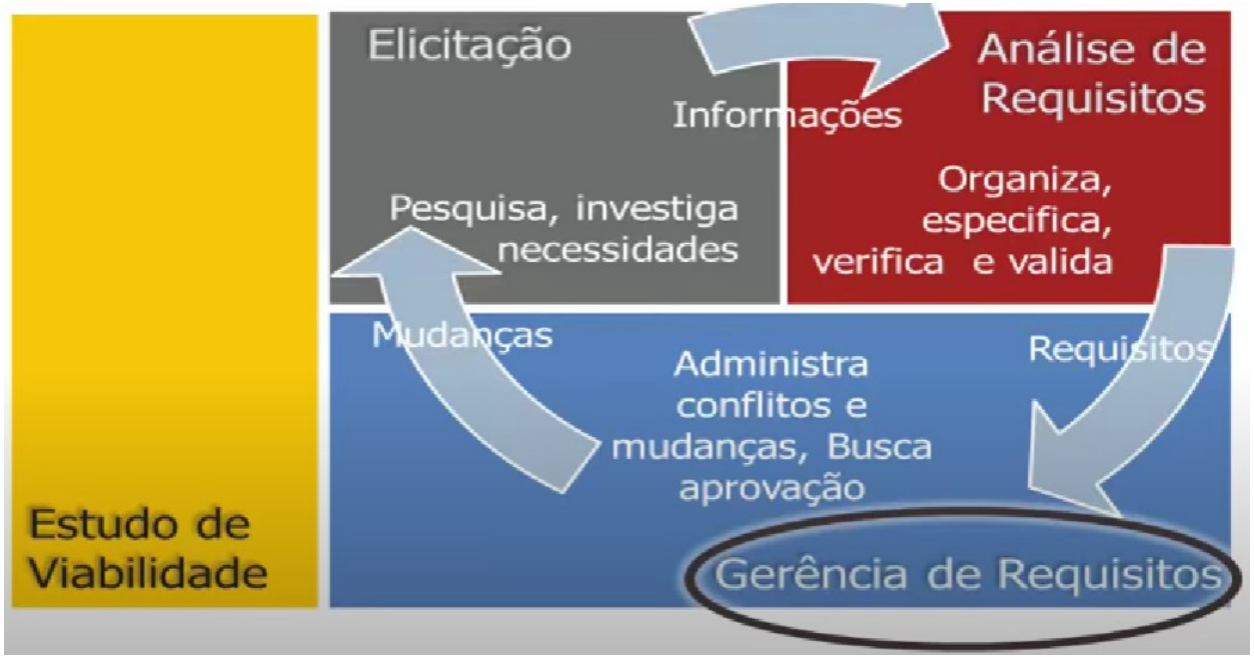
Modelos, Métodos e Técnicas de Engenharia de Software

Prof.: Sônia Aparecida Santana

una

Recordando...

- Tarefas da Engenharia de Requisitos

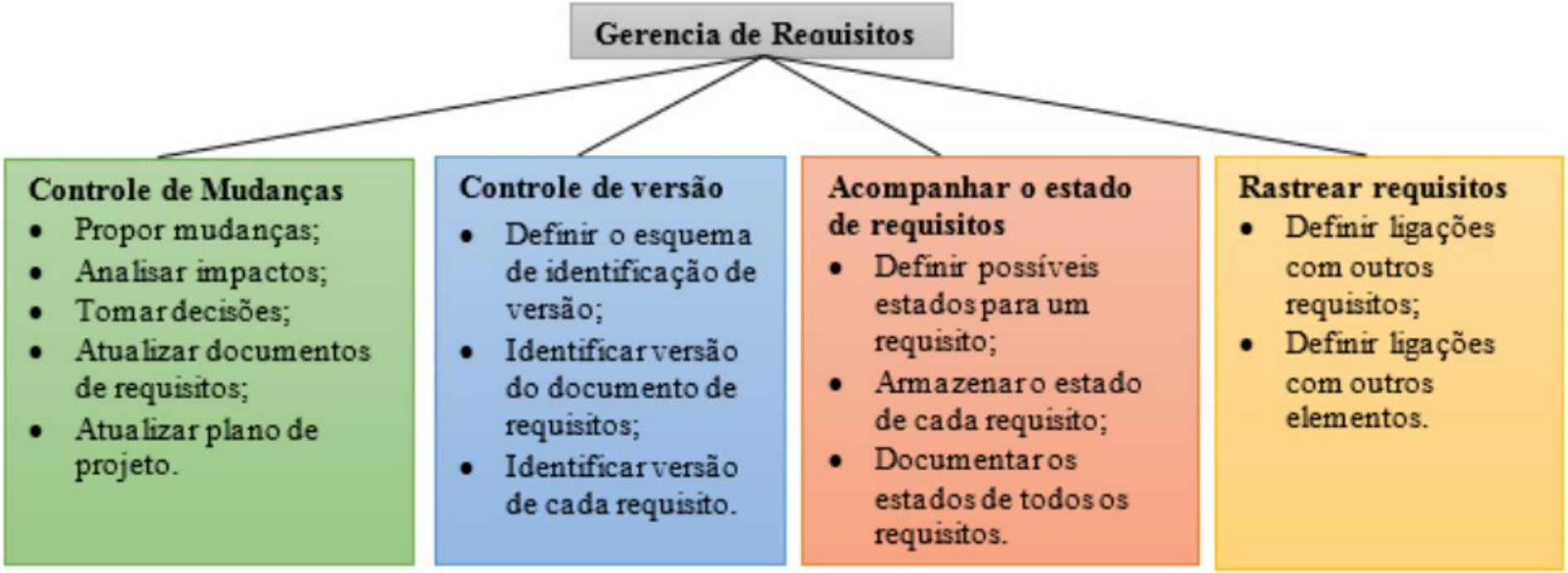


Gerenciamento de Requisitos

Determina o processo de gerenciamento de requisitos adequado. Em geral isto é feito pelos responsáveis pela metodologia e/ou processo de desenvolvimento de software da própria organização

- Processo de **controle mudança de requisitos**
- Quais **atributos dos requisitos** serão capturados
  - Referência, Autor, Complexidade, Proprietários, Prioridade, Risco, Origem, Estabilidade, Estado, Urgência e Outros
- Nível de **rastreabilidade dos requisitos**
- Estabelecer o processo de **priorização de requisitos**
- Repositório e ferramentas

Atividades de Gerência de Requisitos



Rastreabilidade

- Uma grande preocupação que deve ser levada em consideração quando fazemos **análise de requisitos**, é a necessidade de possuir elementos para podermos investigar toda a **trajetória** de um determinado requisito.
- Quando temos estes elementos registrados no sistema, como: origem, documentação, dependências, relacionamentos, grau de importância, histórico, sua aplicação nos modelos; caso o projeto e o sistema já tenham sido construídos, podemos ter condições de fazer uma boa **rastreabilidade** nos requisitos do sistema.

Rastreabilidade

Tipos de rastreabilidade (Sommerville, 2021):

- **Rastreabilidade de fontes de requisitos**: ligação entre os requisitos e as pessoas ou documentos que especificam os requisitos. Podemos citar como exemplo dessa rastreabilidade de requisitos, uma funcionalidade que emite nota fiscal estar vinculada com as normas da Receita Estadual.
- **Rastreabilidade lógica de requisitos**: ligação do requisito com a descrição de o porquê deste requisito ter sido especificado. Pode ser a essência da informação de diversas fontes. Podemos citar como exemplo dessa rastreabilidade de requisitos, uma funcionalidade que emite nota fiscal estar vinculada a diferentes legislações quando um produto é vendido dentro e fora do país.

Rastreabilidade

Tipos de rastreabilidade (Sommerville, 2021):

- **Rastreabilidade requisito-requisito**: ligação dos requisitos com outros requisitos que são dependentes entre eles. Pode ser vista de duas maneiras: dependentes (onde existe dependência de ambas as partes, ou seja, um requisito depende do outro e vice-versa) e depende de (quando há uma dependência unilateral, ou seja, um requisito é totalmente dependente do outro, mas a recíproca não é verdadeira). Podemos citar como exemplo da dependência existente entre dois casos de uso.
- **Rastreabilidade requisito-arquitetura**: ligação entre o requisito e os subsistemas onde esses requisitos são implementados. Esse quesito é importante onde o desenvolvimento é feito por uma organização. Podemos citar como exemplo, que uma determinada aplicação será utilizada ao mesmo tempo na web e por meio de dispositivos móveis.

Rastreabilidade

Tipos de rastreabilidade (Sommerville, 2021):

- **Rastreabilidade requisito-projeto**: ligação dos requisitos com os componentes de hardware ou software. Podemos citar como exemplo, uma funcionalidade que precisa realizar o envio de e-mail e deve ter à sua disposição uma conexão permanente com a internet.
- **Rastreabilidade interface-requisito**: ligação dos requisitos com as interfaces externas usadas pelo sistema. Podemos citar como exemplo, que uma determinada funcionalidade está sendo atendida por uma ou mais telas do sistema.

Rastreabilidade

Matriz de rastreabilidade (Sommerville, 2021):

- Uma forma de visualização do relacionamento entre os requisitos. Por meio dela, são listados os requisitos nas linhas e nas colunas da matriz, depois se aplica um asterisco (*) nas células correspondentes, caracterizando os relacionamentos entre os requisitos.

	Depende do				
	Requito A	Requito B	Requito C	Requito D	Requito E
Requito A			x		
Requito B				x	x
Requito C		x			
Requito D					x
Requito E	x				

Matriz de Rastreabilidade

- Casos de Uso e Requisitos Funcionais (exemplo)

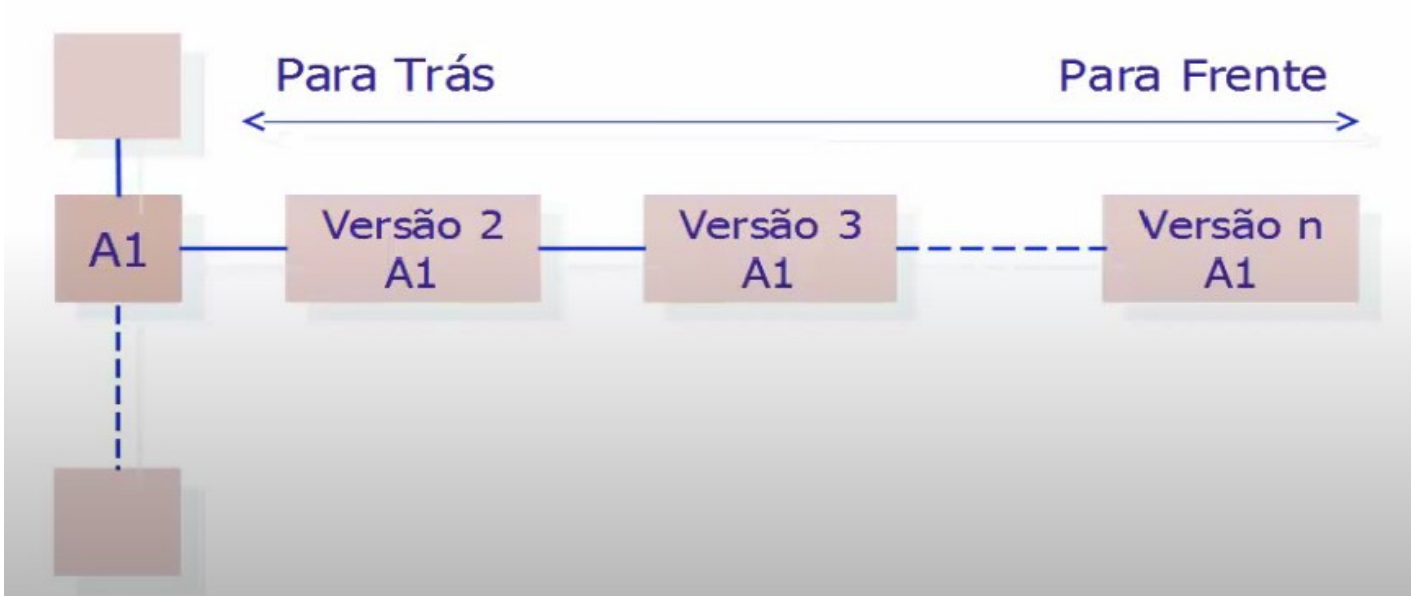
Requisitos X Casos de Uso	[UC-001] - Incluir Recebimento	[UC-002] - Alterar Recebimento	[UC-003] – Validar Contas Recebidas	[UC-004] – Conciliar Contas	[UC-005] - ...
[RF-001] – Recebimento de Conta	X	X			
[RF-001] - Transmissão de Contas Recebidas			X		
[RF-003] - Transmissão de Depósitos				X	
[RF-010] – Conciliação Fiscal Automática					
[RF-011] - ...					

Matriz de Rastreabilidade

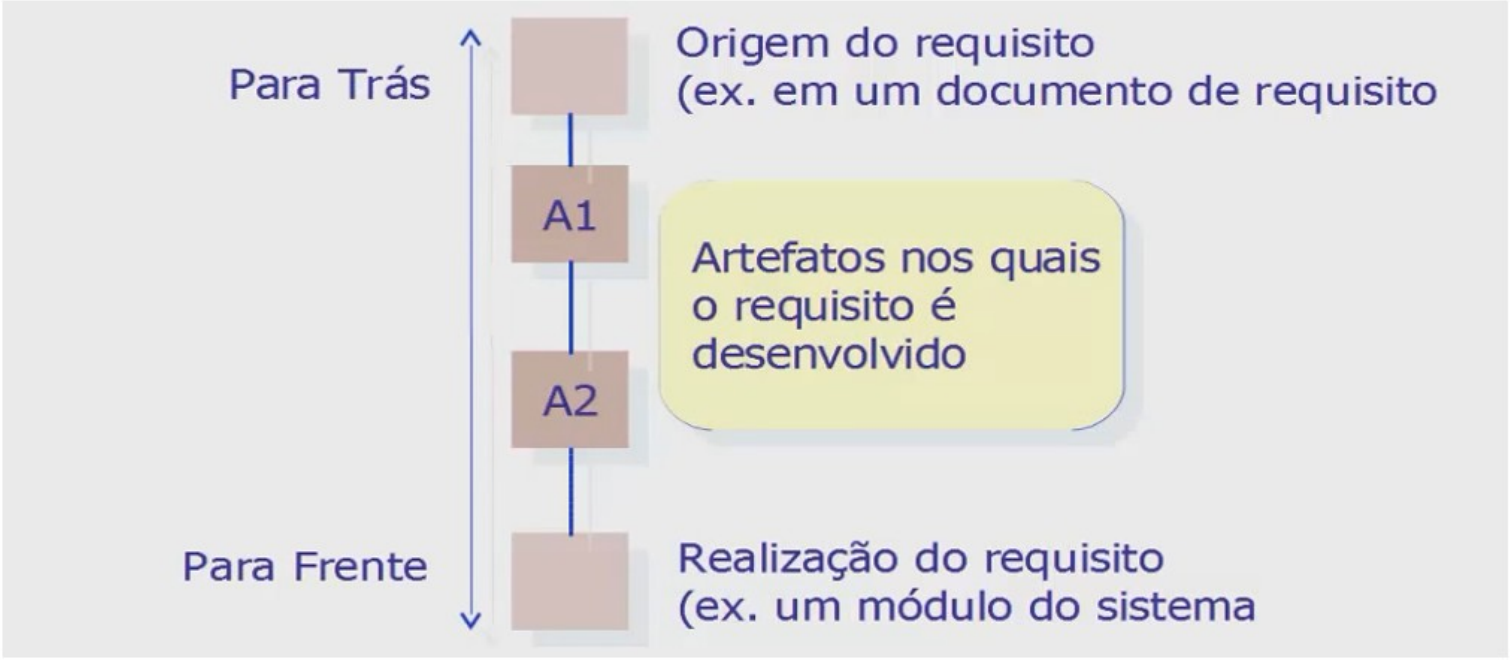
- Requisitos Funcionais e Requisitos Funcionais

Requisitos Funcionais X Requisitos Funcionais	[RF-001] – Recebimento de Conta	[RF-002] - Transmissão de Contas Recebidas	[RF-003] - Transmissão de Depósitos	[RF-004] - ...
[RF-001] – Recebimento de Conta	X	X	X	
[RF-002] - Transmissão de Contas Recebidas	X		X	
[RF-003] - Transmissão de Depósitos	X	X	X	
[RF-004] - ...				

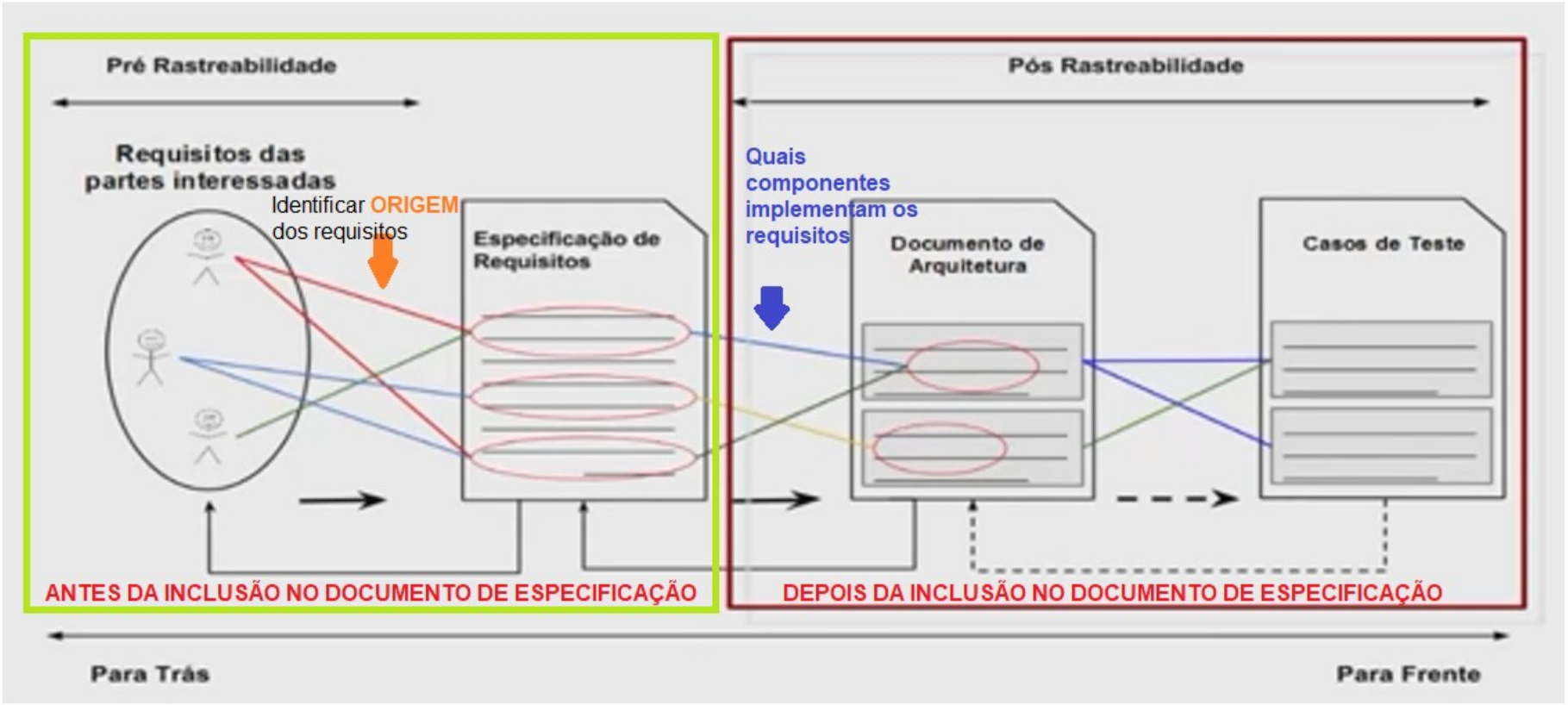
Rastreabilidade Horizontal



Rastreabilidade Vertical



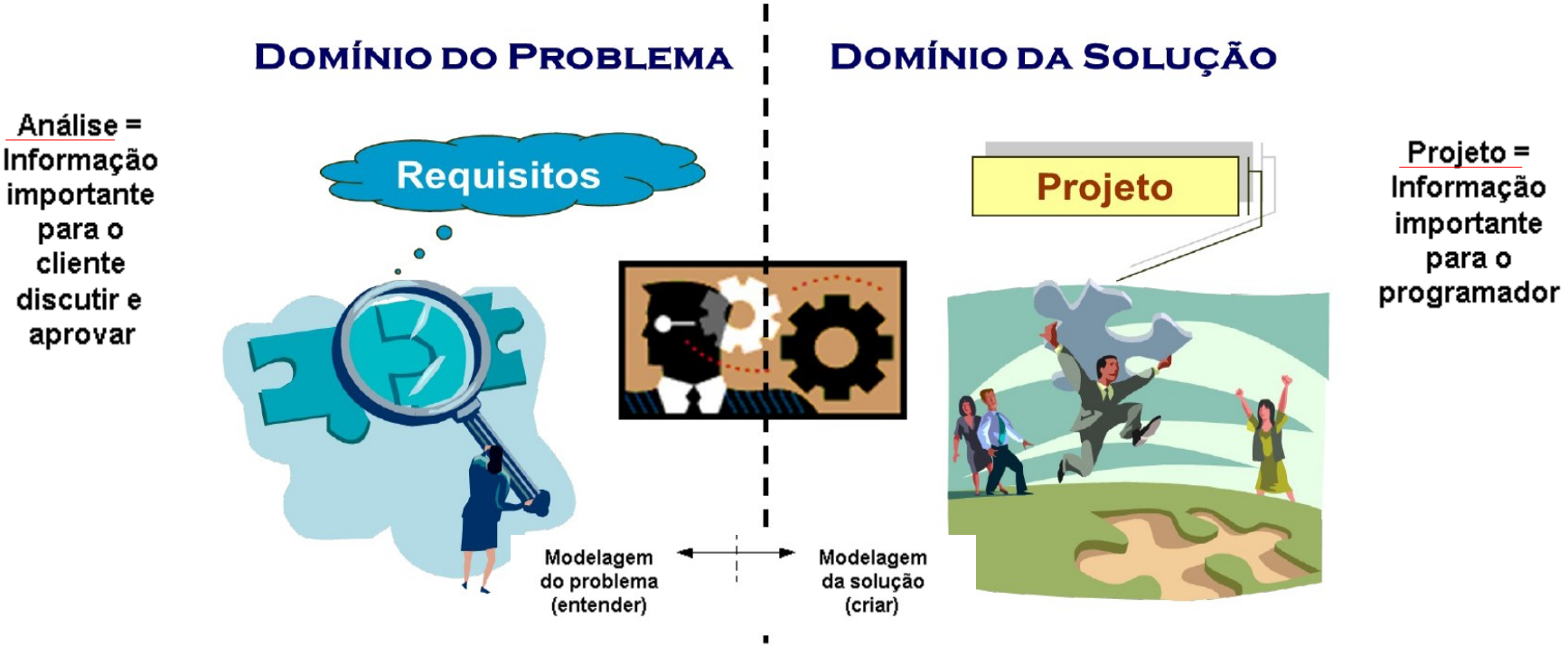
Pré e Pós Rastreabilidade



Projeto de Software

Princípios de Projeto

Introdução

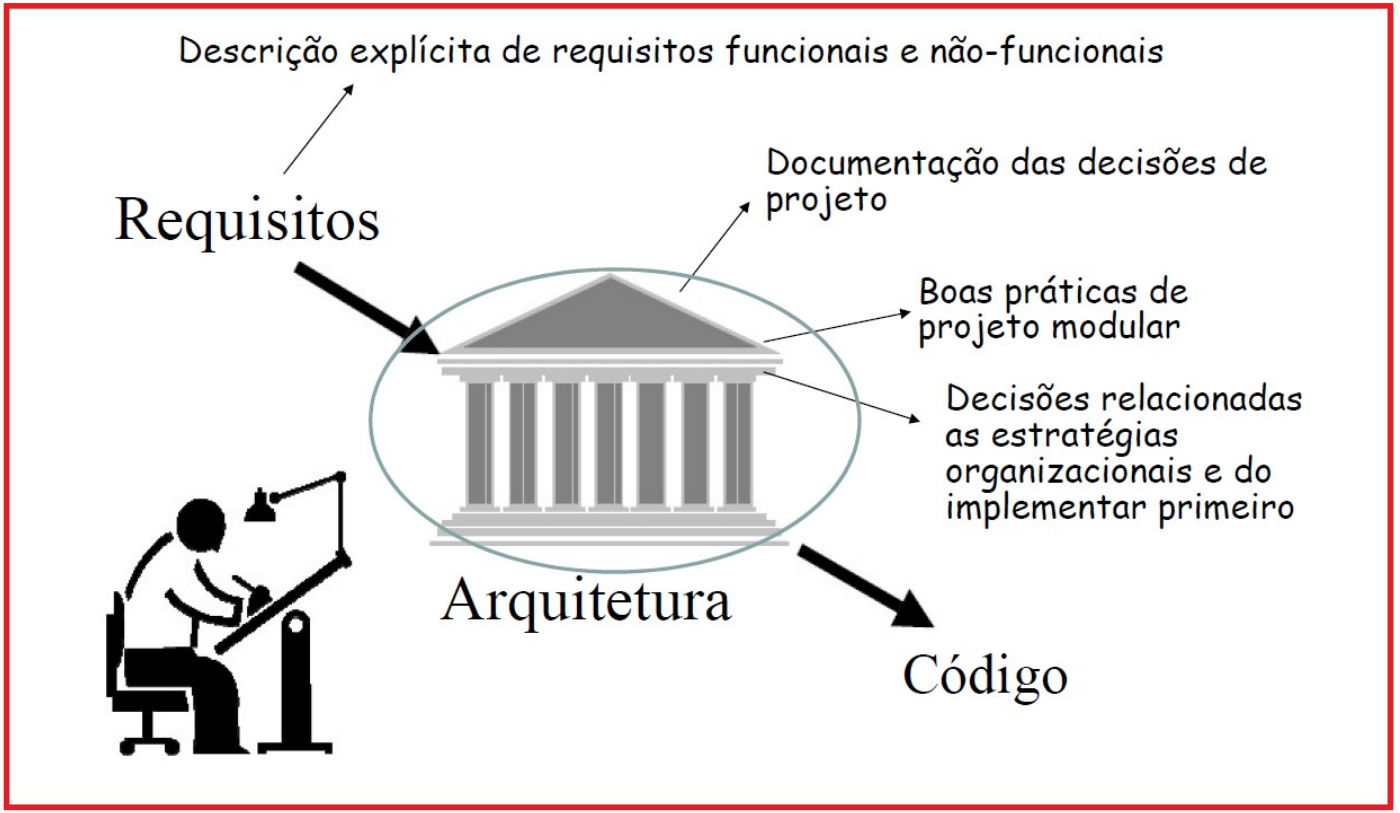


Fonte: Aula de Design Pattern, Frameworks e Componentes ©2014 - Prof. Dr. Jean Carlo Rossa Hauck (adaptado)

Projeto vs Design

- Em Inglês, temos duas palavras:
  - Project: esforço colaborativo para resolver problemas
  - Design: desenho ou proposta de uma solução
- Em Português, temos uma única palavra: projeto
- Neste módulo, **projeto = design**

Visão


Descrição explícita de requisitos funcionais e não-funcionais
Requisitos
Documentação das decisões de projeto
Boas práticas de projeto modular
Decisões relacionadas as estratégias organizacionais e do implementar primeiro
Arquitetura
Código

Abordagens de Construção

- **Desenvolvido**
  - Desenvolver uma aplicação personalizada
- **Comprados e personalizados**
  - Comprar um sistema pronto e personalizá-lo
- **Terceirizado**
  - Contar com um fornecedor, desenvolvedor ou provedor de serviços externo para construir o sistema.

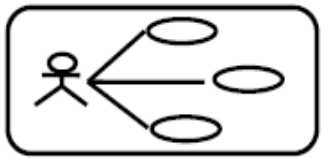
Abordagens de Construção

	Desenvolver	Comprar	Terceirizar
Necessidade Operacional	A necessidade operacional é exclusiva	A necessidade operacional é comum	A necessidade operacional não é essencial para a empresa
Experiência interna	Existe experiência funcional e técnica na casa	Existe experiência funcional interna	Não existe experiência funcional e técnica internamente
Qualificações do projeto	Há um desejo de desenvolver qualificações internas	As habilidades não são estratégicas	A decisão de terceirizar é uma decisão estratégica
Gerenciamento do projeto	O projeto tem um gerente de projeto altamente qualificado e uma metodologia comprovada	O projeto tem um gerente de projeto que pode coordenar os esforços do fornecedor	O projeto tem um gerente de projeto altamente qualificado no nível da empresa que corresponde ao escopo de acordo com a tercerização
Cronograma	O cronograma é flexível	O cronograma é curto	O cronograma é contínuo e flexível

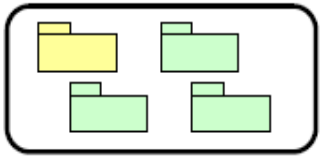
Projeto de Software

- ***Como*** satisfazer as necessidades especificadas no documento de requisitos?
  - arquitetura do sistema
  - estruturas de dados, diagramas, algoritmos
  - projeto de módulos
- Um projeto geralmente abrange níveis.
  - **projeto alto-nível** (projeto do sistema):
    - decidir quais módulos são requeridos
    - especificação dos módulos
    - interconexão dos módulos
  - **projeto baixo-nível** (projeto detalhado):
    - projeto interno dos módulos

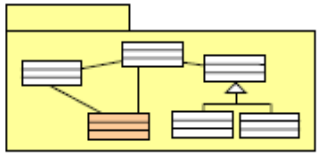
Níveis de Projeto



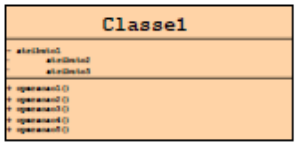
1 - Sistema de Software



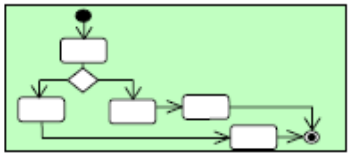
2 – Divisão em subsistemas/ pacotes



3 – Divisão em classes dentro dos pacotes



4 – Divisão em dados e rotinas dentro das classes



5 – Projeto interno de rotina

Níveis de Projeto

- Nível 1 – Sistema de Software
  - O primeiro nível é o sistema inteiro. Alguns desenvolvedores pulamdiretamente do nível de sistema para o projeto de classes, mas normalmente é vantajoso cogitar a respeito de combinações de mais alto nível entre as classes, como os subsistemas ou pacotes.

Níveis de Projeto

- Nível 2 – Divisão em subsistemas ou pacotes
  - O principal produto do projeto de software neste nível é a identificação de todos os subsistemas importantes. Os subsistemas podem ser grandes: banco de dados, interface com o usuário, regras de negócio, interpretador de comandos, mecanismos de relatório e etc..
  - Mostrar como cada subsistema utilizará os outros.

Níveis de Projeto

- Nível 3 – Divisão em classes
  - Identificar todas as classes do sistema.
  - Detalhes das maneiras pelas quais cada classe interage com o restante do sistema também são especificados à medida que as classes são especificadas.(Interface)
  - Nesse nível é necessário certificar de que todos os subsistemas tenha sido decompostos em um nível de detalhe suficiente para que se possa implementar suas partes como classes individuais.

Níveis de Projeto

- Nível 4 – Divisão em rotinas
  - Divisão dos dados e rotinas nas classes.
  - Definir totalmente as rotinas da classe em geral resulta em um entendimento melhor da interface da classe.
  - Esse nível de decomposição e projeto de software é frequentemente delegado ao programador como atividade individual.

Níveis de Projeto

- Nível 5 – Projeto interno de rotina
  - Esboçar a funcionalidade detalhada das rotinas.
  - Consiste em escrita de pseudocódigo, pesquisa de algoritmos, decisão sobre como organizar os parágrafos de código em uma rotina e a escrita do código em linguagem de programação.

Modularidade

- A quebra de algo complexo em pedaços menores que sejam gerenciáveis proporciona um melhor entendimento do sistema e partes podem ser desenvolvidas separadamente

Propriedades de Projeto

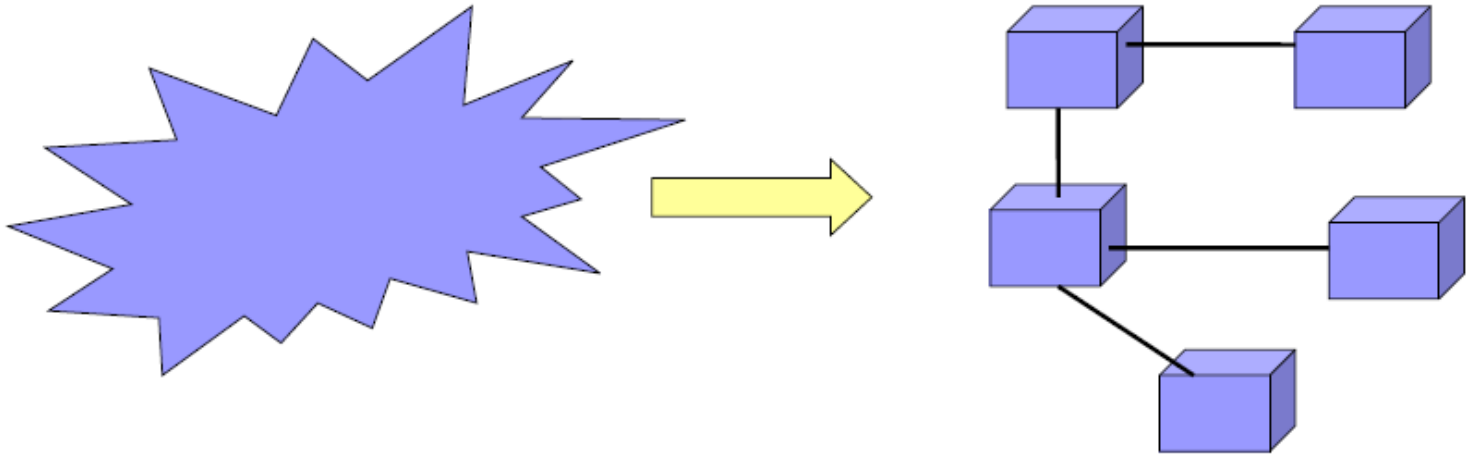
Propriedades de Projeto de Software

- Decomposição e Modularização
- Abstração
- Integridade Conceitual
- Ocultamento de Informação
- Coesão
- Acoplamento

Decomposição e Modularização

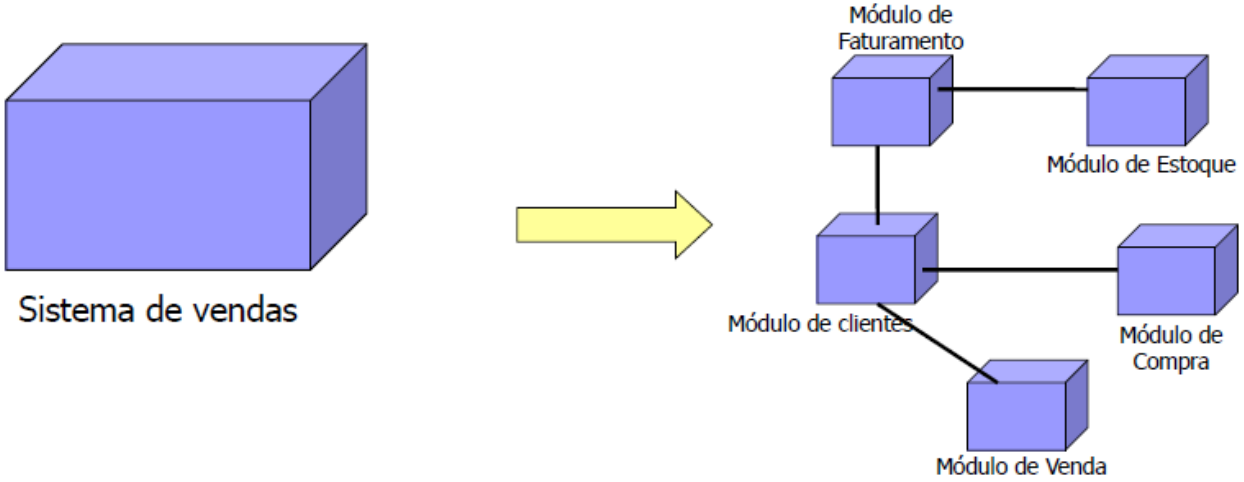
Modularidade

- **Decomponibilidade**: decomposição de um problema em vários subproblemas, cujas soluções possam ser alcançadas separadamente



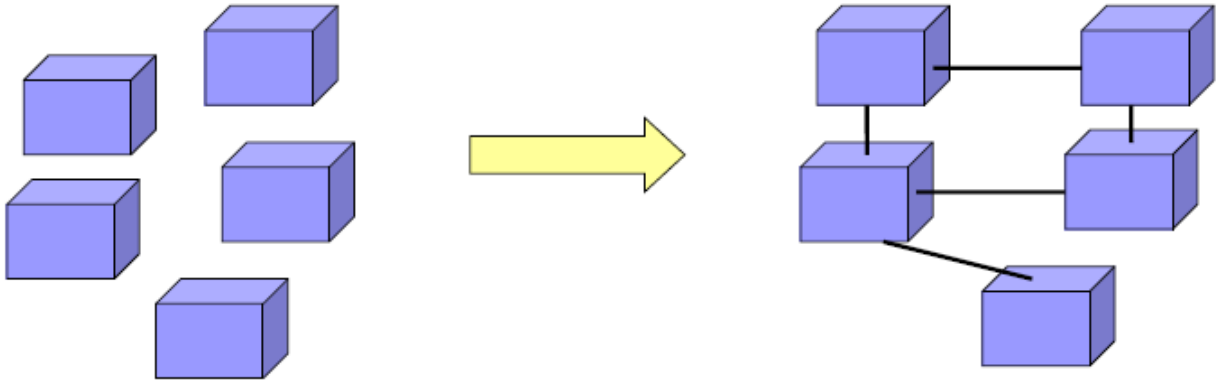
Modularidade

- Exemplo



Modularidade

- **Componibilidade**: produção de elementos de software os quais podem ser livremente combinados com outros para produzir novos sistemas



Modularidade

- Em Engenharia de Software:
  - Partes menores = módulos (pacotes, componentes, classes, etc)

Modularidade

- Controlar a **quantidade** e a forma da comunicação entre módulos é um passo fundamental para obtenção de arquiteturas modulares
  - **Poucas interfaces**: cada módulo deve se comunicar com tão poucos outros quanto possível
  - **Pequenas interfaces**: Se dois módulos se comunicarem, eles devem trocar tão pouca informação quanto possível

Abstração

Abstração

- Conceito que permite "usar" um módulo sem conhecer detalhes de sua implementação
  - Exemplo: Classe Scanner
  - Implementar essa classe pode ser difícil
  - Mas usar é simples:
    - String token = obj1.nextLine();

Integridade Conceitual

Integridade Conceitual

- Funcionalidades de um sistema devem ser coerentes
- Sistema não pode ser um "amontoado" de funcionalidades sem nenhuma coerência ou consistência

Exemplos

- Botão "sair" é idêntico em todas as telas
- Se um sistema usa tabelas para apresentar resultados, todas as tabelas têm o mesmo leiaute
- Todos os resultados são mostrados com 2 casas decimais

Integridade Conceitual vale também para o projeto e código de um sistema

Exemplos (em nível de projeto/código)

- Todas as variáveis seguem o mesmo padrão de nomes
  - contra-exemplo: `nota_total` vs `notaMedia`
- Todas as páginas usam o mesmo framework (mesma versão)
- Se um problema é resolvido com uma estrutura de dados X, todos os problemas parecidos também usam X

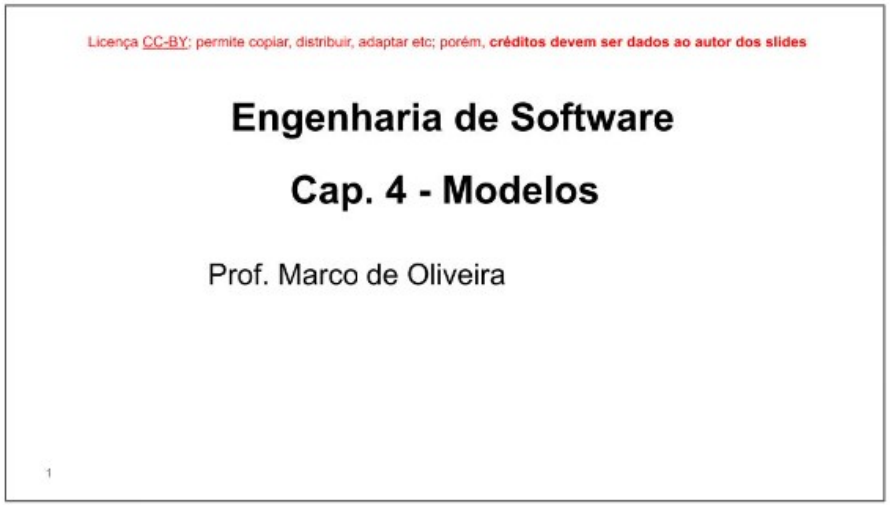
Integridade Conceitual = coerência e padronização de funcionalidades, projeto e implementação

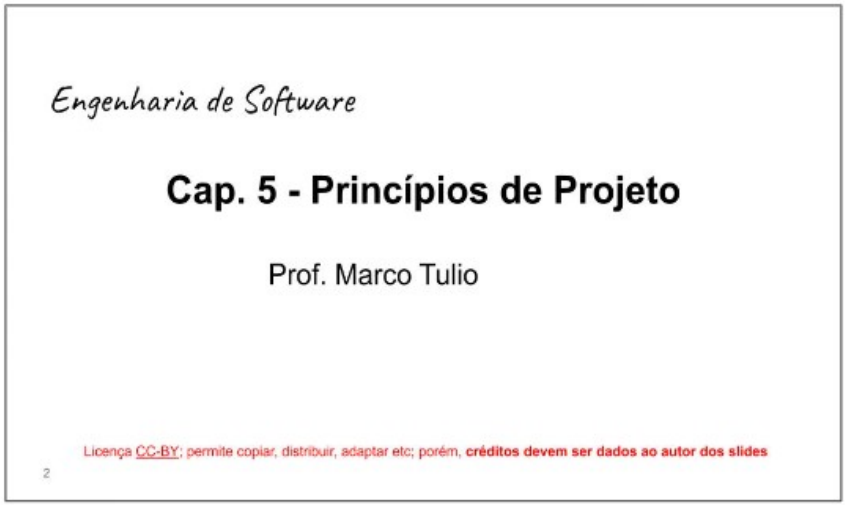
Exemplo

Contra-Exemplo

Exemplo

- Por que falta integridade conceitual nesses slides?





Motivação: integridade conceitual facilita uso e entendimento de um sistema

Exemplo

- Comente o cenário abaixo:
  - Em um conhecido sistema de blogs, quando um usuário incluía um sinal de interrogação no título de um post, uma janela era aberta, solicitando que ele informasse se desejava receber respostas para esse post.No entanto, essa possibilidade deixava os usuários confusos, pois já existia no sistema a possibilidade de comentar posts.

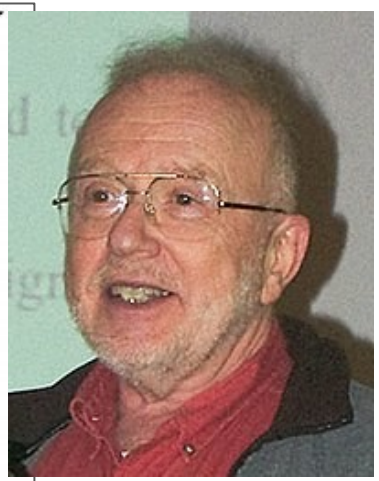
Samuel Roso e Daniel Jackson, pesquisadores do MIT, nos EUA (adaptado),

Ocultamento de Informação
Information Hiding

Origem do conceito (David Parnas, 1972)

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University



This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

Benefícios do Ocultamento de Informação

- Desenvolvimento em paralelo;
- Flexibilidade a mudanças;
- Facilidade de entendimento.

Critério

- Ocultamento de informação recomenda que classes devem esconder detalhes de implementação que estão sujeitos a mudanças (requisitos que ela implementa, algoritmos e estruturas de dados).
- Modernamente, os atributos e métodos que uma classe pretende encapsular são declarados com o modificador de visibilidade privado.

Como?

- Porém, se uma classe encapsular toda a sua implementação ela não será útil. Dito de outra forma, uma classe para ser útil deve tornar alguns de seus métodos públicos, isto é, permitir que eles possam ser chamados por código externo. Código externo que chama métodos de uma classe é dito ser cliente da classe. Dizemos também que o conjunto de métodos públicos de uma classe define a sua interface. A definição da interface de uma classe é muito importante, pois ela constitui a sua parte visível.

```java
import java.util.Hashtable;

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");
  }

}
```

```java
import java.util.Hashtable;

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");
  }

}
```

placa

modelo

```java
import java.util.Hashtable;

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");
  }

}
```

Construtor, cria a Hashtable

```
import java.util.Hashtable;

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");
  }
}
```

Problema: clientes precisam manipular uma estrutura de dados interna da classe, para estacionar um veículo

Problema

- Classes precisam de um pouco de "privacidade"
- Até para evoluir de forma independente dos clientes
- Código anterior: clientes manipulam a hashtable
- Comparação: clientes não podem entrar na cabine do estacionamento e eles mesmo anotar os dados do seu carro no "livro" do estacionamento

Agora uma versão com ocultamento de informação

1

```java
import java.util.Hashtable;

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");
  }

}
```

```java
import java.util.Hashtable;

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");
  }

}
```

2

```java
import java.util.Hashtable;

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");
  }

}
```

3

Resultado: classe Estacionamento fica livre para alterar a sua estrutura de dados interna

Ocultamento de Informação

- Classes devem ocultar detalhes internos de sua implementação (usando modificador **private**)
- Principalmente aqueles sujeitos a mudanças
- Adicionalmente, interface da classe deve ser estável
- **Interface** = conjunto de métodos públicos de uma classe

Curiosidade

- Mensagem de Bezos para a equipe de desenvolvedores da Amazon: diretrizes para projeto de software (adaptada):
  - Todos os times devem, daqui em diante, garantir que os sistemas exponham seus dados e funcionalidades por meio de interfaces.
  - Os sistemas devem se comunicar apenas por meio de interfaces.

Curiosidade

- Não deve haver outra forma de comunicação: sem links diretos, sem leitura direta em bases de dados de outros sistemas, sem memória compartilhada ou variáveis globais ou qualquer tipo de backdoor. A única forma de comunicação permitida é por meio de interfaces.
- Não importa qual tecnologia vocês vão usar: HTTP, CORBA, PubSub, protocolos específicos — isso não interessa. Bezos não liga para isso.

Curiosidade

- Todas as interfaces, sem exceção, devem ser projetadas para uso externo. Ou seja, os times devem planejar e projetar interfaces pensando em usuários externos. Sem nenhuma exceção à regra.
- Quem não seguir essas recomendações está demitido.
- Obrigado; tenham um excelente dia!

Coesão

Coesão

- Uma classe deve ter uma única função, isto é, oferecer um único serviço
- Vale também para outras unidades: funções, métodos, pacotes, etc.

Coesão

- É o **nível de integridade/projeto** interno de uma classe
- Termo utilizado para indicar o **grau** em que uma classe tem um **único** e bem definido **propósito**.
- Benefício de **alta coesão**: Classes fáceis de manter (e menos freqüentemente alteradas) e tendência a uma maior reusabilidade que classes com um menor nível de coesão.
- Quão proximamente são relacionadas as atividades dentro de um única classe (componente) ou entre um grupo de classes?
  - Componentes altamente coesos = relacionados a apenas UMA funcionalidade

Coesão

- Propriedade de **auto contenção** de um objeto. Quanto mais coeso, menos acoplado.
- Classes com **alta coesão** têm responsabilidades bem definidas e são difíceis de dividir em duas ou mais classes;
- Classes com **baixa coesão** tratam de responsabilidades diversas e em geral podem ser divididas;
- Objetiva-se tentar **maximizar** a coesão das classes.

Contra-exemplo 1

```
float sin_or_cos(double x, int op) {
  if (op == 1)
    "calcula e retorna seno de x"
  else
    "calcula e retorna cosseno de x"
}
```

Contra-exemplo 1

```
float sin_or_cos(double x, int op) {
  if (op == 1)
    "calcula e retorna seno de x"
  else
    "calcula e retorna cosseno de x"
}
```

Deveria ser quebrada em duas funções: sin e cos

Contra-exemplo 2

```
class Estacionamento {
  ...
  private String nome_gerente;
  private String fone_gerente;
  private String cpf_gerente;
  private String endereco_gerente;
  ...
}
```

Contra-exemplo 2

```
class Estacionamento {
  ...
  private String nome_gerente;
  private String fone_gerente;
  private String cpf_gerente;
  private String endereco_gerente;
  ...
}
```

Deveria ser quebrada em duas classes:
Estacionamento e Gerente

Exemplo: Coesão

- Trata-se de uma classe coesa, pois todos os seus métodos implementam operações importantes em uma estrutura de dados do tipo Pilha.

```
class Stack<T> {
  boolean empty() { ... }
  T pop() { ... }
  push (T) { ... }
  int size() { ... }
}
```

Exercício de Fixação

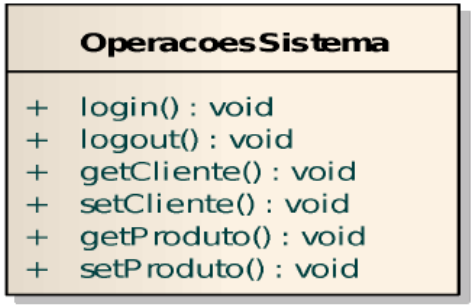
- Classifique as implementações entre coesas e não coesas

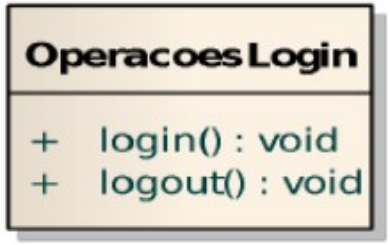
```
1  public class Programa
2   {
3       public void ExibirFormulario()    {
4           //implementação
5       }
6
7       public void ObterProduto()    {
8           //implementação
9       }
10
11      public void gravarProdutoDB   {
12          //implementação
13      }
14  }
```

```
1  public class Programa
2   {
3         public void MostrarFormulario()      {
4           //Implementação
5         }
6
7       public void BotaoGravarProduto( ) {
8           Produto.gravarProduto();
9       }
10
11   }
```

Exercício de Fixação

- Classifique as classes quanto à coesão





```
public class Calculadora {

 public double calculo(double x, double y, int op) {
  if (op == 1)
    return x + y;
  else
    if (op == 2)
      return x - y;
    else
      if (op == 3)
        return x * y;
      else
        if (op == 4)
          return x / y;
        else
          return -1;
 }
```

```
public class Principal {
   public static void main(String args[]){
      Calculadora hp= new Caculadora();
      System.out.println("Soma = " + hp.calculo(5,3,1));
   }
}
```

```
public class Calculadora {

 public double soma(double x, double y) {
   return x + y;
 }

 public double diferenca(double x, double y) {
   return x - y;
 }

 public double produto(double x, double y) {
   return x * y;
 }

 public double divisão(double x, double y) {
   return x / y;
 }
}
```

```
public class Principal {
   public static void main(String args[]){
      Calculadora hp= new Caculadora();
      System.out.println("Soma = " + hp.soma(5,3));
   }
}
```

```
public class Pedido {
  private int pedidoId;
  private double total;
  private String nome_cliente;
  private String endereco_cliente;
  private String cidade_cliente;
  …
}
```

Acoplamento

Acoplamento

- **Congeneridade** é um termo similar a **acoplamento** ou dependência, para não confundir com acoplamento do projeto estruturado, alguns autores utilizam esse termo.
- Do latim **connascence** ”nascidos juntos“
- Elementos congêneres

Acoplamento

- Nenhuma classe é uma ilha ...
- Classes dependem umas das outras (chamam métodos de outras classes, estendem outras classes, etc)
- A questão principal é a **qualidade** desse acoplamento
- Dois tipos:
  - Acoplamento aceitável ("bom")
  - Acoplamento ruim

Acoplamento Aceitável

- Classe A depende de uma classe B:
  - Mas a classe B possui uma interface estável
  - Classe A somente chama métodos da interface de B

```java
import java.util.Hashtable;

public class Estacionamento {

  private Hashtable<String,String> veiculos;

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

  public void estaciona(String veiculo, String placa) {
    veiculos.put(veiculo, placa);
  }

  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");
  }

}
```

Classe Estacionamento depende (está acoplada) à classe Hashtable, mas esse acoplamento é aceitável

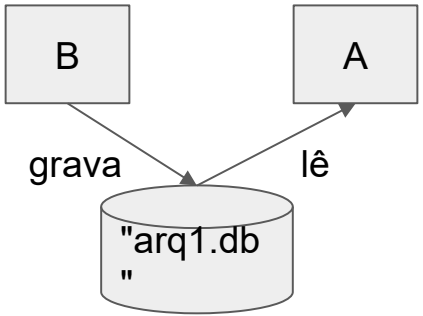
Acoplamento Ruim

- Classe A depende de uma classe B:
  a. Mas interface da classe B é instável
  b. Ou então a dependência não ocorre via interface de B

Como uma classe A pode depender de uma classe B sem ser via a interface de B?

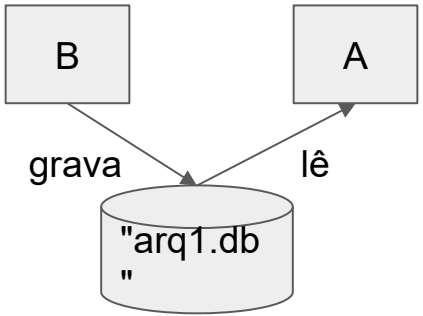
```
class A {

  private void f() {
    int total;
    ...
    File f = File.open("arq1.db");
    total = f.readInt();
    ...
  }

}
```

```
class B {
  private void g() {
    int total;
    // computa valor de total
    File f = File.open("arq1.db");
    f.writeInt(total);
    ...
    f.close();
  }
}
```



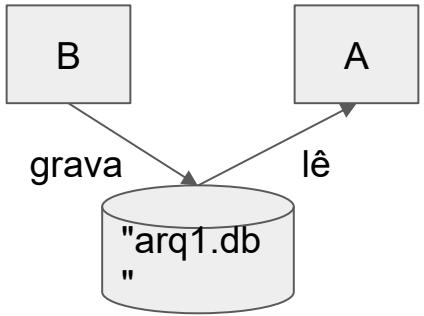
```
class A {

  private void f() {
    int total;
    ...
    File f = File.open("arq1.db");
    total = f.readInt();
    ...
  }

}
```

```
class B {
  private void g() {
    int total;
    // computa valor de total
    File f = File.open("arq1.db");
    f.writeInt(total);
    ...
    f.close();
  }
}
```



Problema

- Classe B não sabe que a classe A é sua "cliente"
- Logo, B pode mudar o formato do arquivo ou mesmo deixar de salvar o dado que é lido por A



Tornando acoplamento ruim em bom

```
class B {

  int total;

  public int getTotal() {
    return total;
  }

  private void g() {
    // computa valor de total
    File f = File.open("arq1");
    f.writeInt(total);
    ...
  }
}
```

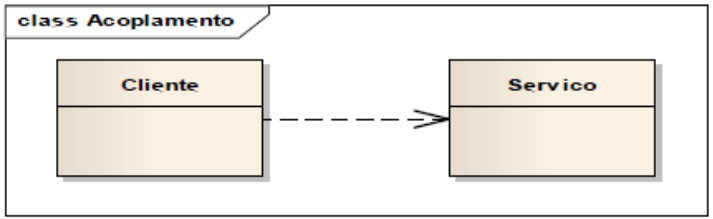
```
class A {

  private void f(B b) {
    int total;
    total = b.getTotal();
    ...
  }
}
```

Tornando acoplamento ruim em bom

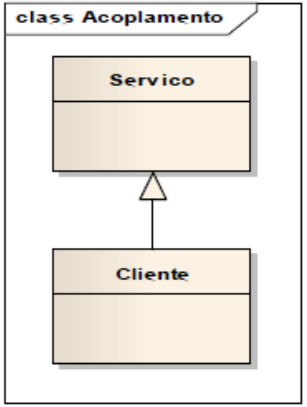
```
class B {

  int total;

  public int getTotal() {
    return total;
  }

  private void g() {
    // computa valor de total
    File f = File.open("arq1");
    f.writeInt(total);
    ...
  }
}
```

```
class A {

  private void f(B b) {
    int total;
    total = b.getTotal();
    ...
  }
}
```

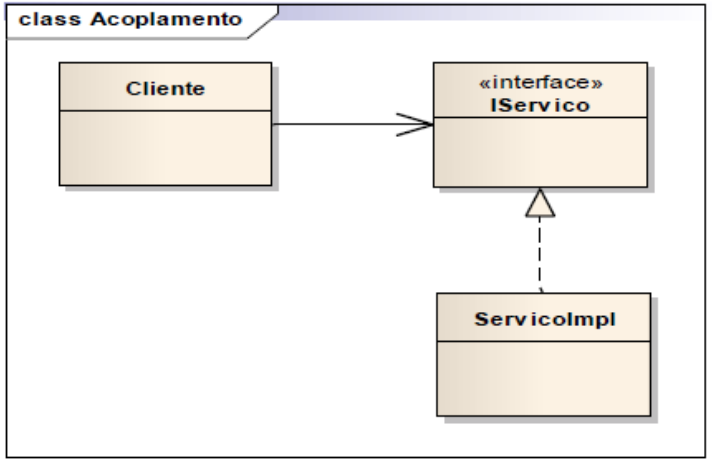
Acoplamento



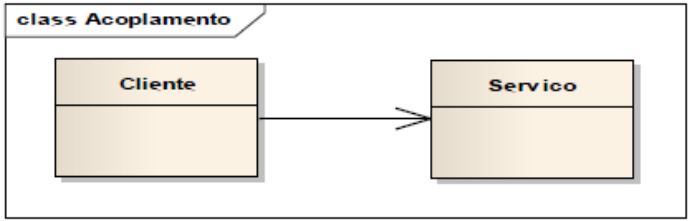
Acoplamento Mais Fraco



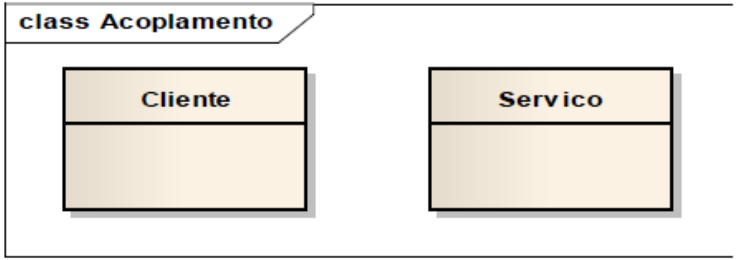
Acoplamento Forte



Acoplamento Fraco Abstrato



Acoplamento Fraco



Sem Acoplamento

Frase muito comum

Maximize a coesão, minimize o acoplamento

Mas cuidado: minimize (ou elimine) principalmente o acoplamento ruim

Acoplamento

- **Com uma classe possuindo forte acoplamento, temos os seguintes problemas:**
  - Mudanças a uma classe relacionada força mudanças locais à classe
  - A classe é mais difícil de entender isoladamente
  - A classe é mais difícil de ser reusada, já que depende da presença de outras classes
  - Causa o efeito cascata(ou dominó)
- Deve-se tentar **minimizar** o **acoplamento** para evitar a propagação de mudanças e para possibilitar a reutilização de classes.
  - Pedaços menos acoplados são mais fáceis de entender, testar, reusar e manter
  - Promove o paralelismo de implementação

Acoplamento

- **Minimizar o acoplamento é um dos princípios de ouro do projeto OO**
- **Acoplamento se manifesta de várias formas:**
  - X tem um atributo que referencia uma instância de Y
  - X tem uma operação que referencia uma instância de Y
    - Pode ser parâmetro, variável local, objeto retornado pela operação
  - X é uma subclasse direta ou indireta de Y
  - X implementa a interface Y
- **A herança é um tipo de acoplamento particularmente forte**
- **Não se deve minimizar o acoplamento criando algumas poucas classes monstruosas (*God classes*)**

Acoplamento

- Se o único conhecimento que uma classe A tem sobre uma classe B é o que a classe B expôs através de sua interface, então as classes A e B têm um **baixo nível** de acoplamento.
- Se a classe A sabe sobre partes da classe B que não são interfaces de B, então as classes A e B têm um **alto nível** de acoplamento.
- Sendo a classe A conectada a classe B o que aconteceria com a classe A caso seja necessário uma **alteração** na classe B ?
- Significa que:
  - Se B for modificado, A terá que ser modificado ou ao menos verificado
  - Pode ocorrer uma modificação no sistema que obrigue modificações conjuntas em A e B

Acoplamento

- Implementação:
  - Acoplamento é a força (*strength*) da **conexão** entre duas classes. Apesar de parecer simples, o conceito possui algumas nuances, as quais derivam da existência de dois tipos de acoplamento entre classes: acoplamento aceitável e acoplamento ruim.
  - Dizemos que existe um **acoplamento aceitável** de uma classe A para uma classe B quando:
    - A classe A usa apenas métodos públicos da classe B.
    - A interface provida por B é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de B não mudam com frequência; e o mesmo acontece como o comportamento externo de tais métodos. Por isso, são raras as mudanças em B que terão impacto na classe A.

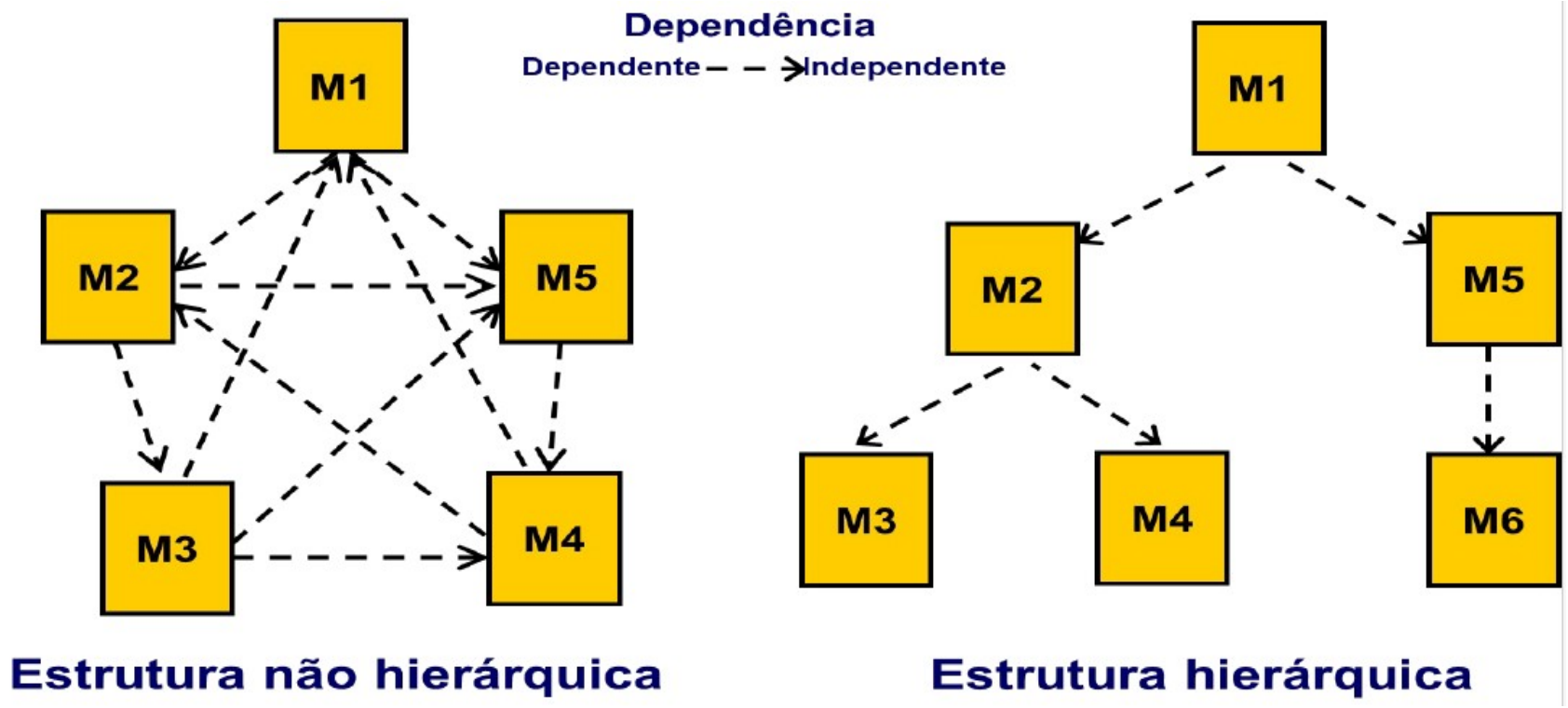
Acoplamento

- Implementação
  - Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:
    - Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
    - Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
    - Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com frequência.

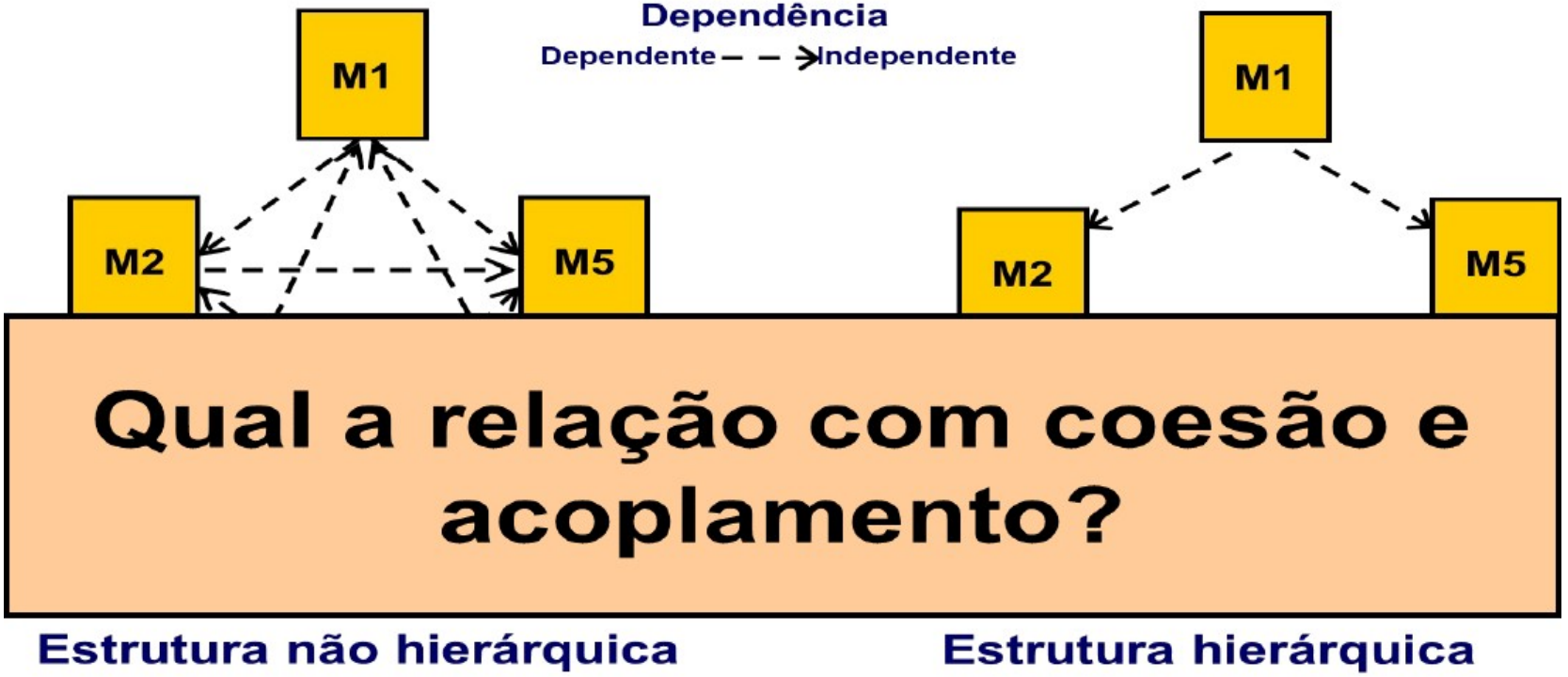
Acoplamento e Coesão

- Não existe 0% de acoplamento, alguma ligação será necessário para o sistema funcionar.
- Uma classe que se relaciona com muitas outras está fortemente acoplada.
- No diagramas de classes que identificar classes quem possuem relacionamentos para tudo quanto é lado não é um bom sinal.
- Muitos Padrões de Projeto ajudam a reduzir o acoplamento.
- Cuidado na divisão!
- Dividir para conquistar sim, dividir para esculhambar não!

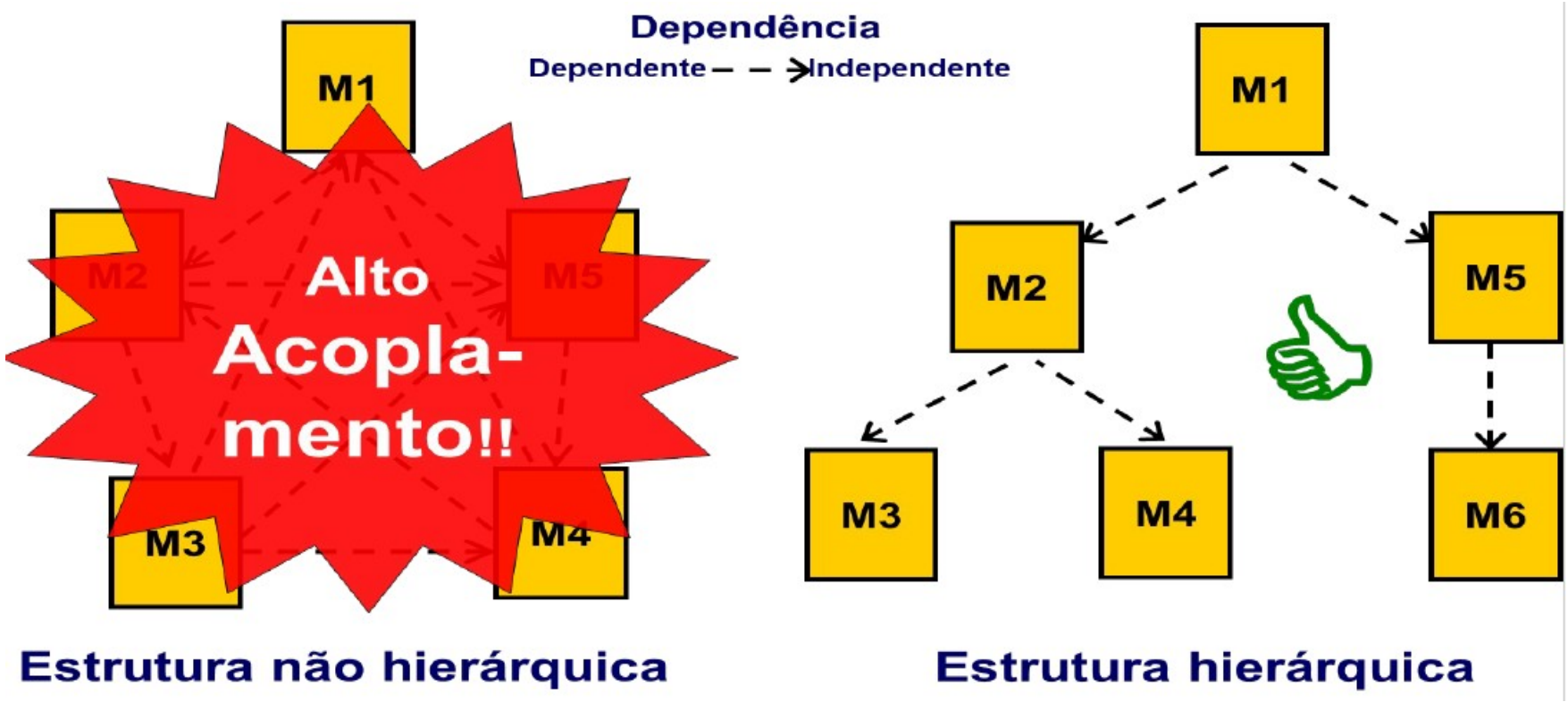
Modularização e Componentização



Modularização e Componentização



Modularização e Componentização



Princípios de Projeto

Princípios de Projeto de Software

- Princípios (ou diretrizes) para projeto de módulos com as propriedades que estudamos antes:
  - Responsabilidade Única
  - Segregação de Interfaces
  - Prefira Interfaces a Classes
  - Aberto/Fechado
  - Demeter
  - Substituição de Liskov

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

Leitura Recomendada

- Busca Ativa 01

- **SOLID** são cinco princípios da programação orientada a objetos que facilitam no desenvolvimento de softwares, tornando-os fáceis de manter e estender. Esses princípios podem ser aplicados a qualquer linguagem de POO.
- Leia o artigo:
  - https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530