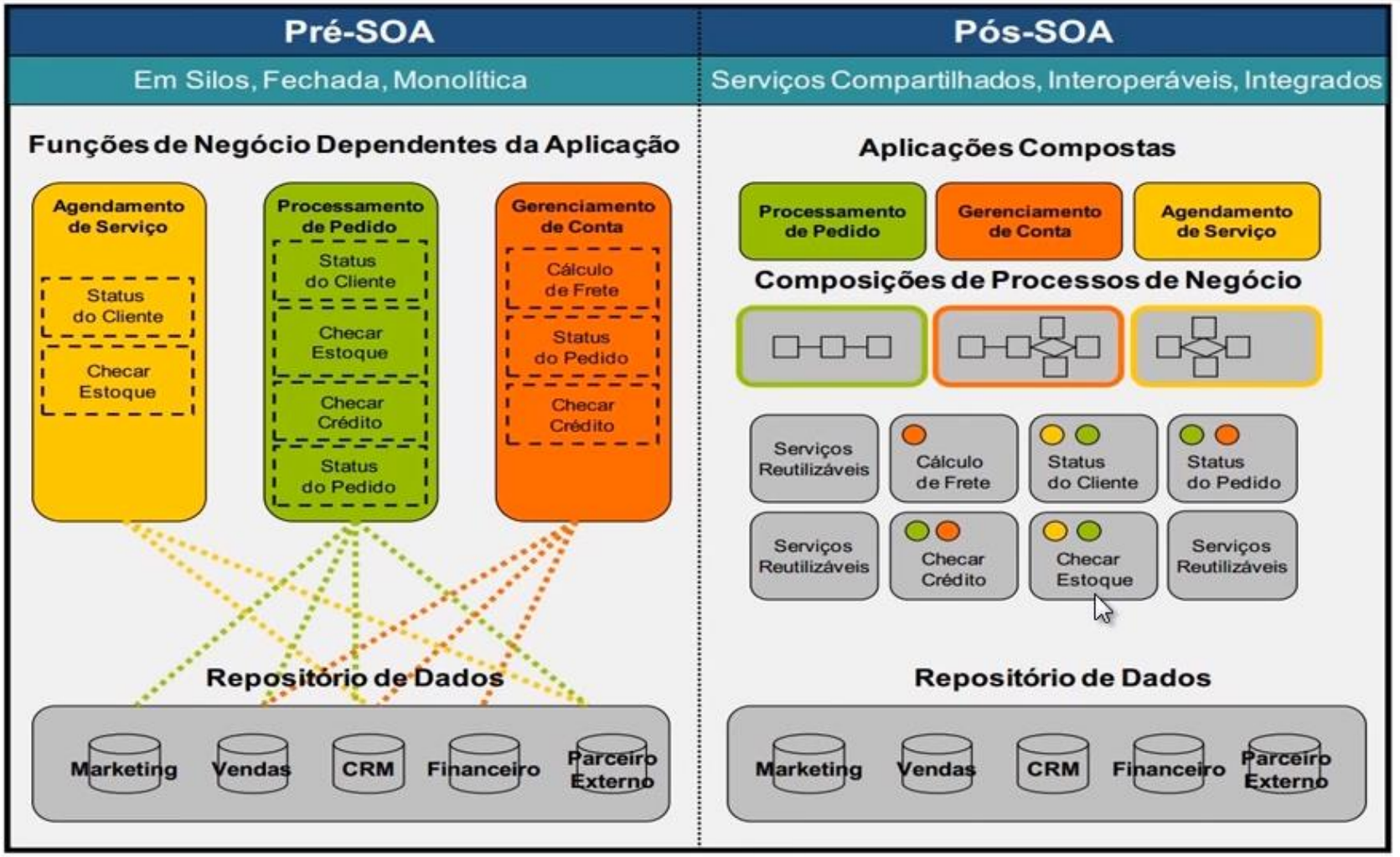


Arquitetura em Camadas

- A arquitetura do sistema de software é organizada num **conjunto de camadas**, oferecendo maior flexibilidade e suporte a portabilidade.
- A identificação do nível de abstração nem sempre é evidente e perde-se desempenho à medida que o número de camadas cresce.
- Exemplo desse estilo compreende os sistemas **Web** de múltiplas camadas que separa cliente, servidores de aplicação, servidores Web e outros clientes Web.

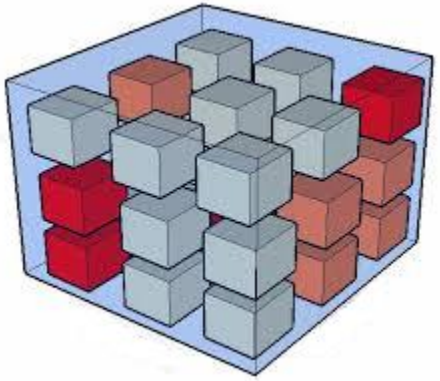


Arquitetura Orientada a Serviços



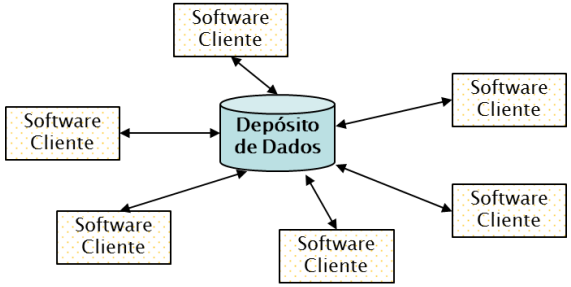
Projeto da Arquitetura de Softwawe

- À medida em que o **modelo de requisitos** é desenvolvido, pode-se perceber que o software deve tratar de **problemas** mais amplos que envolvem toda a aplicação;
- Na prática, várias soluções computacionais podem ser definidas para atender ao modelo de requisitos do software;
- A atividade de arquitetura de software contribui para representar possíveis soluções de software com o uso de **abstrações** (auxílio na visualização e entendimento de propriedades do software) e emprego cada vez maior de **frameworks** visando diminuir o esforço de construção do software.

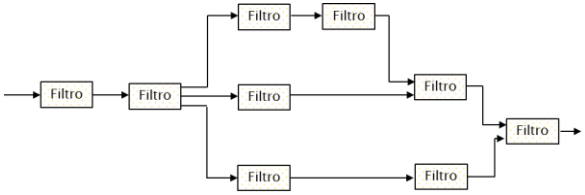


Mapeamento da Arquitetura com Fluxo de Dados

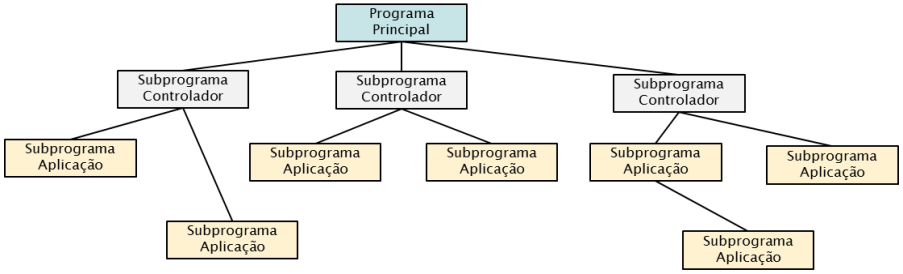
- O projetista de software tem a seu dispor diversos estilos arquitetônicos que são radicalmente diferentes, conforme visto no capítulo anterior;



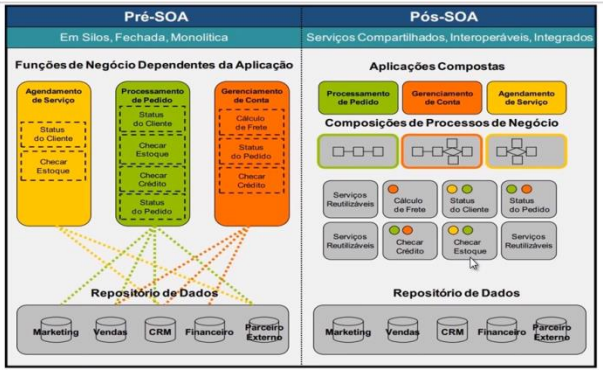
Arquitetura Centralizada em Dados



Arquitetura de Fluxo de Dados



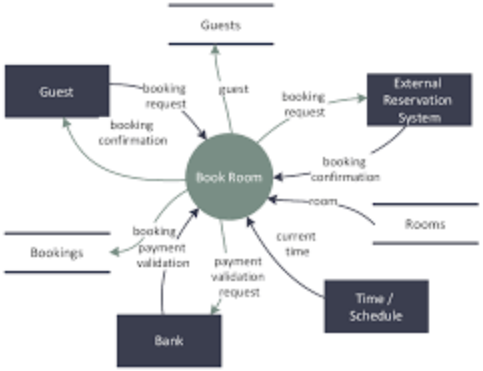
Arquitetura de Chamadas e Retornos

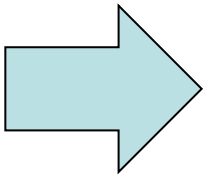


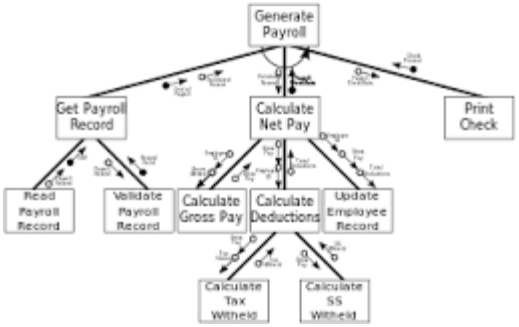
Arquitetura Orientada a Serviços

Mapeamento da Arquitetura com Fluxo de Dados

- Na verdade, **não** existe um mapeamento prático para alguns estilos de arquitetura;
- Uma estrutura extremamente comum para muitos sistemas é a **arquitetura de chamadas e retornos**;
- Uma técnica de mapeamento, chamada **Projeto Estruturado** (Yourdon,79) pode ser utilizada para derivar a arquitetura a partir do fluxo de dados.

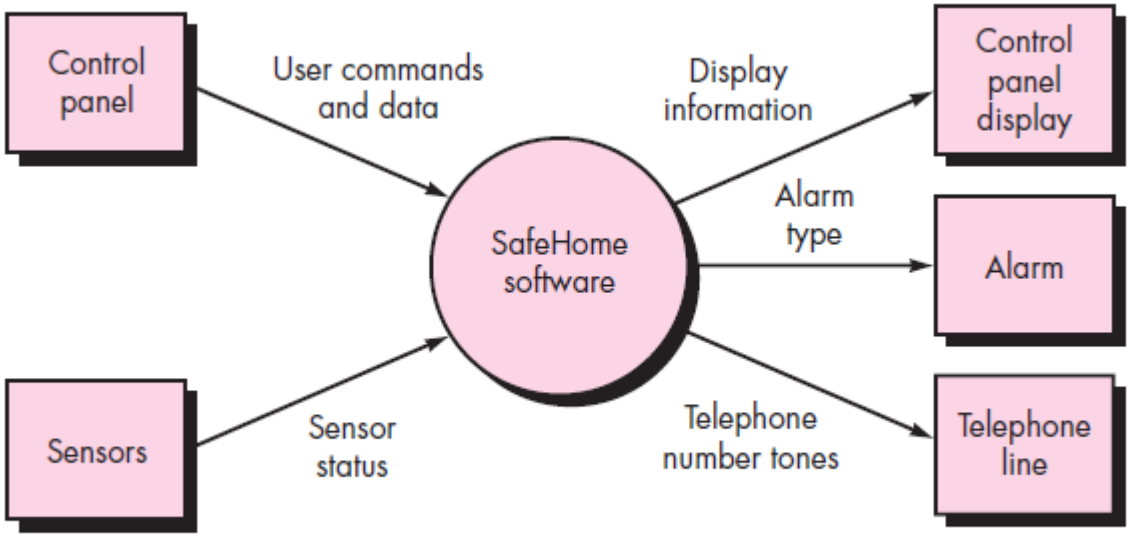






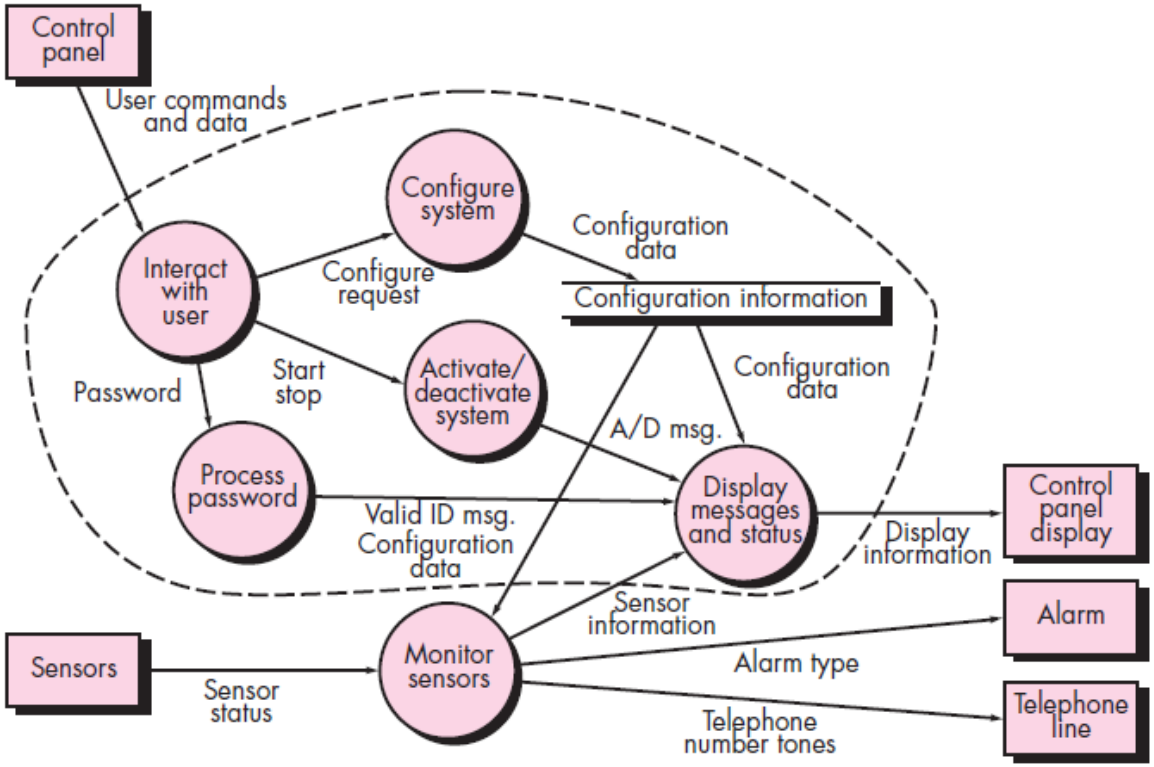
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 1**: Desenho do Diagrama de Contexto do Software – DFD de Nível 0



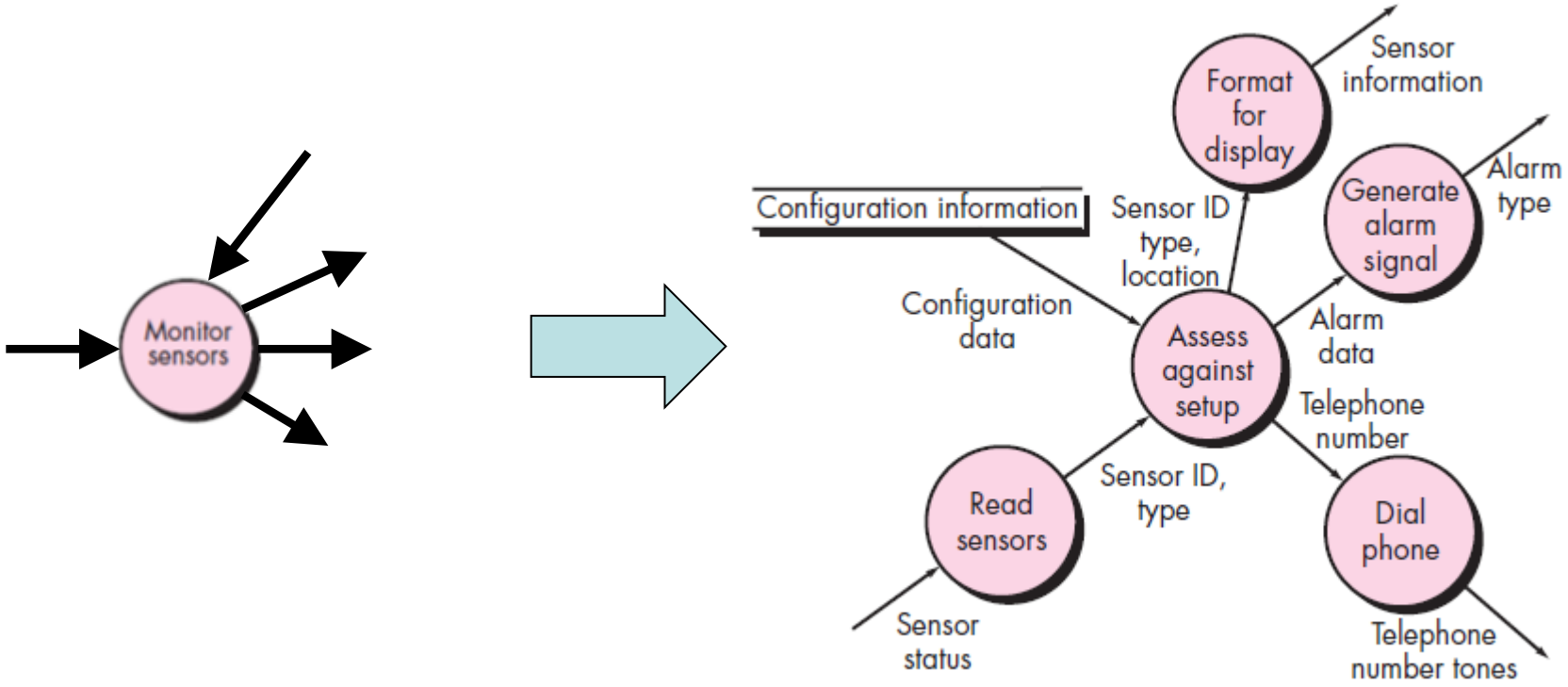
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 2**: Refinamento para obtenção do DFD de Nível 1



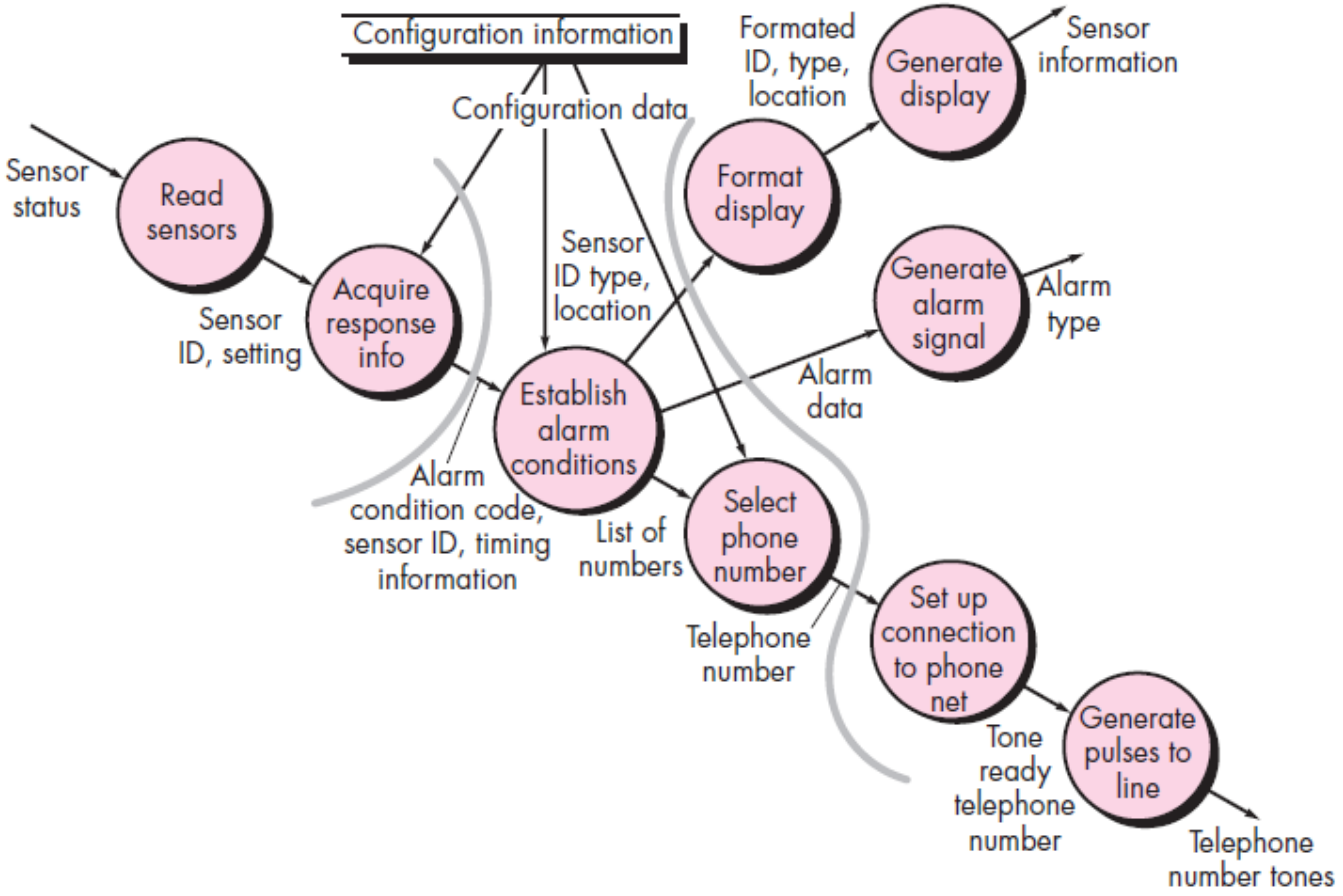
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 2**: Refinamento para obtenção do DFD de Nível 2



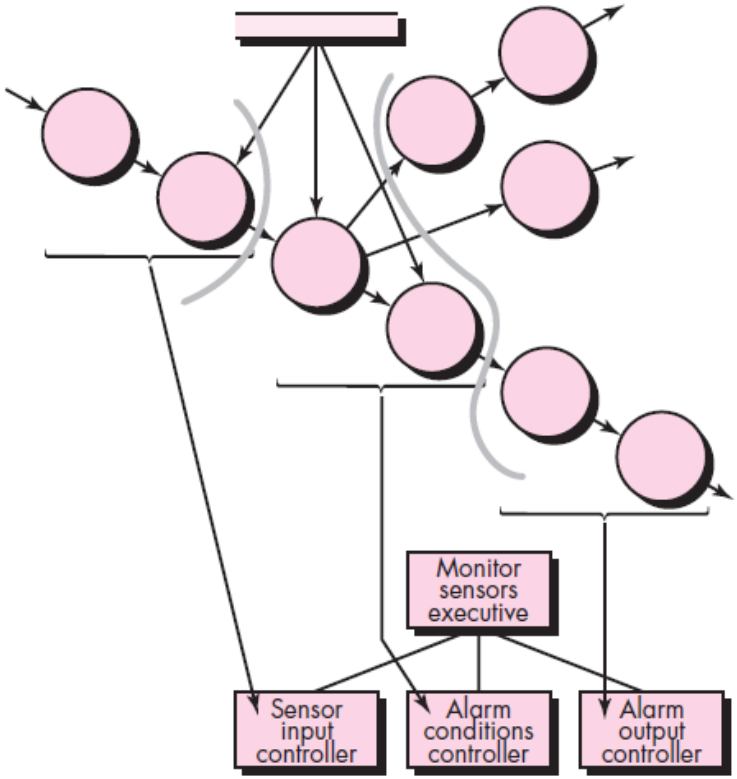
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 3**: Refinamento até obtenção de primitivas funcionais com alta coesão



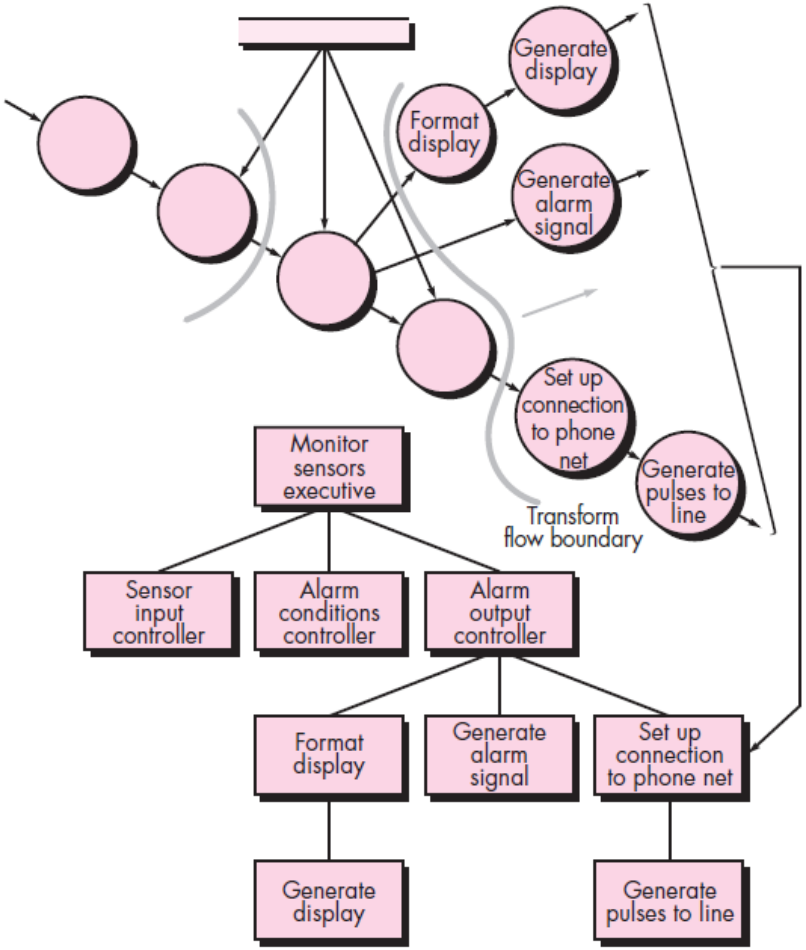
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 4**: Derivação da Estrutura



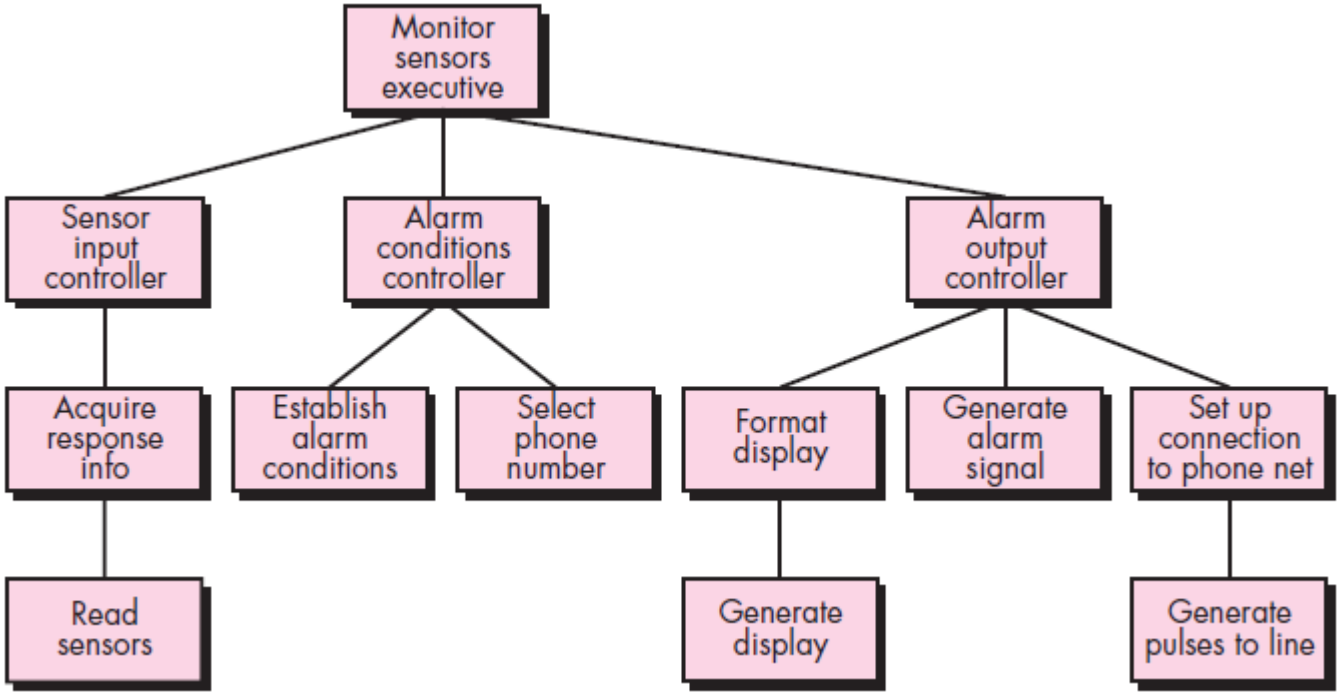
Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 5**: Derivação da Estrutura em outros níveis



Mapeamento da Arquitetura com Fluxo de Dados

- **Passo 6**: Obtenção do Diagrama de Estrutura



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.

Fonte: Pressman

Padrões de Projeto – Programação Orientada a Objetos

Princípios SOLID

Solid

- Padrões **Solid** são considerandos boas práticas de programação orientada a objetos, visando diminuir o acoplamento entre as classes e separar responsabilidades;

Princípio do aberto-fechado (OCP)

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

Fonte: **Pressman**

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)

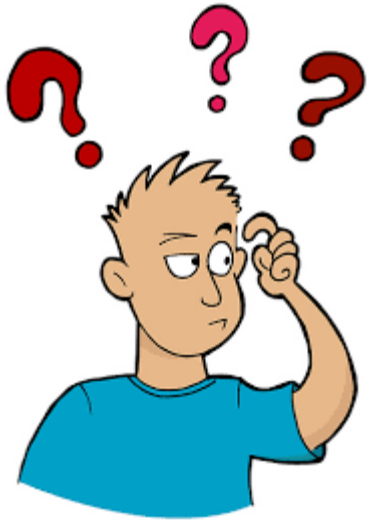
- **Extensibilidade**: É uma das chaves da orientação a objetos. Quando um novo comportamento ou funcionalidade precisar ser adicionada, espera-se que as existentes sejam **estendidas** e não **alteradas**, assim o código original permanece intacto e confiável enquanto as novas são implementadas através 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)?



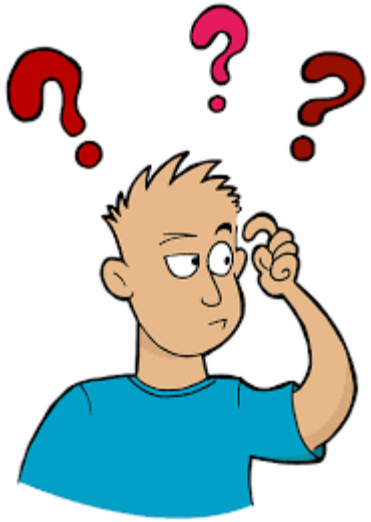
Fácil! Basta acrescentar um IF na classe !

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

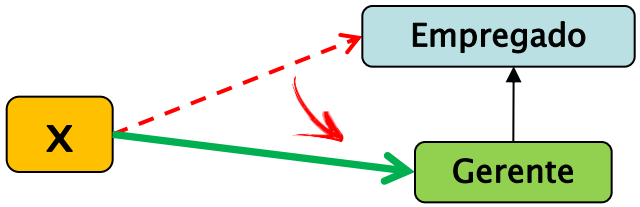
- O tipo de débito em conta de investimento foi implementado **sem modificar classes existentes**, usando-se apenas a extensão.
- Além disso, o código está mais **legível** e mais fácil para se aplicar **cobertura de testes de unidade**.
- Este princípio nos atenta para um melhor design, tornando o software 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.

Fonte: **Pressman**

Princípio de substituição de Liskov (LSP)

LSP – Princípio de Substituição de Liskov

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

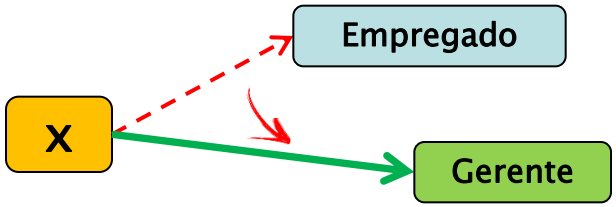


Princípio de substituição de Liskov (LSP)

- 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)

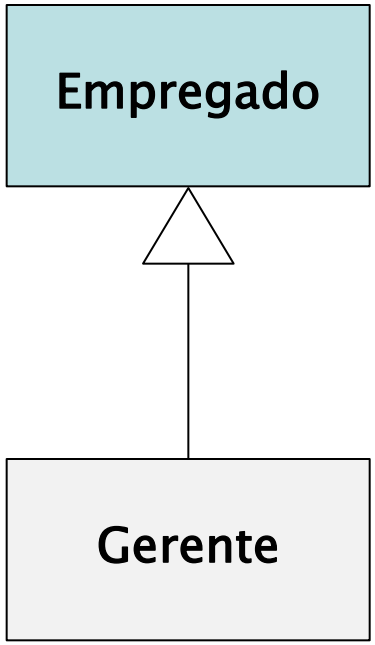
- Instâncias da **superclasse** podem ser substituídas por instâncias da **subclasse**;
- O princípio está fortemente relacionado ao conceito de Herança em Programação Orientada a Objetos;
- Herança permite que se obtenha reutilização de código;
- Porém, na prática, pode haver armadilhas criadas pela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



Fonte: **Pressman**

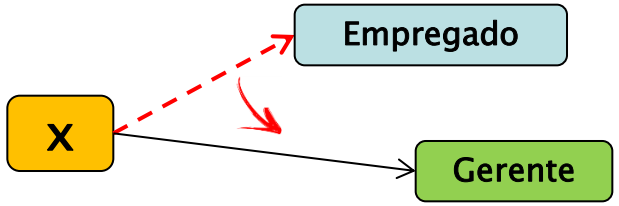
Conceitos de Herança

- 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:



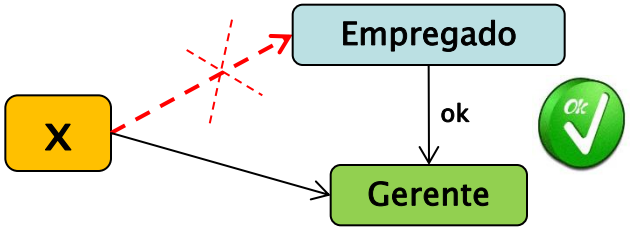

```
package maua;

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**).

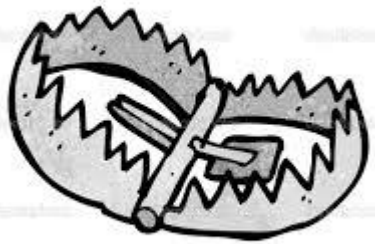
x é do tipo Empregado !

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();
}
```

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();
}
```