ALGORITMOS

CONTEÚDO TEÓRICO E PRÁTICO

- Linguagens;
- Lógica;
- Algoritmos;
- Variáveis;
- Estruturas de Dados;
- Noções de Orientação à objetos;
- Estruturas de Seleção com desvio simples;
- Estruturas de Seleção com desvio múltiplo;
- Estruturas de Repetição;
- Funções e Procedimentos;
- *Arrays*;
- Matrizes;

BIBLIOGRAFIA

- LÓGICA DE PROGRAMAÇÃO
  - Andre Luiz Villar Forbellone e Henri Frederico Eberspacher
- LÓGICA DE PROGRAMAÇÃO
  - Alexandre Berg
- CONCEITOS DE LINGUAGEM DE PROGRAMAÇÃO
  - Robert W. Sebesta

COMO APRENDER
PROGRAMAR

EM UM DIA

LINGUAGEM DE PROGRAMAÇÃO

É uma ferramenta computacional para escrever programas (*softwares*), *onde o* programador **sempre** escreve **para** a máquina. Podemos classificar a linguagem em 3 níveis:

- Linguagem de máquina
- Linguagem de baixo nível
- Linguagem de alto nível

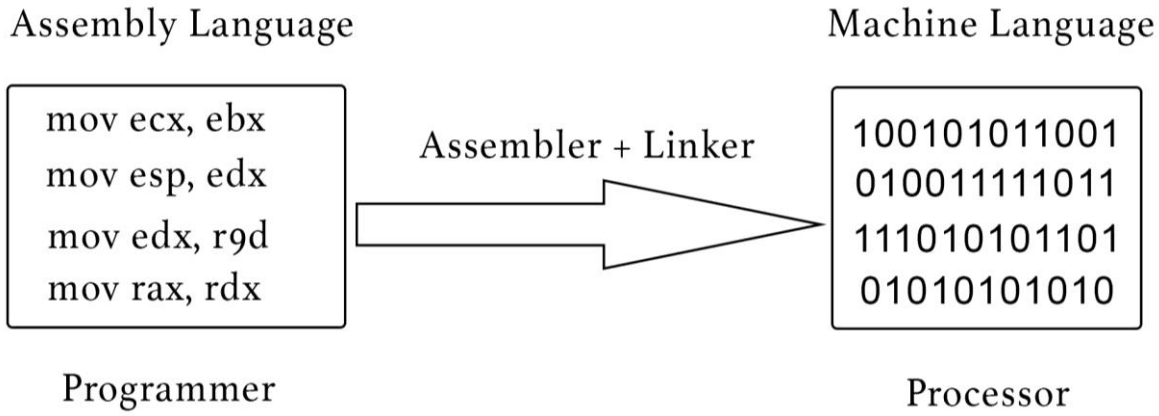
LINGUAGEM DE MÁQUINA

É uma linguagem que é diretamente compreendida pelos hardwares para realizar diversas tarefas. É de grande complexidade para programadores e resume-se em caracteres chamados ***bits*** com valores 0 e 1;

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

LINGUAGEM DE BAIXO NÍVEL

A linguagem de baixo nível foi criada para suprir a dificuldade de se programar em linguagem de máquina. A linguagem **Assembly** é considerada historicamente como o início da Linguagem de Programação, sendo a primeira linguagem de baixo nível. Se código era “traduzido” para linguagem de máquina através de um interpretador, o Assembler.



Assembly Language	Machine Language
ST 1,[801]	00100101 11010011
ST 0,[802]	00100100 11010100
TOP: BEQ [802],10,BOT	10001010 01001001 11110000
INCR [802]	01000100 01010100
MUL [801],2,[803]	01001000 10100111 10100011
ST [803],[801]	11100101 10101011 00000010
JMP TOP	00101001
BOT: LD A,[801]	11010101
CALL PRINT	11010100 10101000
	10010001 01000100

LINGUAGEM DE ALTO NÍVEL

É mais próxima da linguagem humana, com instruções simples para compreensão direta dos **programadores.** Utiliza geralmente comandos no idioma inglês e notações matemática para realizar as tarefas.

Alguns exemplos são COBOL, Fortran, Java, C, C++, Delphi, PHP, entre outros. As linguagens de alto nível dividem-se ainda em 3 tipos: não-estruturada, estruturada e orientada à objetos;

LINGUAGEM NÃO ESTRUTURADA

São linguagens que não **aceitam blocos de estrutura**, de maneira a organizar o código em módulos sequenciais, isto é, o código é desenvolvido em um só bloco. O controle de fluxo de execução se utiliza de **desvios condicionais** para ir de um ponto a outro.

Isso causava problemas de sintaxe, semântica e dificultava a verificação de erros.

Alguns exemplos são COBOL e Fortran.

LINGUAGEM ESTRUTURADA

São linguagens que aceitam blocos de estrutura que definem mudança de comportamento do programa de forma modular. O código é desenvolvido em pequenos módulos ou partes, com o objetivo de tornar o código mais compreensível e fácil de ser depurado.

Alguns exemplos são PASCAL, Fortran-77, C, entre outros.

LINGUAGEM ORIENTADA A OBJETOS

São linguagens que contém as mesmas características de linguagens estruturadas e que implementam conceitos específicos como Encapsulamento, Herança e Polimorfismo, com o objetivo de facilitar o tratamento e significado do código. Busca reproduzir no mundo computacional, o modelo dos objetos do mundo real e suas correlações.

Alguns exemplos são C++, Java, Delphi, PHP, entre outras.

BREVE HISTÓRICO DAS LINGUAGENS

ALGOL (1960)

- Primeira LP com sintaxe definida;
- Influenciou várias LP's, embora não muito utilizada;

COBOL (1960)

- Primeira LP recomendada pelo DoD (EUA);
- Destinada a aplicações comerciais;
- Boa legibilidade / Má redigibilidade;

BREVE HISTÓRICO DAS LINGUAGENS

BASIC (1964)

- Criada na Universidade de Darmouth;
- Fácil aprendizado para uso de estudantes e ciências humanas;

PASCAL (1971)

- Criado para aprendizado de programação estruturada;
- Foco na simplicidade;

BREVE HISTÓRICO DAS LINGUAGENS

C (1972)

- Projetada para programação de sistemas;
- Usada para desenvolvimento do SO UNIX;

PROLOG (1972)

- Trabalha estritamente com lógica;
- Muito usada em Inteligência Artificial;

BREVE HISTÓRICO DAS LINGUAGENS

SMALLTALK (1972)

- Primeira LP orientada à objetos;
- Utiliza ambiente com GUI, hoje amplamente disseminada;

ADA (1983)

- Grande e complexa. 8 anos de desenvolvimento;
- Programação concorrente e sistemas de tempo real;

HISTÓRICO DAS LINGUAGENS

C++ (1985)

- Baseada em C, com orientação à objetos;
- Com rápida aceitação, tornou-se muito complexa;

JAVA (1995)

- OO, baseada em C++, porém bem mais simples;
- Fácil aprendizado, enfatiza a portabilidade;

LÓGICA

Os primeiros conceitos de Lógica surgiram com Aristóteles, na Grécia Antiga. Hoje podemos definir Lógica como uma sequência de etapas naturais para obter um resultado esperado. Na lógica computacional, escrevemos instruções de comportamento para solução de um problema computacional.

No aprendizado de Lógica, pouco importa a Linguagem de Programação

ARGUMENTOS

- Os argumentos são objetos do raciocínio lógico. São proposições que englobam **premissas** e uma **conclusão.** A conclusão sempre é justificada pelas premissas:

  - Todo aluno de Computação precisa estudar Lógica;
  - José é aluno de Computação;
  - Logo, José precisa estudar Lógica;

ARGUMENTOS

Argumentos tem por objetivo justificar uma conclusão, conseguida a partir de dados de problema. Por exemplo, analise o raciocínio abaixo:

- **Você me traiu. Pois, disse que ia estudar e meu irmão lhe viu na boate;**

A conclusão foi tirada a partir de premissas, sem que necessariamente sejam verdadeiras. A Lógica procura validar ou invalidar um argumento, verificando se ele está de acordo com as premissas.

Ao analisar as premissas e propor uma conclusão, realizamos um processo chamado **Inferência**;

ARGUMENTOS

Vamos fazer um exercício de análise sobre alguns argumentos para verificar se eles são válidos ou não.

- **Se eu ganhar na loteria, serei rico;**
- **Eu ganhei na loteria;**
- **Logo, sou rico;**

- **Se eu ganhar na loteria, serei rico;**
- **Eu não ganhei na loteria;**
- **Logo, sou pobre;**

DEDUÇÃO E INDUÇÃO

Um argumento dedutivo prevê uma conclusão, baseada nas premissas. Quando as premissas são verdadeiras, é **impossível** que a conclusão seja falsa. Do contrário, somos levados a uma conclusão inválida.

- **Todos os mamíferos são mortais;**
- **Todas as cobras são mortais;**
- **Logo, se eu jogar outra pedra no lago ela irá afundar;**

- **Todos os diamantes são duros;**
- **Alguns diamantes são joias;**
- **Algumas joias são duras;**

DEDUÇÃO E INDUÇÃO

Já um argumento indutivo prevê uma conclusão baseada em observação ou experiência, nem sempre relacionada com as premissas.

- **Joguei uma pedra no lago e ela afundou;**
- **Joguei outra pedra no lago e ela afundou;**
- **Todas as cobras são mamíferos;**

- **A vacina funcionou nos ratos;**
- **A vacina funcionou nos macacos;**
- **Logo funcionará nos humanos;**

- **80% vão votar no candidato A, logo ele ganhará.**

OPERADORES LÓGICOS

Operadores lógicos são operações que produzem um resultado lógico, verdadeiro ou falso.

- **CONJUNÇÃO:** quando temos uma operação de conjunção (“conjunto”), todas as premissas devem ser verdadeiras para o resultado também ser verdadeiro.
- **DISJUNÇÃO:** quando temos uma disjunção (“disjuntor”), todas as premissas devem ser falsas para o resultado também ser falso.
- **NEGAÇÃO:** quando temos uma operação de negação, o resultado lógico será invertido.

OPERADOR DE CONJUNÇÃO

Você conhece JAVA	Você conhece C++	Resultado
FALSO	**FALSO**	**FALSO**
FALSO	**VERDADEIRO**	**VERDADEIRO**
VERDADEIRO	**FALSO**	**VERDADEIRO**
VERDADEIRO	**VERDADEIRO**	**VERDADEIRO**

OPERADOR DE DISJUNÇÃO

Você conhece JAVA	Você conhece C++	Resultado
FALSO	**FALSO**	**FALSO**
FALSO	**VERDADEIRO**	**VERDADEIRO**
VERDADEIRO	**FALSO**	**VERDADEIRO**
VERDADEIRO	**VERDADEIRO**	**VERDADEIRO**

VARIÁVES

Variáveis são espaços reservados na memória para armazenar dados. Os valores armazenados podem sofrer variações ao longo da execução, por isso o nome.

Algumas linguagens são **fortemente tipadas**, isto é, os dados são armazenados em espaços na memória com tipo definido, significando que as variáveis precisam ser tipadas antes de serem utilizadas.

Outras linguagens são fracamente tipadas, isto é, os dados são armazenados em espaços na memória sem tipo definido, significando que as variáveis são genéricas e podem armazenar qualquer tipo a qualquer tempo.

TIPOS DE DADOS

Em linguagens fortemente tipadas, cada variável armazena um tipo de dado distinto e único, já definido anteriormente. Este tipo está diretamente ligado ao espaço que será reservado para o armazenamento daquele valor.

Na lógica de programação, aprendemos que são 4 tipos básicos:

- Inteiro
- Real
- Literal
- Lógico

TIPO DE DADOS INTEIRO

Compreende todos os números negativos ou positivos (inclusive nulos) inteiros, sem casas decimais ou partes fracionárias.

O tipo inteiro permite que sejam feitos cálculos com os dados, exceto se a operação tiver possibilidade de produzir resultado decimal.

Exemplo de dados inteiros: 5, -10, 0, 321351;

TIPOS DE DADOS REAL

Compreende todos os números reais (inclusive nulos) inteiros ou com casas decimais / partes fracionárias.

O tipo real pode também armazenar um número inteiro simples, mas este será tratado como um número real, isto é, possivelmente acrescentado com zeros após a vírgula ou com denominador 1. Aceita também qualquer tipo de cálculo.

Exemplo de dados reais: 5165, 30.6, 0, -2.5;

TIPO DE DADOS LITERAL

Compreende qualquer caractere, texto, ou números, sendo todos estes tratados como informação textual, geralmente representada sendo delimitada por aspas.

Um número armazenado em variável do tipo literal não permite cálculo, pois este é tratado como um caractere.

Exemplo de dados literais: “João”, “130”, “Estrada do Fundão, nº 10”;

TIPO DE DADOS LÓGICO

Um variável do tipo lógico armazena apenas resultados do tipo lógico, sendo binário, verdadeiro ou falso, permitindo apenas uma resposta armazenada.

Embora seja uma informação textual, não deve ser confundido com o tipo literal, pois um valor lógico é tratado internamente como bit.

Exemplo de dados lógicos: “Sim/Não”, “Verdadeiro/Falso”, “Ligado/Desligado”;

DECLARAÇÃO DE VARIÁVEIS

Nas linguagens fortemente tipadas, toda variável deve ser declarada, antes da utilização para reconhecimento do sistema ao interpretar/compilar. Na declaração, a variável deve possuir um nome e um tipo; em algumas linguagens também deve possuir um valor inicial, em outras o valor é implícito.

A nomeação de variáveis segue uma regra específica e convencionada:

- Nomes de variáveis não podem conter espaços;
- Não podem conter acentos ou caracteres especiais;
- Não deve começar com números (mas pode conter números);
- Não devem ser palavras reservadas da linguagem;
- Por convenção, deve-se usar letras minúsculas;

PALAVRAS RESERVADAS			
aleatorio	e	grauprad	passo
abs	eco	inicio	pausa
algoritmo	enquanto	int	pi
arccos	entao	interrompa	pos
arcsen	escolha	leia	procedimento
arctan	escreva	literal	quad
arquivo	exp	log	radpgrau
asc	faca	logico	raizq
ate	falso	logn	rand
caracter	fimalgoritmo	maiusc	randi
caso	fimenquanto	mensagem	repita
compr	fimescolha	minusc	se
copia	fimfuncao	nao	sen
cos	fimpara	numerico	senao
cotan	fimprocedimento	numpcarac	timer
cronometro	fimrepita	ou	tan
debug	fimse	outrocaso	verdadeiro
declare	função	para	xou

Palavras Reservadas do Pascal				
and	downto	in	packed	to
Array	else	inline	procedure	type
asm	end	interface	program	unit
begin	file	label	record	until
case	for	mod	repeat	uses
const	foward	nil	set	var
constructor	function	not	shl	while
destructor	goto	object	shr	with
div	if	of	string	xor
do	Implementation	Or	Then	

ATRIBUIÇÃO DE VALORES

Por vezes, não raramente, o valor de uma variável é definido sem entrada manual do usuário. Essa operação é chamada de atribuição, quando definimos via código o conteúdo da variável.

O símbolo de atribuição varia de linguagem para linguagem. No Pascalzim o símbolo é **:=** (sinal de menor, hífen), enquanto no Visualg o símbolo **<-** (dois pontos, igual). Para variáveis do tipo literal, também podemos atribuir o valor vazio. Veja outros exemplos.

nome <- 'João';
altura <- 1.70;

EXPRESSÕES

No processamento dos dados, podemos trabalhar com expressões matemáticas para cálculos. Porém, para realizar cálculos matemáticos em algoritmos, todas as expressões matemáticas devem ser linearizadas, já que podemos usar apenas parênteses.

$$\left\{\left[\frac{2}{3}-(5-3)\right]+1\right\}.5$$

```
((2/3-(5-3))+1)*5
```

Operador	Usado	Exemplos
Adição	+	a + b;
Subtração	–	a – b;
Multiplicação	*	a * b;
Divisão	/	a/b;
Divisão inteira	\	a\b;
Resto	%	a%b;
Exponenciação	^ (base^expoente)	a^b;

Operadores Relacionais	
=	Igual a
<> ou !=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a

EXERCÍCIOS

1) Criar um programa para calcular a área do círculo, recebendo os valores adequados como entrada;

2) Criar um programa para calcular o Índice de Massa Corpórea (IMC), recebendo os valores adequados como entrada. Deve-se dividir o peso pela altura ao quadrado;

3) Criar um programa para calcule a média de um curso, sendo dois testes, um trabalho e uma prova com peso 2;

DEFINIÇÕES

Quando temos uma sequência de etapas organizadas de maneira lógica, damos o nome à essa sequência de **ALGORITMO.** A esquematização de um algoritmo pode ser definida em partes:

- **Entrada** – Recebimento dos dados;
- **Processamento** – Manipulação geral dos dados;
- **Saída** – Conclusão ou Resultado adequado à entrada e ao processamento;

Um algoritmo pode ser representado de 3 formas: Narrativa/Descritiva, Fluxograma e Código/Pseudocódigo.

ALGORITMO POR NARRATIVA

Ocorre quando temos a sequência de etapas em forma descritiva, com textos, se assemelhando a uma ‘receita’. Por exemplo, um algoritmo para tomar um banho seria:

- Despir-se;
- Abrir o chuveiro;
- Enxaguar-se;
- Ensaboar-se;
- Enxaguar-se novamente;
- Fechar o chuveiro;
- Secar-se;
- Vestir-se;

EXERCÍCIO

Escrever o algoritmo descritivo/narrativo para levar a cabra, o leão e o capim para o outro lado do rio, 1 por vez, sem que o predador aja.

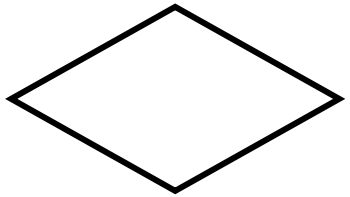
ALGORITMO POR FLUXOGRAMA

É uma maneira de representação gráfica das etapas, utilizando figuras padronizadas. É indicado para algoritmos de pequeno porte, pois do contrário o esquema gráfico fica complexo e exageradamente grande, dificultando a análise.

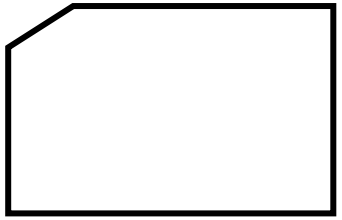
É importante notar que o fluxograma não oferece recursos para representar ou descrever os dados, apenas focando nas etapas. Podem também ocorrer pequenas variações nas representações, a depender do software e cultura de mercado aplicada.

Processamento

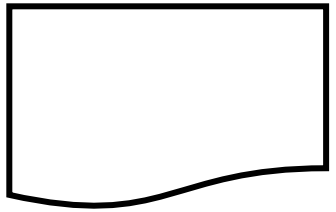
Sequência



Decisão

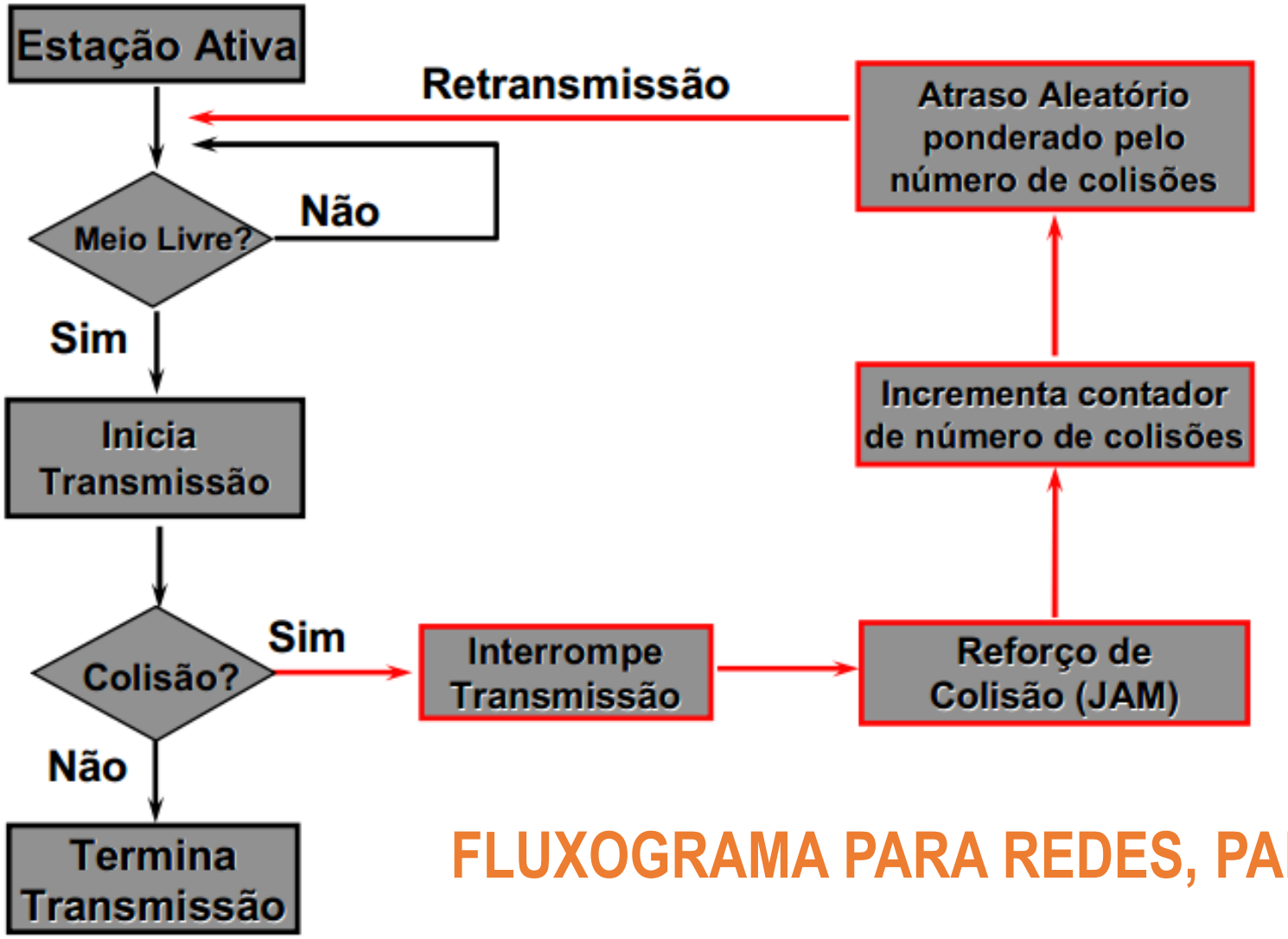


Entrada



Saída

Terminal: Início e Fim



FLUXOGRAMA PARA REDES, PADRÃO 802.3

ALGORITMO POR CÓDIGO

Uso de palavras em forma de comandos, geralmente em inglês, podendo ser em português, para escrever expressões lógicas de modo estruturado. Não se trata de uma linguagem de programação real, portanto, não é padronizado.

Geralmente utiliza linguagens e estruturas exclusivas para aprendizado de lógica, não gerando softwares, sistemas, etc. Para fins de teste, é recomendado sempre simular a execução do algoritmo, comparando o resultado obtido com o resultado esperado. Essa simulação é comumente chamada de **Teste de Mesa**.

Um dos exemplos é a linguagem Pascal.

```
Program soma;
var a, b, c:integer;
  Begin
        read (a)
  read (b)
        c:= A+B
        write (c)
  End.
```

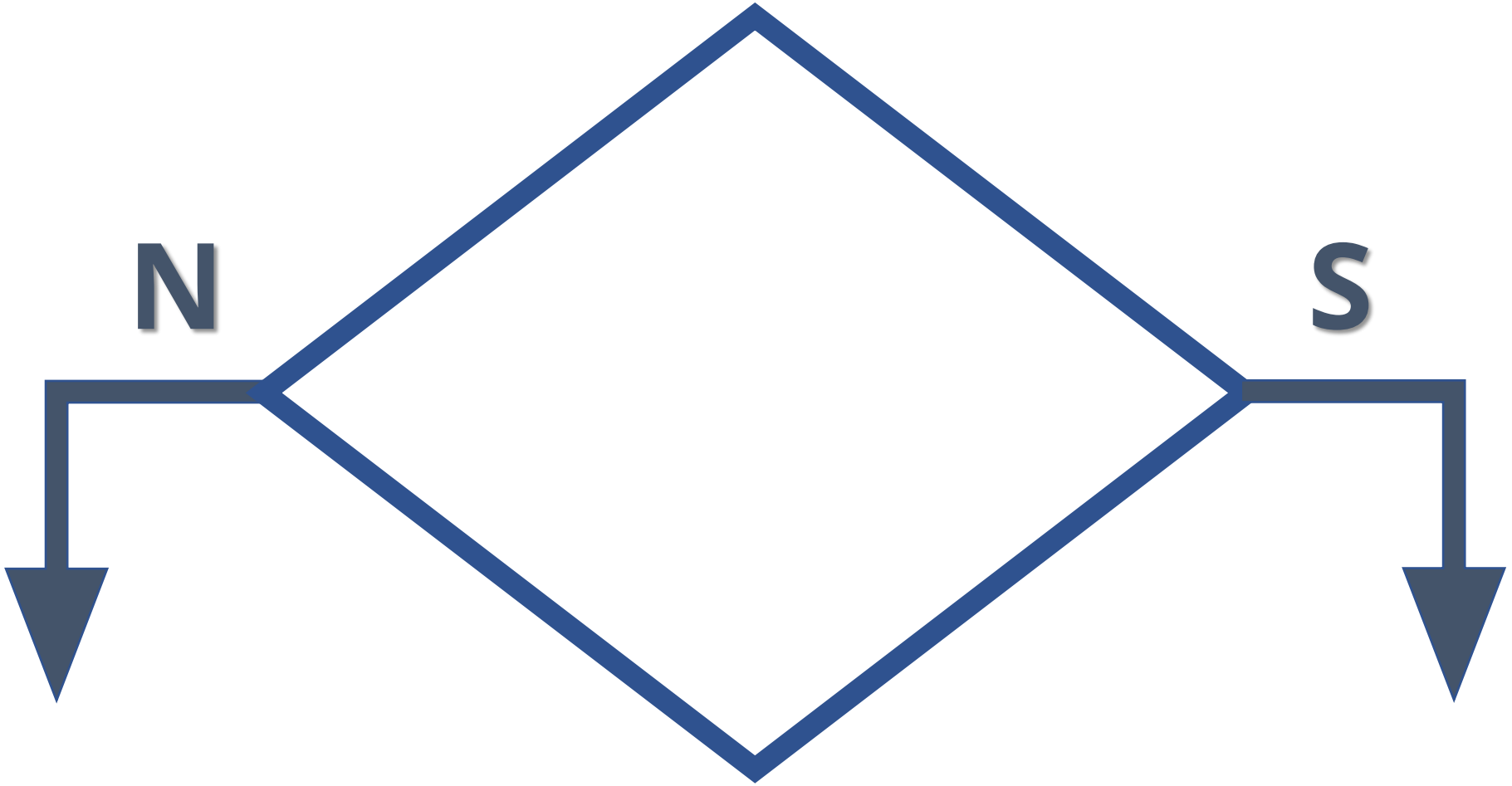
Algoritmo para somar 2 números quaisquer

DESVIOS CONDICIONAIS

Acontecem quando, em determinado momento do código, é preciso fazer uma decisão entre **SIM E NÃO**. Isso faz com que nem todas as linhas do código sejam executadas (mas todas são escritas!).

O código e o diagrama de blocos tomam uma decisão booleana para decidir qual caminho seguir ou qual linha executar. Usamos para isso os operadores relacionais para testar determinada condição, isto é, comparar valores. Se o resultado for verdadeiro (SIM) ocorre uma ação diferente daquela se o resultado for falso (NÃO).

REPRESENTAÇÃO NO FLUXOGRAMA



DESVIO CONDICIONAL SIMPLES

No desvio simples, apenas dois caminhos são possíveis. Um condição sempre é testada como **verdadeira**. Caso seja, o código fará um "desvio" do seu fluxo normal. É importante notar que o código somente desviará se a condição for verdadeira. Caso contrário, seguirá seu fluxo normal. Veja a sintaxe:

```
se <condição> então
        ações/instruções
fim
```

```
se tenho dinheiro então
        como pizza
fim
```

EXEMPLO PRÁTICO

Ler um número qualquer digitado pelo usuário. Caso seja negativo, informar com a mensagem “NÚMERO INVÁLIDO!”

```
var
        x:real
inicio
        leia x
        se x<0 então
                escreva “NÚMERO INVÁLIDO“
        fim
fim
```

DESVIO CONDICIONAL COMPOSTO

No desvio composto, apenas dois caminhos são possíveis, porém é previsto no código uma ação para o resultado “falso”. Desta forma, o código sempre sofrerá um desvio, com uma ação prevista para ambos os casos.

```
se <condição> então
        ações/instruções
senão
        ações/instruções
fim
```

```
se tenho dinheiro então
        como pizza
senão
        miojão
fim
```

EXEMPLO PRÁTICO

Ler um número qualquer, diferente de zero, digitado pelo usuário e informar se é positivo ou negativo.

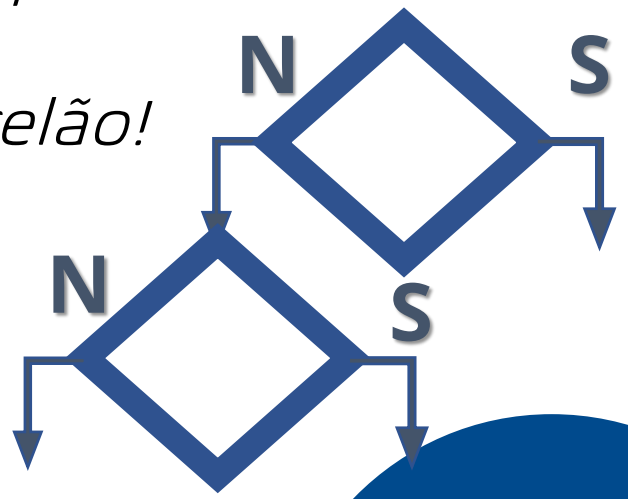
```
var
        x:real
inicio
        leia x
        se x<0 então
                escreva "NÚMERO NEGATIVO"
        senão
                escreva "NÚMERO POSITIVO"
         fim
fim
```

DESVIO CONDICIONAL ENCADEADO

O desvio condicional encadeado faz testes consecutivos. Ocorre quando testamos uma condição e seu resultado (seja verdadeiro ou falso) leva a um novo teste.

```
se <condição> então
        se <condição> então
                ações/instruções
        senão
                ações/instruções
        fim
senao
        ações/instruções
Fim
```

```
se tenho dinheiro então
        se pizza = atum então
                atum top!
        senão
                mussarelão!
        fim
senao
        miojão!
fim
```



DESVIO CONDICIONAL MÚLTIPLO

É usado quando a condição testada oferece muitas opções de resposta. Porém, somente funciona quando a comparação é de **igualdade.** Caso não haja correspondência, há uma última possibilidade de escape.

```
caso <condição>
        seja <valor>: ações/instruções
        seja <valor>: ações/instruções
        seja <valor>: ações/instruções
        senao ações/instruções
fim
```

```
se codigo_pagamento
        seja 1: pagar com dinheiro
        seja 2: pagar com cheque
        seja 3: pagar com cartão
        senao lavar pratos
fim
```

EXEMPLO PRÁTICO

Ler um time digitado pelo usuário. Caso seja o Palmeiras, escrever “Verdão eô!”, caso seja o Corinthians, escrever “Vai curintia!”. Para qualquer outro time escrever “Uuuh!”

```
var
        time:string;
inicio
        leia time;
        caso time
                seja “Palmeiras”: escreva “Verdão eô!”
                seja “Corinthians”: escreva “Vai Curintia!”
                senão: escreva “Uuuh!”
        fim
fim
```

ESTRUTURAS DE REPETIÇÃO

As estruturas de repetição são usadas para repetir a execução de um código todo ou um trecho específico, de modo finito.Ao repetir o código, as variáveis não são reinicializadas, exceto se isso for especificado pela lógica do algoritmo.

Uma estrutura de repetição pode ainda conter uma ou mais estruturas de desvios dentro dela, ou até mesmo uma outra estrutura de repetição. O principal objetivo é evitar a redigitação de um trecho de código.

Temos em geral, 3 estruturas para repetição.

ESTRUTURA ENQUANTO/WHILE

Estrutura onde testamos se uma sentença é verdadeira. Caso seja, o código é repetido. O teste é feito ao iniciar a estrutura, isto é, antes de rodar o código candidato a repetição, o que significa que ele pode não ser executado nenhuma vez, caso a sentença não seja verdadeira logo no início.

```
enquanto <sentenca> faça
        ações/instruções
fim
```

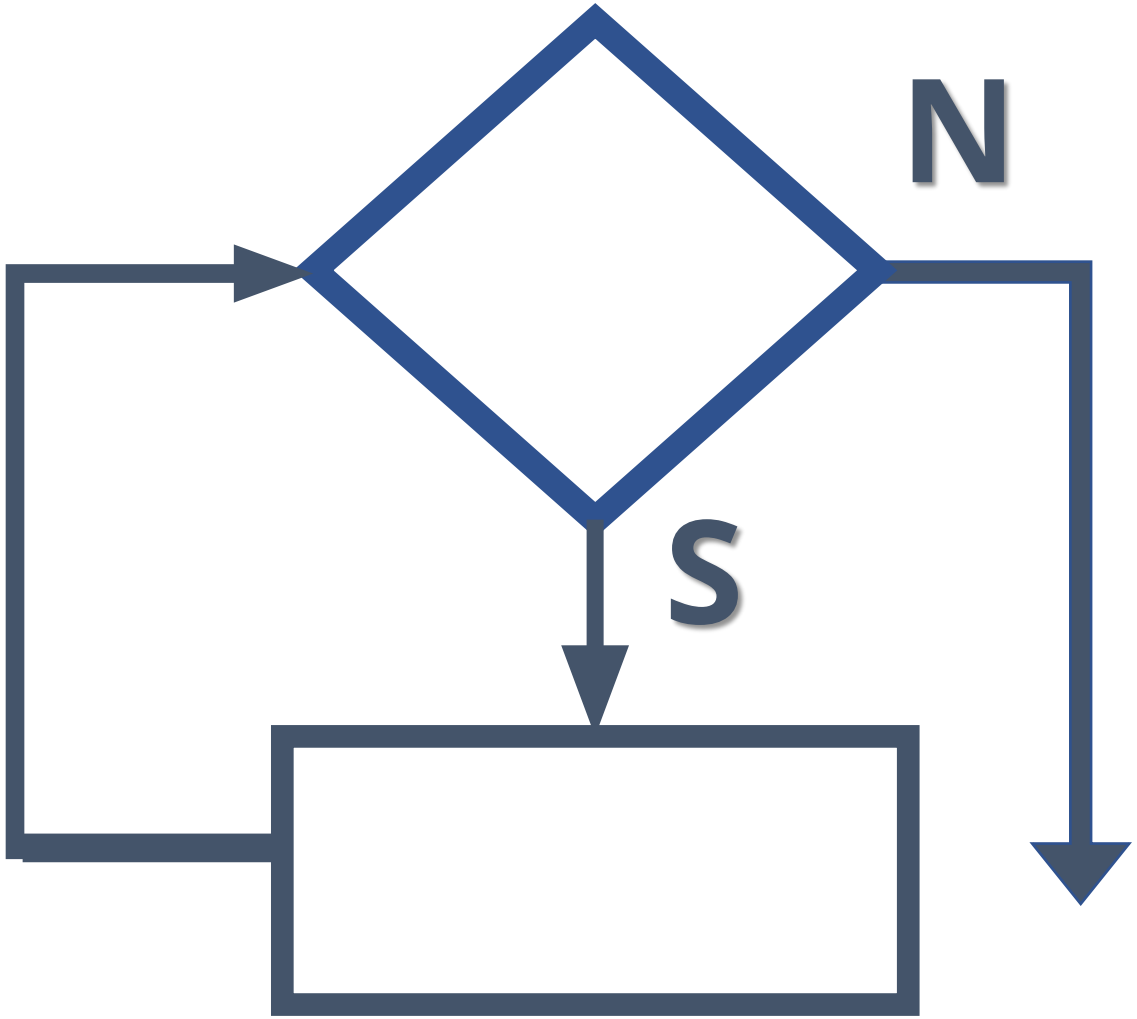
```
enquanto tenho dinheiro faça
        comer pizza
fim
```

EXEMPLO PRÁTICO

Imagine que o usuário deva adivinhar um número. Para isso deve ir tentando digitar números e quando acertar o sistema mostrará uma mensagem avisando-o do acerto.

```
var
        sorteado, numero:integer;
inicio
        sorteado:= 23;
        enquanto (numero<>sorteado) do
                escreva "Digite um número"
                leia (numero)
        fim
        escreva "Parabéns, você acertou";
fim
```

REPRESENTAÇÃO NO FLUXOGRAMA



ESTRUTURA REPITA / REPEAT

É semelhante à estrutura enquanto / while, porém com duas diferenças principais. A sentença testada deve ser falsa para haver repetição e o teste da sentença é feito ao finalizar a estrutura, garantindo que o código seja então executado pelo menos uma vez.

```
repita
        ações/instruções
até <sentenca>
```

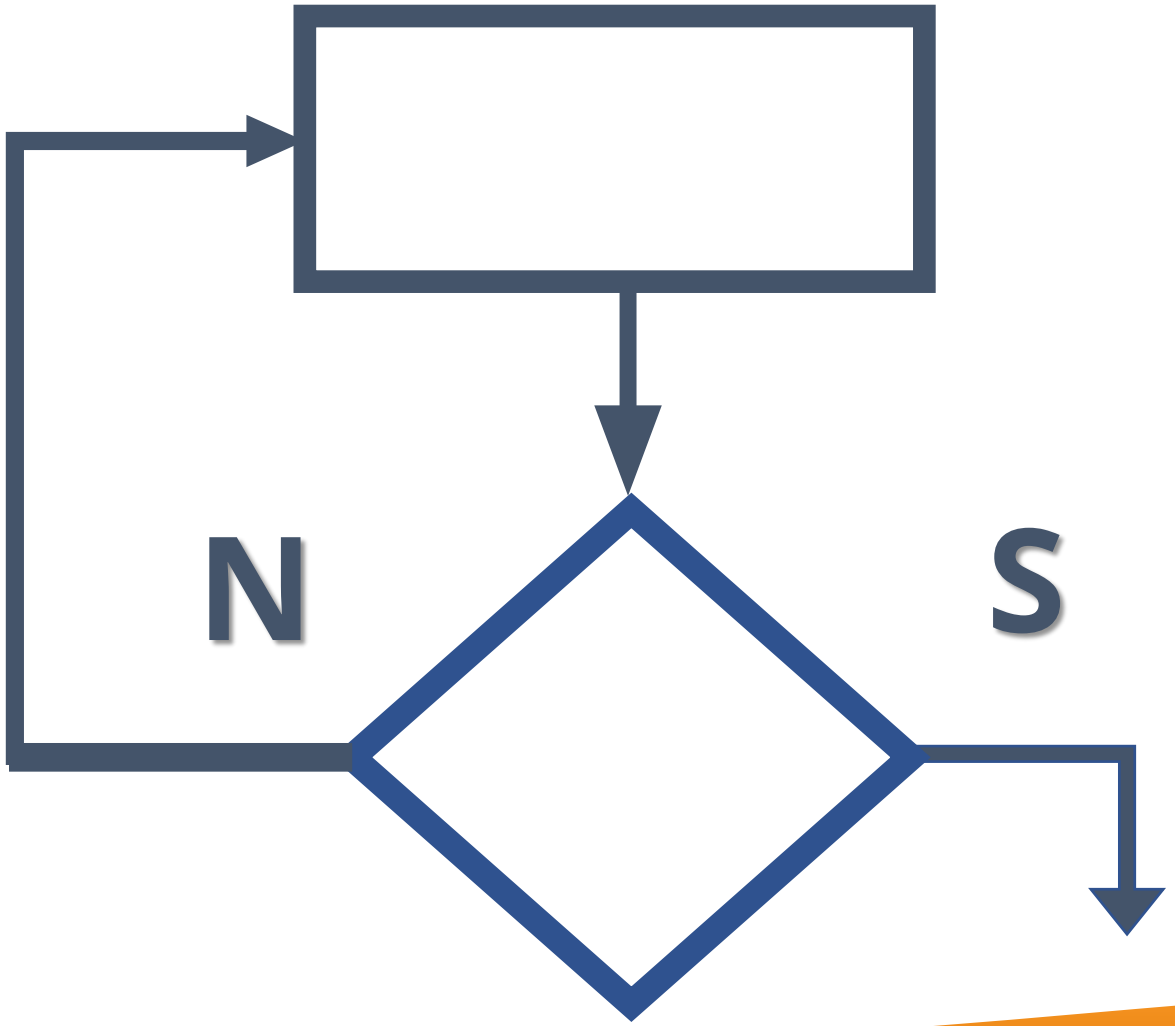
```
repita
        comer pizza
até não ter dinheiro
```

EXEMPLO PRÁTICO

Imagine que o usuário deva adivinhar um número. Para isso deve ir tentando digitar números e quando acertar o sistema mostrará uma mensagem avisando-o do acerto.

```
var
        sorteado, numero:integer;
inicio
        sorteado:= 23;
        repita
                escreva "Digite um número"
                leia numero
        até numero=sorteado
        escreva "Parabéns, você acertou";
fim
```

REPRESENTAÇÃO NO FLUXOGRAMA



ESTRUTURA PARA / FOR

A estrutura para / for é recomendada quando sabe-se o número de repetições do algoritmo. Como a quantidade de repetições é definida, há obrigatoriamente uma variável de controle que conta o número de repetições. Esta variável deve ser necessariamente do tipo inteiro.

```
para <variavel> inicio fim
        ações/instruções
fim
```

```
para contador :=1 a 17 faça
        comer mais um pedaço
fim
```

EXEMPLO PRÁTICO

Ler um número digitado pelo usuário e exibir sua tabuada do 1 ao 10.

```
var
        numero, contador:integer;
inicio
        escreva "Digite um número"
        leia numero
        para contador :=1 a 17 faça
                escreva numero * contador
        fim
fim
```

VETORES E MATRIZES

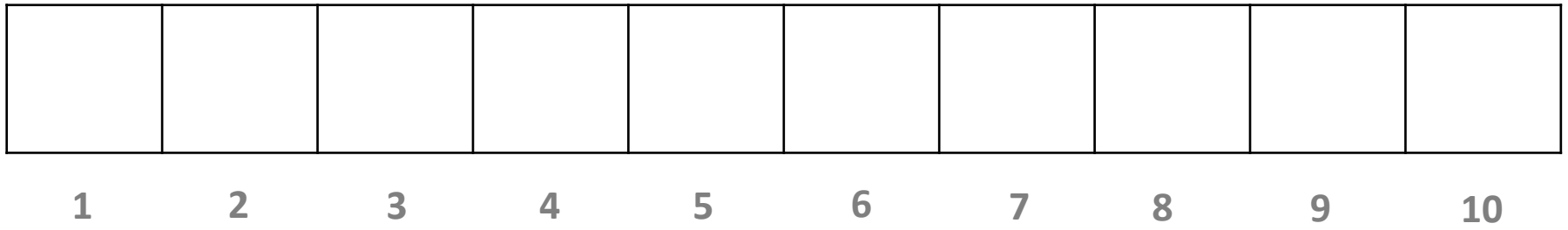
Vetores e Matrizes são considerados coleções de dados dentro da lógica de programação. As variáveis com os tipos primitivos caracterizam por armazenar apenas um valor. Quando precisamos guardar mais valores na mesma variável esta se torna então, uma coleção de dados.

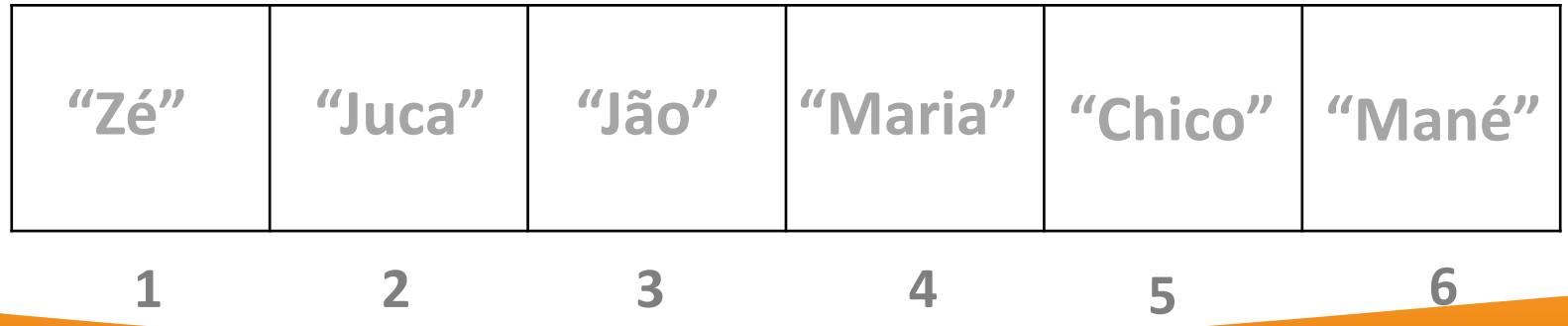
Para organizar a coleção, os valores são numerados, facilitando sua localização. Esses números são chamados de índice. Dependendo da linguagem de programação, o índice pode começar em 0 ou 1.

Na lógica de programação, as coleções guardam apenas um tipo de dados.

AGRUPAMENTO LINEAR

Também chamado de matriz unidimensional, armazena os dados de forma linear. Os índices são organizados sequencialmente, como num livro. Em cada espaço, é armazenado um valor.





DECLARAÇÃO E MANIPULAÇÃO

As coleções nas linguagens de programação tem o nome genérico de **array**. Para declarar uma variável do tipo array, deve-se obrigatoriamente indicar seu índice entre colchetes e o tipo.

A manipulação de coleções de dados está ligada ao uso de estruturas de repetição. A mais comum para uso é a estrutura PARA / FOR, com sua variável contadora referenciando os índices do vetor.

SINTAXE

O vetor é declarado com a palavra chave conjunto, com um tamanho (número total de índices) limitado (fim) e com o tipo de dados que ele armazena. Usamos então a estrutura PARA que possui a variável contadora a ser usada como índice do vetor.

Imagine que eu irei guardar 5 pedaços de pizza no estômago.

```
var
        vetor:array(1..fim) of tipo;
        contador:integer;
inicio
        for variável:=inicio to fim do
                vetor(variavel);
inicio
```

```
var
estomago:array(1..5) of string;
        contador:integer;
inicio
        for variável:=inicio to fim do
                estomago(contador):='pizza'
```

EXEMPLO PRÁTICO

Preencher um vetor de 10 elementos com números inteiros. Ao final, percorrê-lo e mostrar quantos números do conjunto são maiores do que 25.

```
var
        total, contador:integer;
        vetor:array(1..10) of integer;
inicio
        para contador  de 1 até 10 faça
                leia (vetor(contador));
        para contador de 1 até 10 faça
                se vetor(contador)>25 então
                        total:= total+1;
        fim
        escreva total;
fim
```

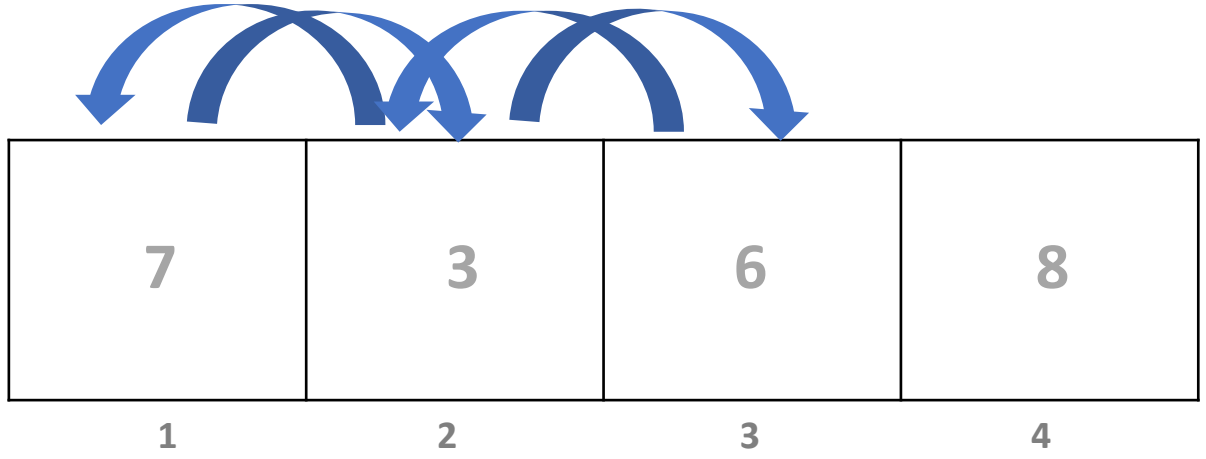
ORDENAÇÃO DE VETORES

Uma das áreas mais estudadas é a ordenação de conjuntos de dados. Dezenas de métodos são propostos, uns mais rápidos, outros mais lentos, dependendo do algoritmo. Um dos métodos mais simples é o **bubble sort** (literalmente, ordenação por bolha).

Bastante simples, nada mais faz do que comparar cada elemento do vetor com o próximo valor. Caso necessário, faz um troca de posição entre os valores para ordená-los. A cada troca a contagem é reiniciada, isto é, pega-se novamente o primeiro elemento e compara-se com os demais.

EXEMPLO PRÁTICO

Imagine o vetor com 4 elementos acima. Pegamos então o primeiro valor (7) e comparamos com o próximo (3). Como o segundo é menor, é necessário trocá-los de posição. Após a troca, recomeça-se o ciclo, então pegamos novamente o primeiro valor (agora o 3) e compararmos com o terceiro elemento. As trocas vão sendo efetuadas, até que todos os números seja comparados entre si



PESQUISA EM VETORES

Ao pesquisar um elemento, podemos fazê-lo com o vetor desordenado, o que leva a um desempenho menor, ou ordenado, o que traz rapidez à pesquisa. Para vetores desordenados, fazemos uma busca sequencial. Para conjuntos ordenados, utilizamos a busca binária.

Na busca sequencial, comparamos o elemento buscado com cada posição no vetor, isto consome muito tempo e processamento em vetores grandes.

Na busca binária, o vetor obrigatoriamente estará ordenado. A ideia é dividir o vetor sempre em 2 e procurar em apenas uma das metades. É eficiente, pois a cada novo corte, elimina-se metade dos dados.

MATRIZES

Outra forma de coleção é a matriz bidimensional, ou simplesmente matriz. Os dados são dispostos em linhas e colunas, como em uma planilha. O índice passa a ser composto por 2 números, uma para linha e outro para a coluna

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

DECLARAÇÃO E MANIPULAÇÃO

Na declaração da matriz, dentro do colchetes são informados o início e fim dos índices das linhas e colunas, separados por vírgula. A manipulação de matrizes é mais complexa e requer na maioria dos casos, uso de estruturas de repetição encadeada (geralmente usa-se a estrutura PARA / FOR).

Usando uma estrutura PARA encadeada, posicionamos a linha em 1 e então percorremos todas as colunas. Após isso, posicionamos a linha em 2 e percorremos novamente todas as colunas. O processo é repetido até acabarem as linhas da matriz.

SINTAXE

Usando uma estrutura PARA encadeada, posicionamos a linha em 1 e então percorremos todas as colunas. Após isso, posicionamos a linha em 2 e percorremos novamente todas as colunas. O processo é repetido até acabarem as linhas da matriz.

```
var
        matriz: array(1.. linhas, 1.. colunas) of integer;
        linha, coluna:integer;
inicio
        para linha  de 1 até 10 faça
                para coluna de 1 até 10 faça
                        read (matriz(linha, coluna))
        fim
fim
```

FUNÇÕES E PROCEDIMENTOS

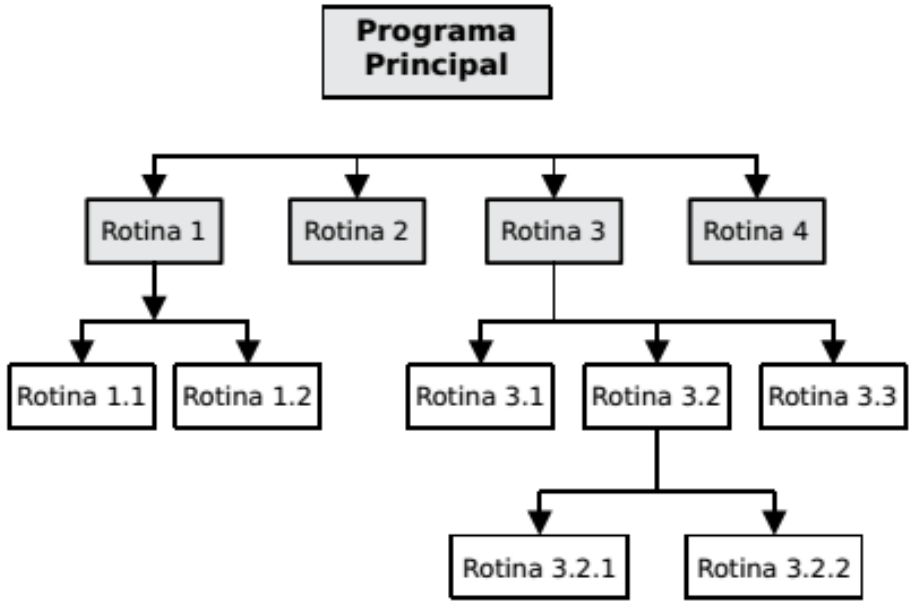
As funções e procedimentos são mini-programas, blocos de código que são executados separadamente em relação ao programa principal. Uma das grandes vantagens é a reutilização de código, isto é, pode-se executar várias vezes, em qualquer tempo, sem que seja necessitário redigitar o código.

Construir funções ou procedimentos é nada mais do que dividir o programa principal em códigos menores, criando subprogramas.

Com o código principal reduzido, basta “chamar” ou “invocar” o subprograma toda vez que desejar utilizá-lo.

ABORDAGEM TOP-DOWN

Também chamado de Refinamento Sucessivo, o método top-down, visa a organização do algoritmo, quebrando o código em porções menores para facilitar a resolução do problema. Não é um processo simples, pois requer experiência e bom entendimento em abstração (visão geral), lógica (construção do código principal) e análise (detalhamento das subrotinas)



ESCOPO GLOBAL E LOCAL

Funções e procedimentos podem conter variáveis próprias, de modo separado do programa principal. As variáveis, neste caso, são locais, isto é, funcionam apenas dentro da função ou procedimento em que estão declaradas.

Já as variáveis declaradas no programa principal são globais, ou seja, servem para todo algoritmo e podem ser usadas em qualquer ponto do código, inclusive dentro das subrotinas.

PROCEDIMENTOS / PROCEDURES

A principal característica de um procedimento é **não retornar valores**, isto é, não gera dados para o programa principal. Isto não impede de utilizar comandos para saída de dados como ESCREVA / WRITE.

O nome do procedimento segue a mesma regra para o nome das variáveis. A sintaxe de declaração é:

```
procedure nome (parâmetro:tipo)
```

Após a declaração, o restante é semelhante ao programa principal, com declaração de variáveis, processamento e saída.

FUNÇÕES / FUNCTIONS

As funções devem obrigatoriamente retornar um valor para o programa principal, através do comando RETORNE / RETURN, não necessitando utilizar comandos de saída. O nome da função segue a mesma regra para o nome das variáveis.

A sintaxe de declaração é

```
function nome (parâmetro:tipo) : retorno
```

Observe que temos que declarar um tipo de dados para o retorno. Após a declaração, o restante é semelhante ao programa principal, com declaração de variáveis, processamento e saída.

PARÂMETROS

Os subprogramas, geralmente necessitam de dados para trabalhar seu código interno, sendo eles vindos do programa principal. Esse dados são chamados de parâmetros, podendo ser locais/ou formais (de uso interno do subprograma) ou globais/reais, de uso do programa principal.

```
var
        x,y: integer;
procedure somar (a, b:integer)
inicio
        escreva (a+b);
fim
inicio
        leia x;
        leia y;
        somar (x, y);
fim
```

PARÂMETROS LOCAIS/FORMAIS!

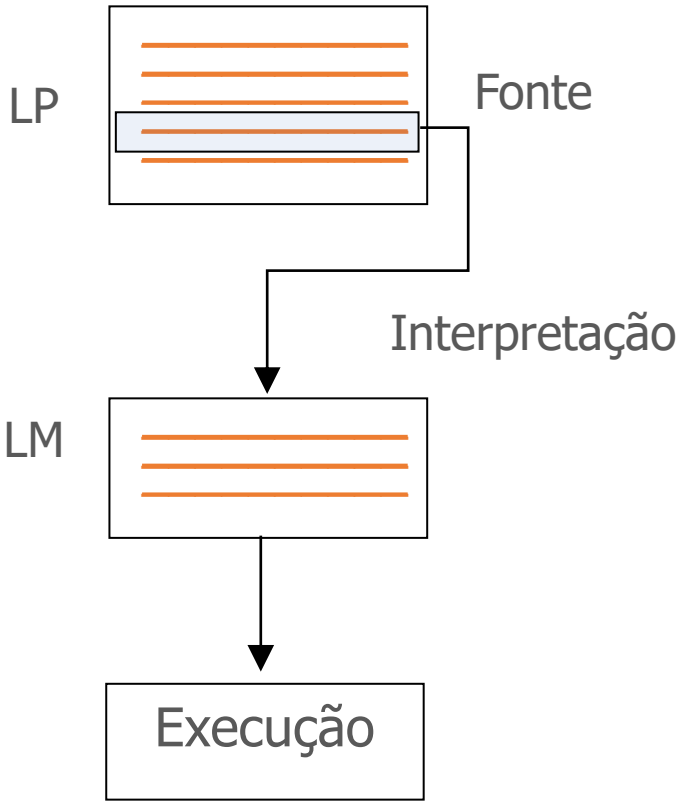
PARÂMETROS GLOBAIS/REAIS!

COMPILAR X INTERPRETAR

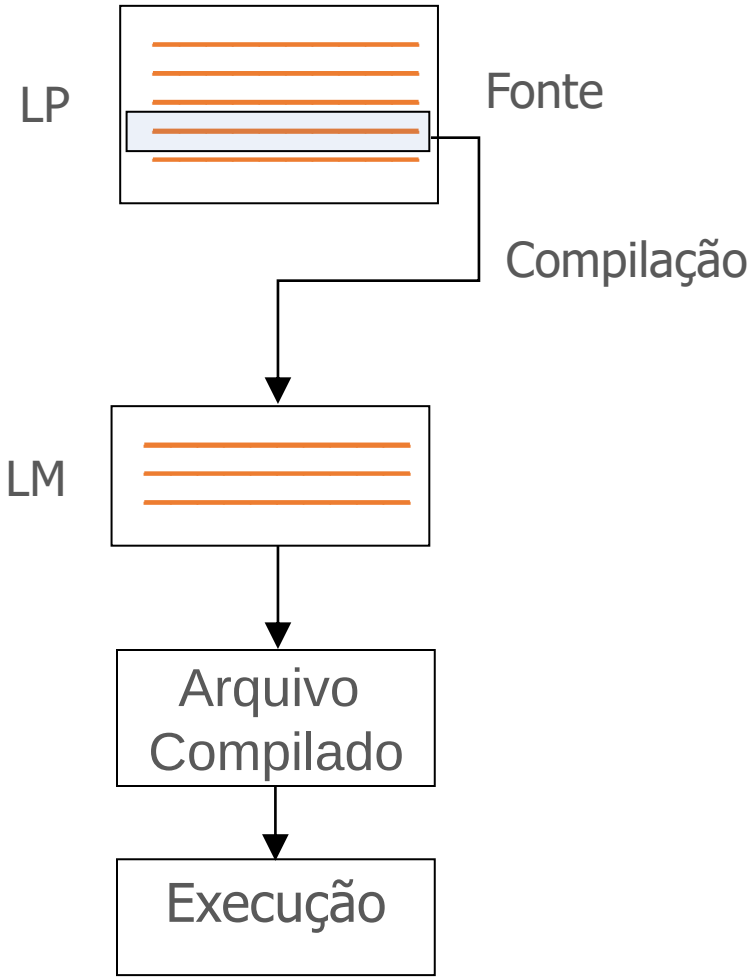
Em linguagens de alto nível, o código pode ser tanto compilado, quando interpretado.

Na interpretação, o código de alto nível é lido e executado ‘em tempo real’ por um programa externo. O interpretador, após verificar a consistência da sintaxe, converte o código de alto nível em linguagem de máquina para a execução.

Já na compilação, o código de alto nível é lido e com ele é gerado um novo arquivo, ilegível aos humanos. É este arquivo que será executado em linguagem de máquina.



Javascript, PHP, HTML, Python, etc.



Java, Delphi, Visual Basic, etc.

ESTRUTURA DE DADOS

É um ramo da Ciência da Computação que estuda os mecanismos de armazenamentos e organização de dados. Determinadas estruturas de dados influenciam de um modo geral no acesso à memória, capacidade e velocidade do processamento. Algumas estruturas são:

- Array
- Listas Ligadas, Duplamente Ligadas e Circulares
- Fila
- Pilha
- Árvore
- Árvore Binária

ARRAY

Um Array organiza dados como uma “coleção”, em posições sucessivas na memória, onde cada posição é identificada por um índice. A manipulação de um array é feita com 3 operações básicas: Inserir, Remover, Listar. A complexidade das operações varia em arrays ordenados ou não. Podemos encontrar dois tipos de Array.

- **Vetores** constituem *arrays* unidimensional, armazenando dados de forma linear;
- **Matrizes** podem ser definidas como “vetor de vetores”, de forma multidimensional (como uma grade);

10	**2**	**5**	**40**	**23**	**25**	**29**

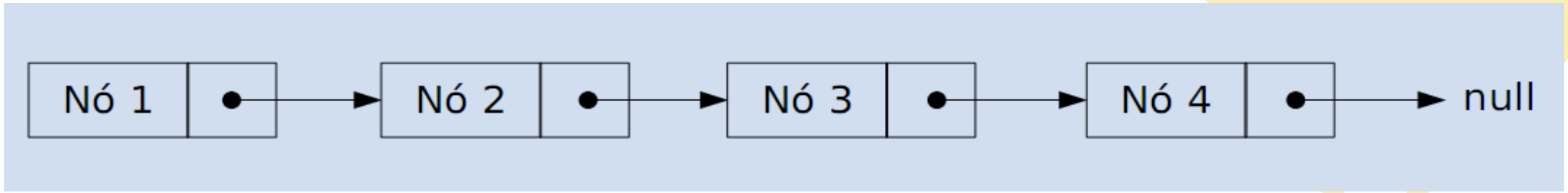
1	2	3
4	5	6
7	8	9

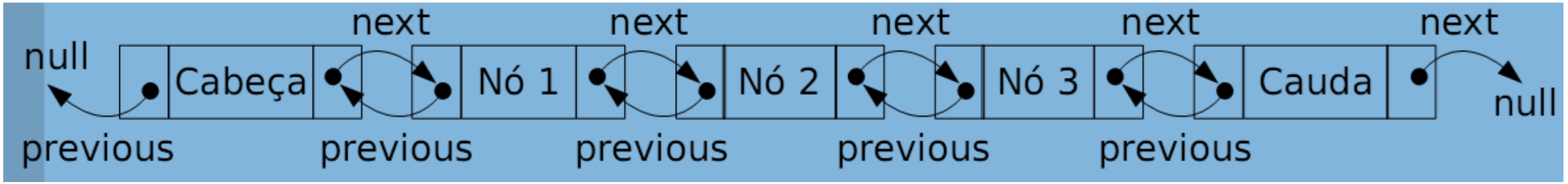
LISTAS LIGADAS, DUPLAMENTE LIGADAS E CIRCULARES

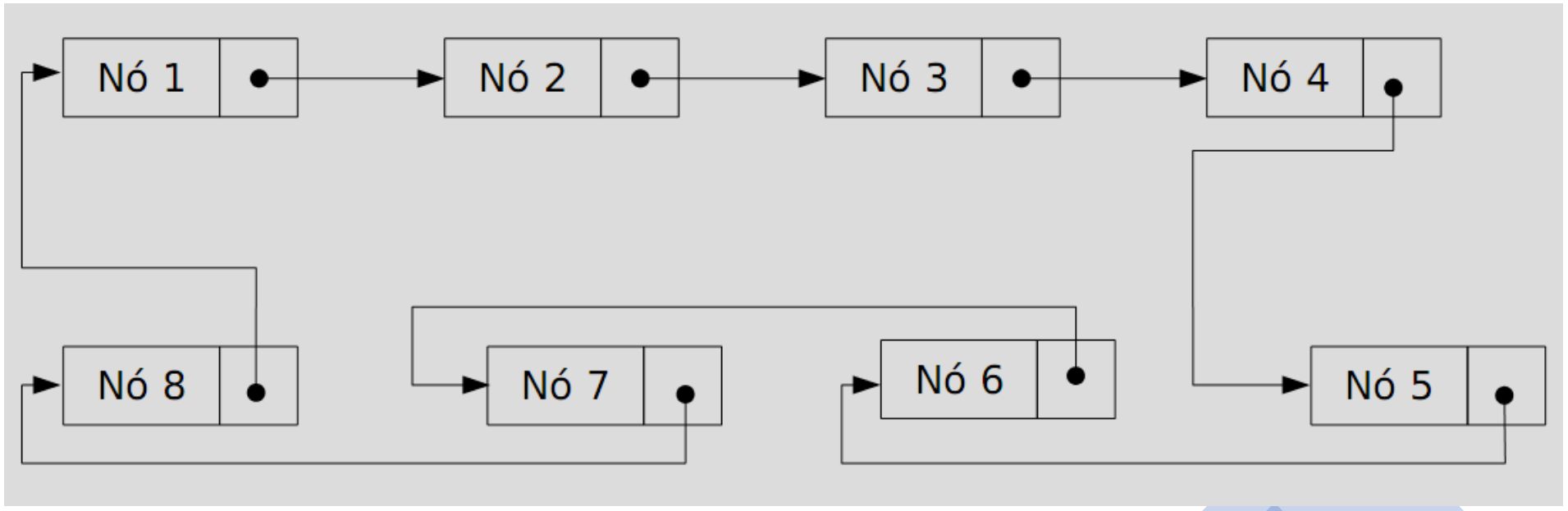
Uma **lista ligada** é uma coleção de objetos (*nós*) em ordem linear. A particularidade é que cada nó contém um referência para outro nó, indicando o próximo elemento da lista. O primeiro nó da lista é chamado de cabeça (*head*) e o último é chamado de cauda (*tail*). Uma lista ligada não possui limite de tamanho, nem índices.

Já uma **lista duplamente ligada** permite navegação nas duas direções, já que cada nó contém duas referências: uma para o próximo e outra para o nó anterior.

E nas **listas circulares**, não há cauda nem cabeça, sendo que o último nó da lista aponta para o primeiro. Para controle eficiente, é necessário um nó-referência.


Nó 1
Nó 2
Nó 3
Nó 4
null

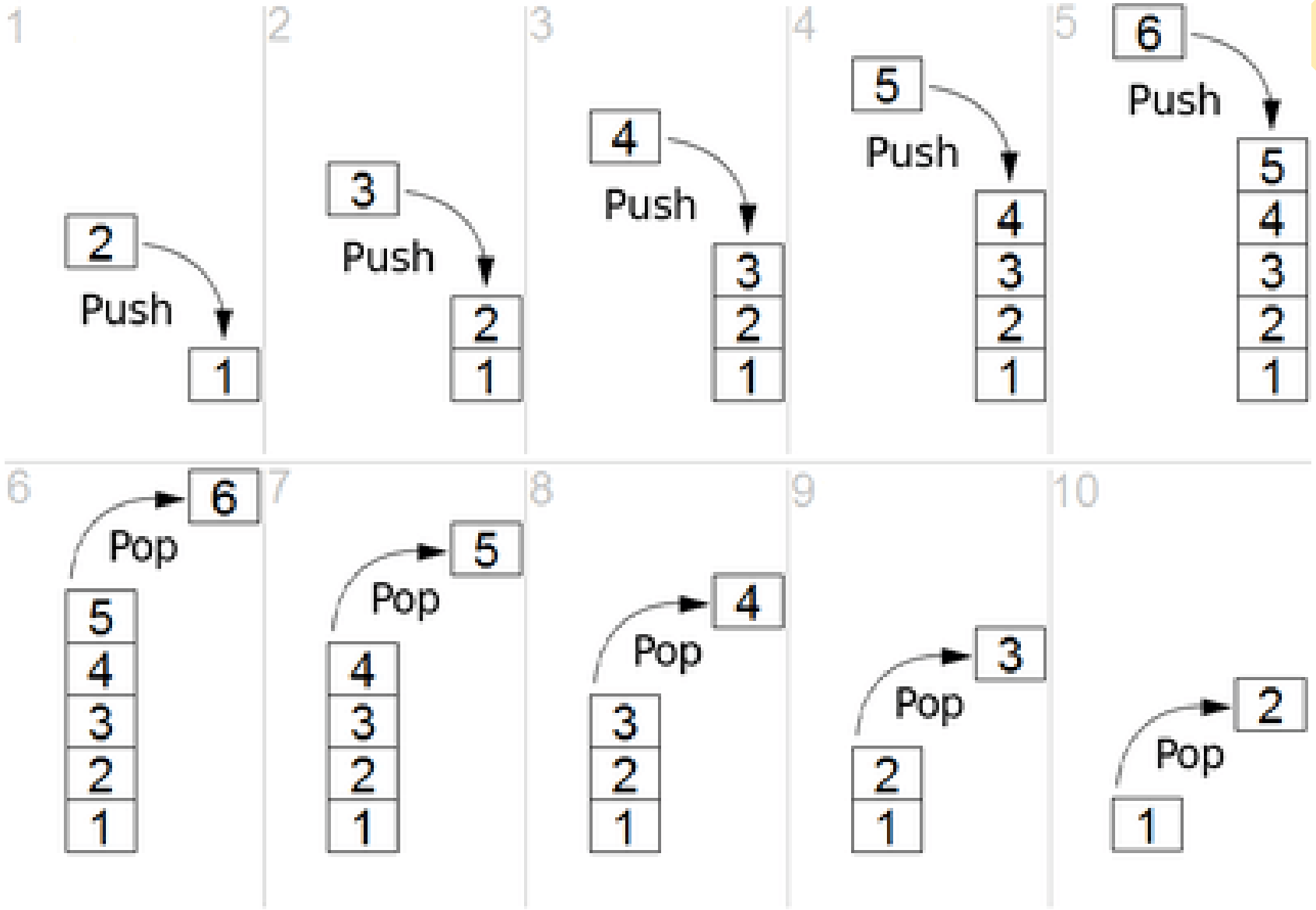

null
Cabeça
Nó 1
Nó 2
Nó 3
Cauda
null
next
next
next
next
next
previous
previous
previous
previous
previous


Nó 1
Nó 2
Nó 3
Nó 4
Nó 8
Nó 7
Nó 6
Nó 5

PILHA

A pilha é uma estrutura que faz abstração do objeto do mundo real (uma pilha de pratos, por exemplo). Seu método de acesso ou modo de trabalho é chamado de LIFO (*Last In, First Out)*, ou seja, adiciona-se ou remove-se sempre do topo.

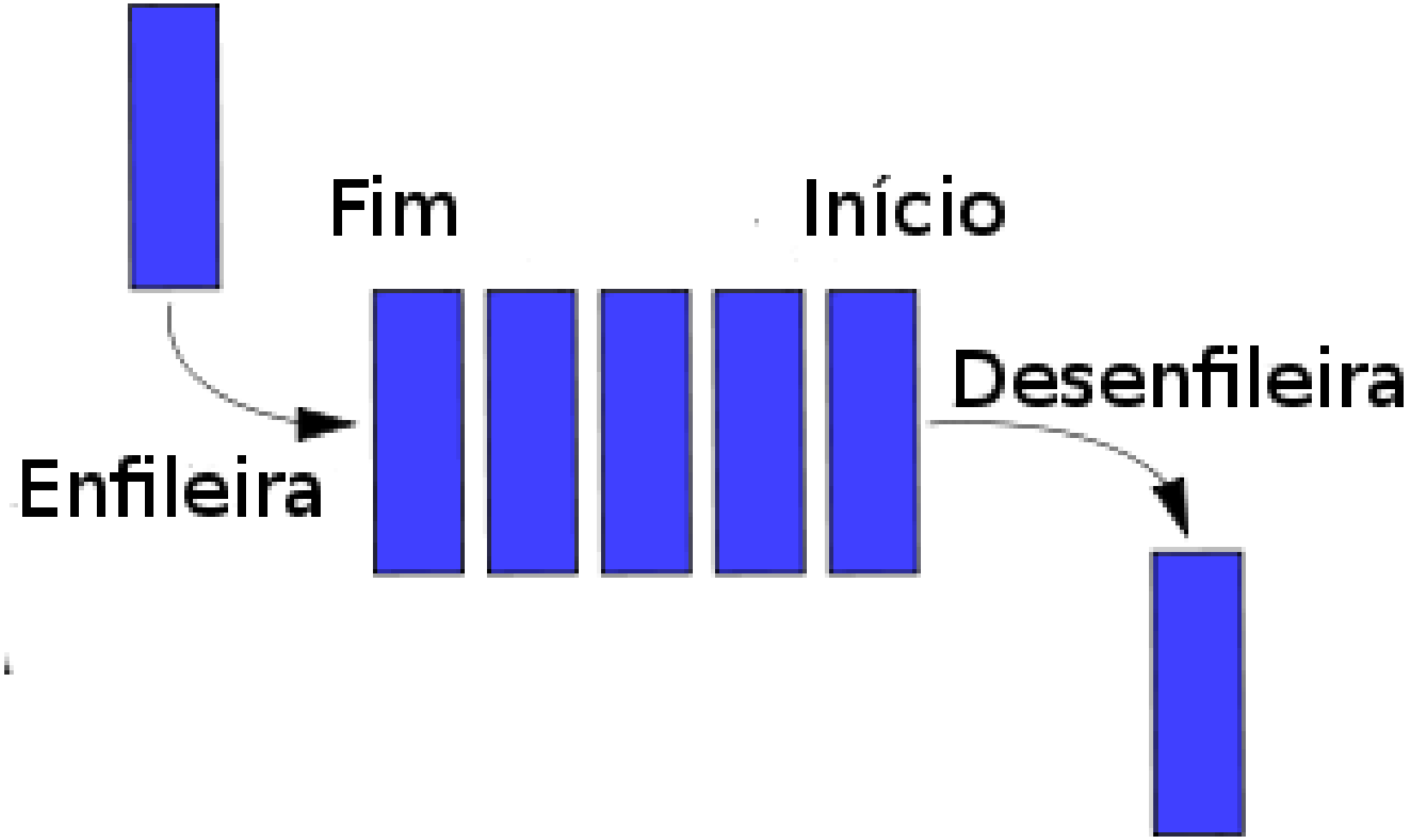
As operações básicas são referenciadas como Inserir (Empilhar), Remover (Desempilhar), Tamanho da Estrutura (Altura) e Selecionar Elemento (Topo)


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

FILA

A fila é uma estrutura que também faz abstração do objeto do mundo real (uma fila de pessoas, por exemplo). Seu método de acesso ou modo de trabalho é chamado de FIFO (*First In, First Out)*, ou seja, adiciona-se no final e remove-se no começo.

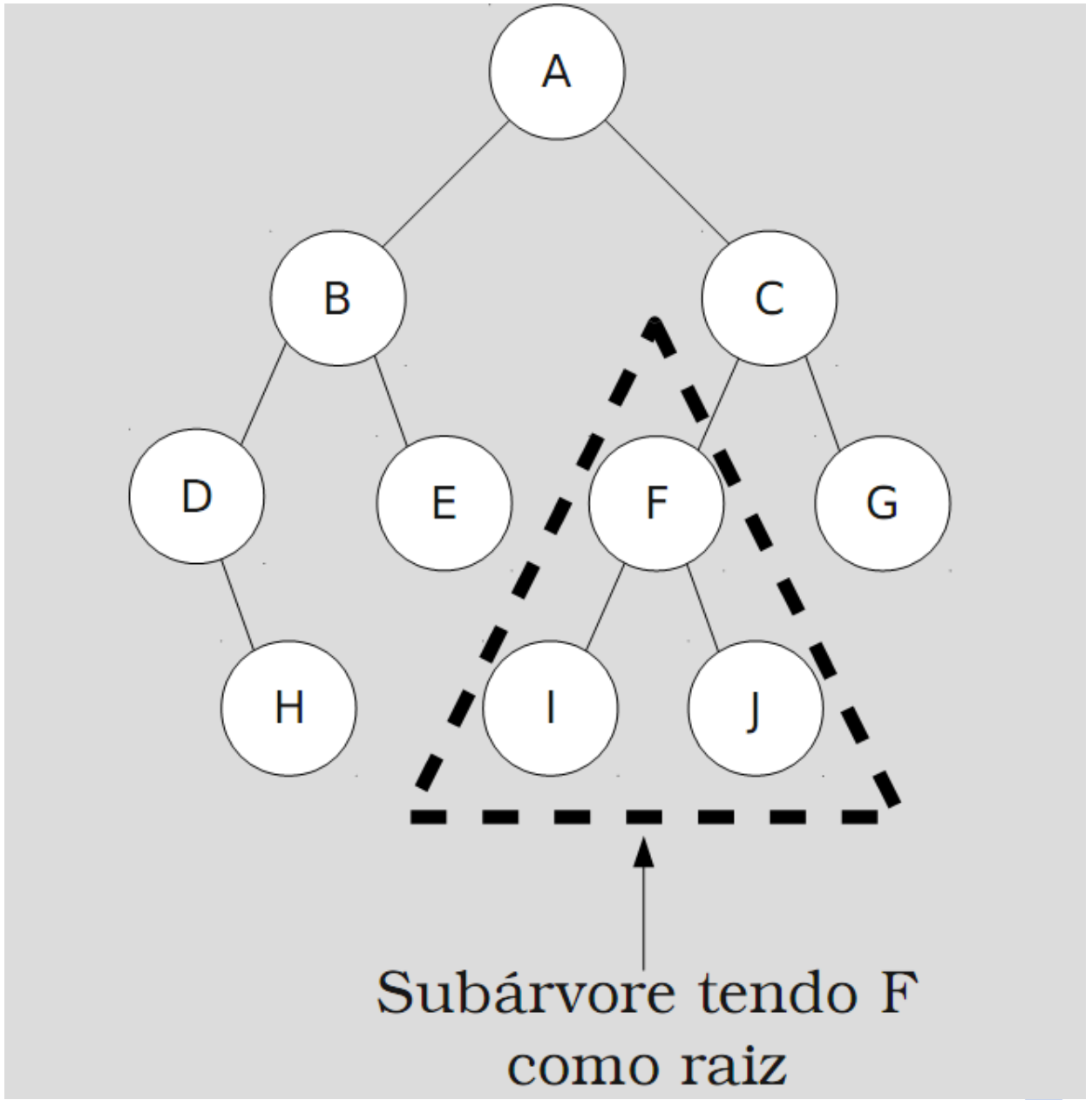
As operações básicas são referenciadas como Inserir (Enfileirar), Remover (Desenfileirar), Tamanho da Estrutura (Comprimento) e Selecionar Elemento (Próximo/Primeiro)


Fim
Início
Enfileira
Desenfileira

ÁRVORE

A árvore é basicamente uma coleção de nós em hierarquias, ao invés de sequenciais. Possui um nó superior, chamado de **Raiz,** que é o começo da árvore (o grafo é uma ‘árvore invertida’).

Todos os nós (exceto a raiz) possui um nó **Pai** e um nó sem filhos é chamado de **Folha.** Pode-se também formar subárvores com os filhos.


A
B
C
D
E
F
G
H
I
J
Subárvore tendo F
como raiz

ÁRVORE BINÁRIA

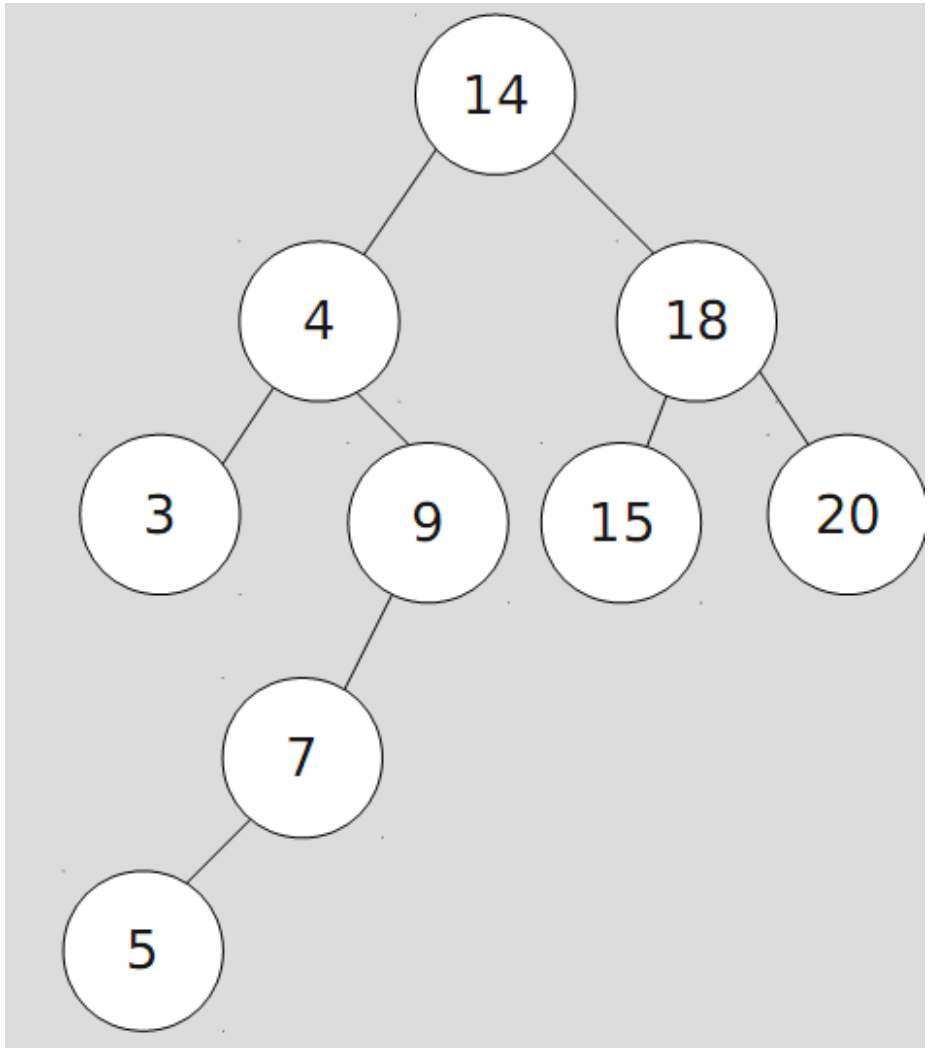
É uma estrutura de árvore, onde os pais tem no máximo 2 filhos, sendo útil em algoritmos com desvios no caminho. Por convenção, os filhos com valores maiores que o pai são armazenados **à direita** (partindo da raiz) e os filhos com valores menores que o pai são armazenados **à esquerda** (partindo da raiz).

A árvore também pode não conter elemento algum, sendo uma **árvore vazia.** A principal utilização de árvores binárias são as **árvores binárias de busca**.

ÁRVORE BINÁRIA

Vamos a um exemplo de aplicação em árvore binária. Vamos armazenar os seguintes valores: 14, 18, 4, 9, 7, 15, 3, 4, 20, 9, 5;

ÁRVORE BINÁRIA



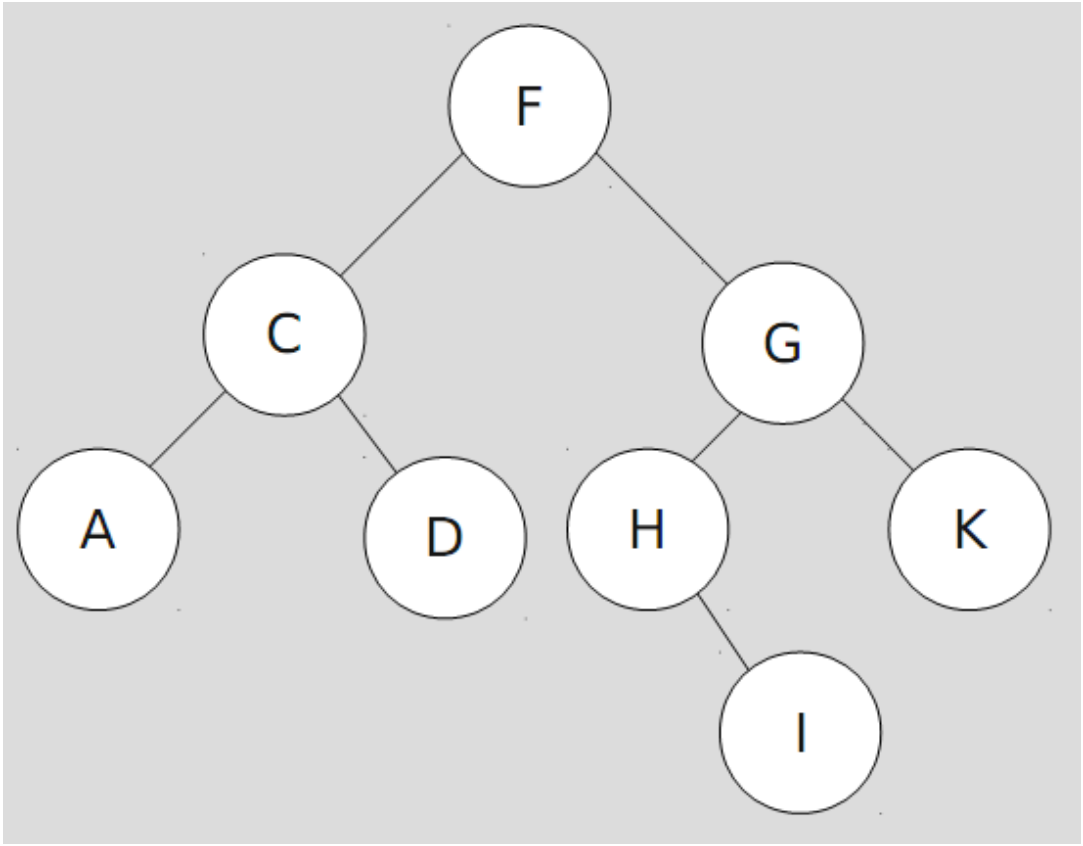
Vamos a um exemplo de aplicação em árvore binária. Vamos armazenar os seguintes valores: 14, 18, 4, 9, 7, 15, 3, 4, 20, 9, 5;

Agora para pesquisas, podemos fazer de 3 formas diferentes:

- **PRÉ-ORDEM** : RAIZ – ESQUERDA - DIREITA
- **ORDEM**: ESQUERDA – RAIZ – DIREITA
- **PÓS- ORDEM**: ESQUERDA – DIREITA - RAIZ

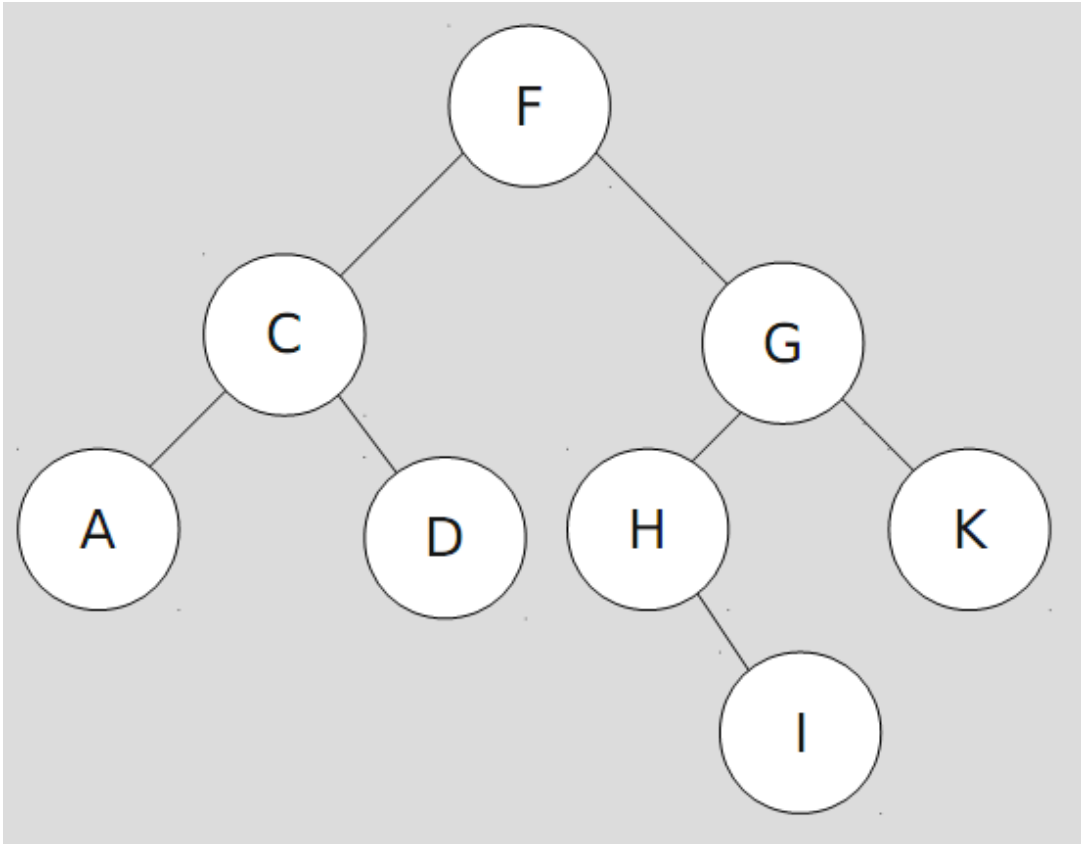
ÁRVORE BINÁRIA

Considerando a árvore ao lado, como ficariam os resultados em pesquisas de ordem, pré-ordem e pós-ordem?



ÁRVORE BINÁRIA

Considerando a árvore ao lado, como ficariam os resultados em pesquisas de ordem, pré-ordem e pós-ordem?



- **PRÉ-ORDEM**: F-C-A-D-G-H-I-K
- **ORDEM**: A-C-D-F-H-I-G-K
- **PÓS- ORDEM**: A-D-C-I-H-K-G-F

BANCO DE DADOS

ORGANIZAÇÃO E ROBUSTEZ

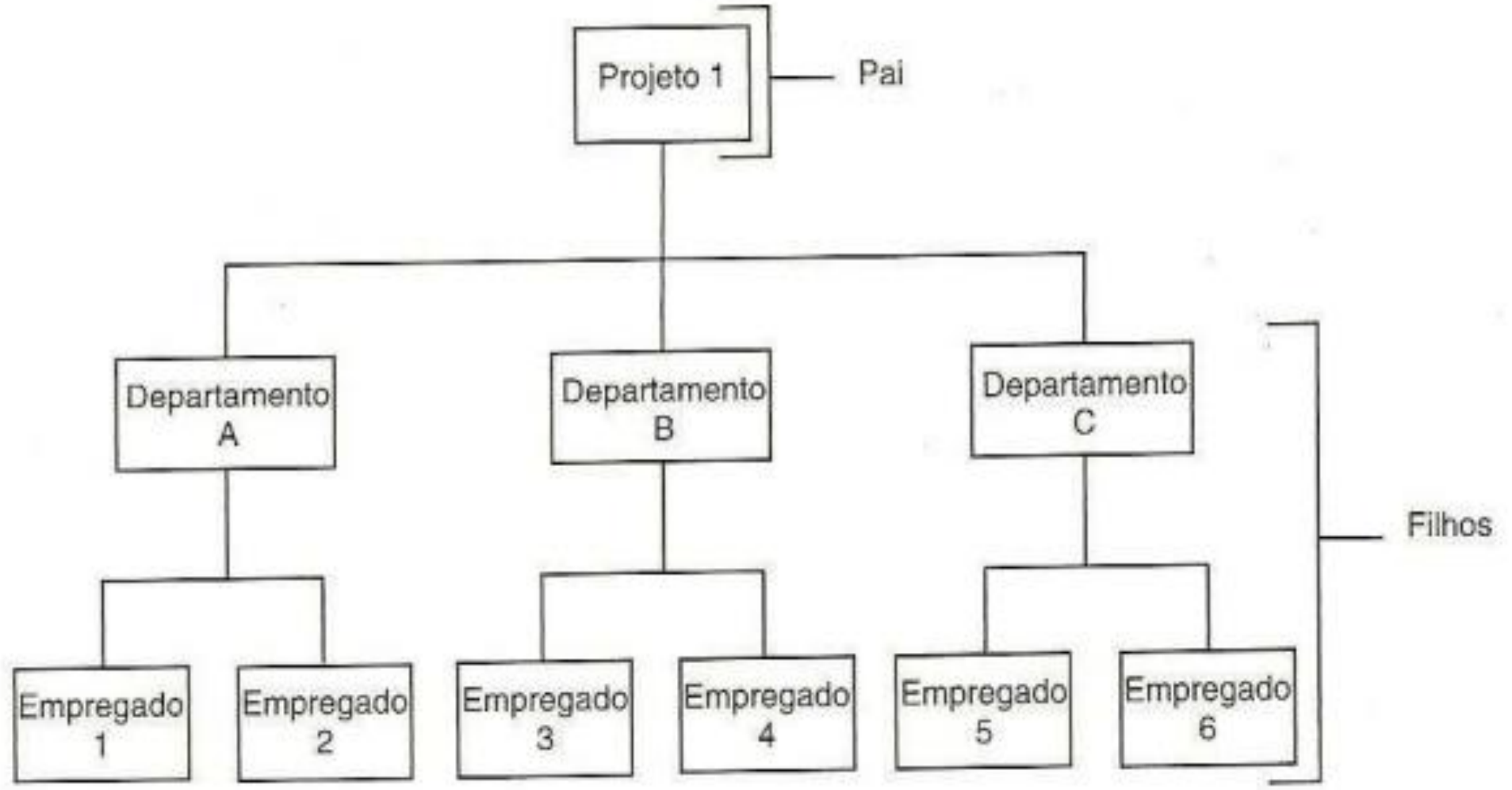
Os bancos de dados constituem uma camada importante na organização de arquivos, pois podem armazenar facilmente milhões de registros. Um banco de dados é composto essencialmente de:

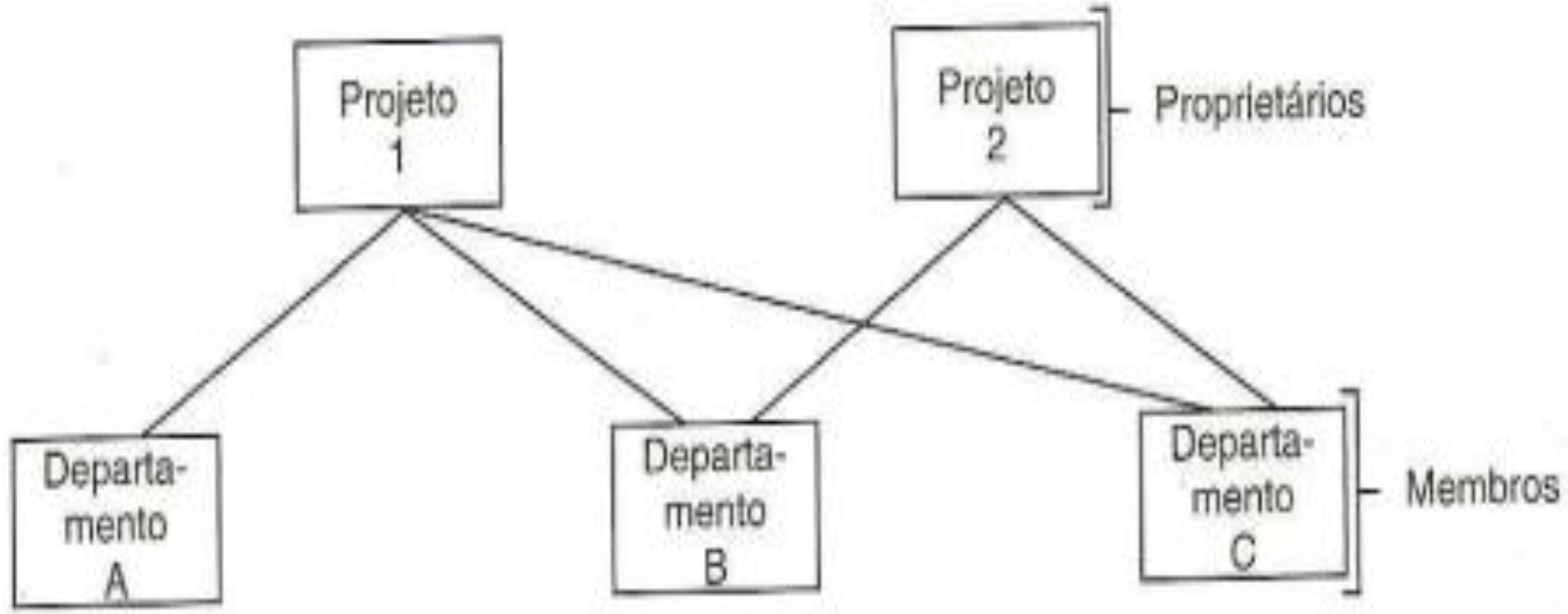
- Uma base de dados
- Um sistema gerenciador de banco de dados (SGBD)
- Linguagem de exploração
- Programas voltados a necessidades objetivas

Além disso, podemos ainda considerar o hardware que irá abrigar toda a estrutura e o pessoal envolvido (corpo técnico e usuários).

MODELOS DE BANCO DE DADOS

- **Hierárquico ou em árvore**: A informação é organizada em níveis. Os registros “superiores” podem ter um, vários ou nenhum “filho”. Os registros inferiores devem ter apenas um “pai”. O acesso é realizado de nó em nó;
- **Em rede**: Os registros são proprietários e são acessados por uma chave identificadora. Pode-se então mover-se sequencialmente ou diretamente aos registros membros;
- **Relacional**: Os registros são individualizados e ligados por meio de ponteiros identificadores.


Projeto 1
Pai
Departamento A
Departamento B
Departamento C
Filhos
Empregado 1
Empregado 2
Empregado 3
Empregado 4
Empregado 5
Empregado 6


Projeto 1
Projeto 2
Proprietários
Departa-mento A
Departa-mento B
Departa-mento C
Membros

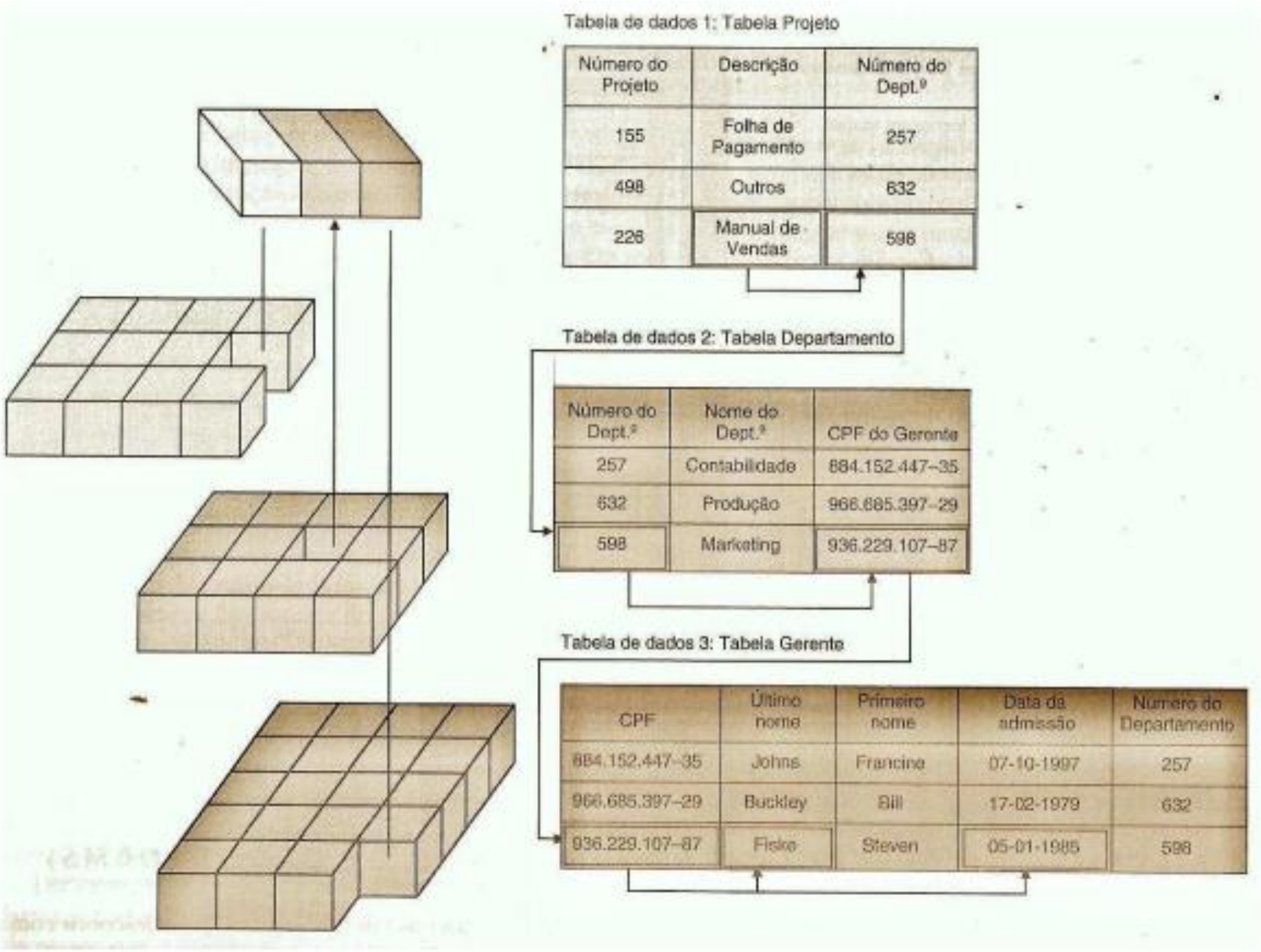


Tabela de dados 1: Tabela Projeto

Número do Projeto	Descrição	Número do Dept.º
155	Folha de Pagamento	257
498	Outros	632
228	Manual de Vendas	598

Tabela de dados 2: Tabela Departamento

Número do Dept.º	Nome do Dept.º	CPF do Gerente
257	Contabilidade	884.152.447–35
632	Produção	966.685.397–29
598	Marketing	936.229.107–87

Tabela de dados 3: Tabela Gerente

CPF	Último nome	Primeiro nome	Data de admissão	Número do Departamento
884.152.447–35	Johns	Francine	07-10-1997	257
966.685.397–29	Buckley	Bill	17-02-1979	632
936.229.107–87	Fiske	Steven	05-01-1985	598

CHAVE IDENTIFICADORA DE REGISTRO

- Para facilitar o acesso aos registros armazenados, são definidas algumas informações-chaves que identificam aquele registro.
- **Chave Primária**: identifica um registro de forma única, ou seja, não podem existir registros com o mesmo valor de chave primária no mesmo arquivo. Os SGBDs possuem consistência para não permitir que sejam inseridos 2 registros com a mesma chave primária.
- **Chave Secundária**: identifica o registro, mas não de forma única, podendo haver repetições no mesmo arquivo. Normalmente é associada com uma chave primária de outro arquivo.

VANTAGENS

Bancos de dados apresentam algumas vantagens em relação ao armazenamento e organização de dados, comparado aos dispositivos tradicionais.

- Independência de dados;
- Controle de redundância
- Garantia de integridade
- Privacidade dos dados
- Facilidade de criação de novas aplicações
- Segurança de dados
- Otimização da utilização do espaço para armazenamento
- Controle automático de relacionamento entre registros.

NÃO É LINGUAGEM DE PROGRAMAÇÃO!

SQL

DDL

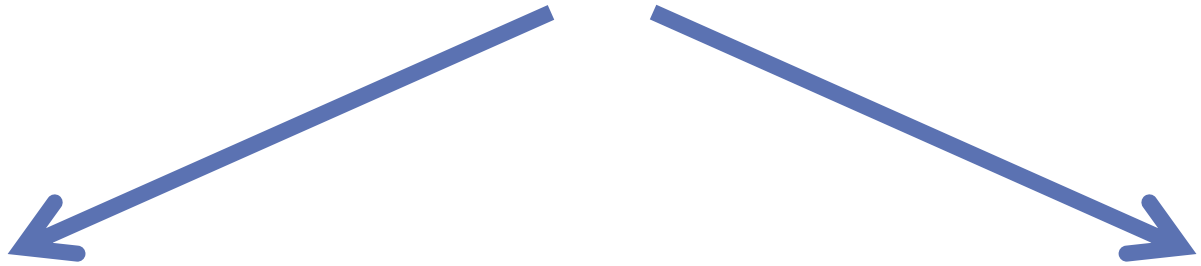
DML

DQL

DCL

DTL

TABELA



CAMPOS

REGISTROS

Todo registro deve possuir um índice ("campo principal")

SISTEMA GERENCIADOR DE BANCO DE DADOS

TIPOS DE DADOS - MySQL

- **INTEGER**: Números inteiros;
- **DECIMAL (p,d)**: Números reais, com uma precisão e o número de casas decimais;
- **CHAR**: Caracteres de tamanho fixo. Máximo de 255;
- **VARCHAR**: Caracteres de tamanho variável. Máximo de 255;
- **TEXT**: Caracteres de tamanho variável. Máximo de 65535;
- **DATE**: Data americana (aaaa-mm-dd);
- **DATETIME**: Data/hora (hh:mm:ss);

Os campos ainda podem conter atributos como UNSIGNED (sem sinal), BINARY (case sensitive) e NOTNULL (proibição de nulo)

TIPOS DE DADOS - PostgreSQL

- **INT, INT4, INT8**: Números inteiros;
- **NUMERIC (p,d)**: Números reais, com uma precisão e o número de casas decimais;
- **CHAR**: Caracteres de tamanho fixo. Máximo de 255;
- **VARCHAR**: Caracteres de tamanho variável. Máximo de 255;
- **TEXT**: Caracteres de tamanho variável. Máximo de 65535;
- **DATE**: Data americana (aaaa-mm-dd);
- **CIDR, INET**: Armazena um endereço IP no formato x.x.x.x/y;
- **MACADDR**: Armazena um endereço físico de um host (MAC);
- **POLYGON**, **BOX**, **CIRCLE**: Utilizado para armazenar valores geométricos de formas em formato de coordenada x, y. (ex: circle é o x,y do seu centro + o seu raio);
- **SERIAL**: Campo para autoincremento;

TERMINAL

O console ou shell é a interface mais simples para acesso e manipulação das bases de dados, aceitando os comandos normalmente. O principal comando no MySQL via console é o de login. O parâmetro -u indica o usuário e o parâmetro -p faz com que a senha seja requisitada após o comando. Pode-se indicar também o ip do servidor com o parâmetro -h

mysql -u root –p

Já no PostgreSQL, o parâmetro -c instrui o terminal a conectar-se a base de dados, passando o nome do banco e o usuário. Também é possível conectar-se diretamente a um banco remoto passando o endereço do host, usuário e banco.

postgres -c psql postgres
psql -h 127.0.0.1 -U usuario -d nomedobanco

DDL - MEXENDO NA ESTRUTURA

Os comandos DDL servem resumidamente para “montar” o banco de dados, criando e alterando bases e tabelas, ou até mesmo excluindo-as. São comandos de definição da estrutura do SGBD, considerados as etapas iniciais na implementação de um banco de dados.

- CRIAR UM BD: ***CREATE DATEBASE*** *nomedobanco*
- SELECIONAR UM BD: ***USE*** *nomedobanco* (MySQL) ***\c*** *nomedobanco* (PostgreSQL)
- MOSTRAR BD EXISTENTES: ***SHOW DATABASES*** (MySQL) ***\d DATABASES*** (PostgreSQL)
- APAGAR UM BD: ***DROP DATABASE*** *nomedobanco*
- APAGAR UMA TABELA: ***DROP TABLE***
- MOSTRAR TABELAS EXISTENTES: ***SHOW TABLES*** (MySQL) ***\d TABLES*** (PostgreSQL)
- MOSTRAR ESTRUTURA DA TABELA: ***DESC*** *tabela* (MySQL) ***\d*** *tabela* (PostgreSQL)
- ESVAZIAR TABELA: ***TRUNCATE*** *tabela*

COMANDOS DDL

- **CRIAR UMA TABELA**: Para criar uma tabela dentro de um banco de dados (selecionado anteriormente), utilizamos o comando **CREATE TABLE**. Possui várias **cláusulas e restrições** (***constraints***)**.** *A sintaxe é* ***CREATE TABLE nomedatabela (campos);***

A sintaxe dos campos é ***nome tipo(tamanho) atributos/cláusulas/restrições.*** *As opções são:*

- ✓ **DEFAULT:** insere um valor padrão, caso não seja fornecido outro na gravação.
- ✓ **UNIQUE:** define que os valores desse campo não podem repetir-se.
- ✓ **NOT NULL:** proíbe que o campo fique vazio.
- ✓ **PRIMARY KEY:** cria uma chave primária simples ou composta
- ✓ **FOREIGN KEY:** cria uma chave estrangeira, sempre referenciando uma chave primária
- ✓ **AUTO_INCREMENT:** define um campo sequencial para cada registro, começando em 1

idade integer(3) not null primary key default 0

codigo integer(3) foreign key(codigo) references TABELA (chave_primária)

COMANDOS DDL

- **ALTERAR UMA TABELA**: para alterar qualquer aspecto de uma tabela. A sintaxe é ***ALTER TABLE nomedatabela comandos.*** Os possíveis comandos são:

✓ **ADD:** adicionar um campo em uma posição específica ou por último. Por exemplo ***ADD endereco varchar(90) not null AFTER campo***
✓ **CHANGE COLUMN:** alterar características de um campo. Por exemplo ***CHANGE COLUMN telefone tel varchar(14) not null***
✓ **DROP COLUMN:** apagar um campo específico. Por exemplo ***DROP COLUMN endereço***
✓ **RENAME TO:** renomear uma tabela. Por exemplo ***RENAME TO clientes***
✓ **ADD CONSTRAINT:** usado comumente para adicionar chaves. Por exemplo:

ADD CONSTRAINT 'minhachave' PRIMARY KEY(codigo)
ADD CONSTRAINT 'apelido' FOREIGN KEY (campos) REFERENCES TABELA(PK)

COMANDOS DDL

No PostgreSQL, para criarmos um campo sequencial, semelhante ao AUTO_INCREMENT do MySQL, podemos utilizar um campo do tipo SERIAL ou utilizamos o comando **CREATE SEQUENCE**

create sequence cliente_id_seq;

Após criar a sequence, podemos manipulá-la com alguns comandos, juntamente com o comando SELECT da DQL;

- **NEXTVAL:** pega o próximo valor da sequence: ***select nextval (cliente_id_seq);***
- **CURRVAL:** pega o valor atual da sequence: ***select currval (cliente_id_seq);***
- **SETTVAL:** define um valor específico para a sequence: ***select setval (cliente_id_seq, 5);***

DML - MANIPULANDO REGISTROS

Após “construir” o banco de dados, os comandos DML permitem manipular diretamente os registros, criando-os, alterando-os ou excluindo-os. Apenas 3 comandos compõe a DML; **insert into**, **update**, e **delete**. Ambos possuem cláusulas que modificam seu funcionamento.

- **INSERT INTO**: É o comando padrão para adicionar um registro a uma tabela específica.

INSERT INTO tabela (campos) VALUES (valores);

INSERT INTO clientes (codigo, nome) VALUES (1, ‘Telésforo‘);

Pode-se omitir quais campos receberão valores, neste caso considera-se todos os campos da tabela, os valores devem ser passados campo-a-campo, na ordem em que foram criados.

INSERT INTO clientes VALUES (1, ‘Telésforo‘);

COMANDOS DML

- **UPDATE**: Utilizado para alterar os valores de um ou mais campos da tabela.

UPDATE tabela SET campo=valor;

UPDATE times SET pontos=0;

Note que todos os times receberão 0 pontos. Para alterar apenas um ou mais campos, exceto todos, utilizamos a **cláusula WHERE.**

UPDATE times SET pontos=0 WHERE nome='Corinthians';

UPDATE times SET pontos=0 WHERE estado='SP';

COMANDOS DML

- **DELETE**: Utilizado para apagar um registro inteiro da tabela

DELETE FROM tabela;

DELETE FROM times;

Com o comando acima, apagaremos todos os registros, isto é, todos os times. Para apagar um ou mais registros específicos, utilize também a cláusula WHERE.

DELETE FROM times WHERE nome='Corinthians';

DELETE FROM times WHERE estado='SP';

DQL - CONSULTANDO O BANCO.

A DQL permite realizar consultas a fim de trazer resultados, com base nos registros armazenados no banco. Possui somente 1 comando, **select**, com diversas cláusulas e funções. É o comando padrão para “procurar” e trazer registros advindos do banco de dados.

```
SELECT campos FROM tabela;
SELECT nome, idade FROM clientes;
```

Quando desejarmos trazer todos os campos, utilizamos * (asterisco). Também aceita a cláusula WHERE para evitar trazer registros indesejados.

```
SELECT * FROM clientes WHERE idade>18;
```

COMANDO SELECT

- **Cláusula AND**: combina logicamente duas cláusulas WHERE.

SELECT * FROM clientes WHERE cidade =‘Cunha’ AND salario>1000;

- **Cláusula OR**: faz uma disjunção lógica entre duas cláusulas WHERE.

SELECT * FROM clientes WHERE cidade =‘Cunha’ OR salario>1000;

- **Cláusula IN**: seleciona os registros onde o conteúdo de um campo esteja dentro de uma lista definida (funciona como um OR encurtado).

SELECT * FROM clientes WHERE cidade IN (‘Cunha’, ‘Lorena’);

COMANDO SELECT

- **Cláusula ORDER BY**: traz registros ordenados, de forma crescente (ASC) ou decrescente (DESC), com base em um ou mais campos. Por padrão, a ordem é crescente

```
SELECT * FROM clientes ORDER BY nome ASC;
SELECT * FROM clientes ORDER BY nome ASC, idade DESC;
```

- **Função CONCAT ()**: Unir texto puro a uma variável dentro do mesmo SELECT.

```
SELECT CONCAT('O nome do cliente é ', nome) FROM clientes WHERE id=3;
```

- **Função COUNT()**: Retorna o número de registros selecionados.

```
SELECT COUNT(*) FROM clientes;
SELECT COUNT(*) FROM clientes WHERE idade>18;
```

COMANDO SELECT

- **Função SUM()**: Soma os registros selecionados, com base em um campo necessariamente numérico.

SELECT SUM (idade) FROM clientes;

- **Função AVG()**: Retorna a média dos registros selecionados, com base em um campo necessariamente numérico.

SELECT AVG(idade) FROM clientes;

- **Função MAX()**: Encontra o maior valor de um campo entre os registros selecionados.

SELECT MAX (idade) FROM clientes;

- **Função MIN()**: Encontra o menor valor de um campo entre os registros selecionados.

SELECT MIN (idade) FROM clientes;

COMANDO SELECT

- **Cláusula GROUP BY**: traz registros agrupados por determinado campo.

SELECT * FROM clientes GROUP BY cidade; //mostra todos os clientes agrupados por cidade

SELECT COUNT(*), estado FROM times GROUP BY estado; //mostra a quantidade de times, por estado.

- **Cláusula DISTINCT**: traz registros sem duplicidade nos resultados.

SELECT DISTINCT campo FROM tabela;

- **Cláusula UNION [ALL]**: combina duas consultas no mesmo resultado (‘dois SELECT em um só’). Faz implicitamente o comando DISTINCT após o resultado. Para funcionar, o número, a ordem e o tipo de dados das colunas devem ser compatíveis.

SELECT campo1, campo2 FROM tabela UNION;

SELECT campo1, campo2 FROM tabela;

COMANDO SELECT

-**Cláusula JOIN**: traz registros combinados, utilizando o relacionamento (chaves primária e estrangeira) entre as tabelas. Podemos fazer INNER JOIN ou OUTER JOIN.

```
SELECT campo FROM tabela1;
INNER JOIN tabela2
ON tabela1.chave = tabela2.chave;
```

No **INNER JOIN**, os dados são consultados observando as correspondências entre as colunas, isto é, os resultados que aparecem em ambas.

```
SELECT * FROM Livros;
INNER JOIN Autores
ON Livros.id_autor = Autores.id_autor;
```

INNER JOIN SQL

O resultado da cláusula anterior traz todos os registros das duas tabelas, onde houver a correspondência entre os resultados, isto é, todos os livros que possuem o autor cadastrado.

id_livro	nome	preco	id_autor	id_autor	nome
100	Poeira em alto mar	69.90	1	1	Telésforo da Silva
101	Incêndio na caixa d'agua	54.90	2	2	Fridundino de Oliveira
102	Meu tio é filho único	29.90	3	3	Holofontina Arantes
103	A vovó virgem	39.90	3	3	Holofontina Arantes

OUTER JOIN SQL

O **OUTER JOIN** retorna registros, mesmo se não houver nenhuma correspondência entre as tabelas (como um SELECT comum). Se divide em LEFT JOIN, RIGHT e FULL JOIN.

```
SELECT campo FROM tabela1 LEFT JOIN tabela2
ON tabela1.chave = tabela2.chave;
```

```
SELECT campo FROM tabela1 RIGHT JOIN tabela2
ON tabela1.chave = tabela2.chave;
```

```
SELECT campo FROM tabela1 FULLL JOIN tabela2
ON tabela1.chave = tabela2.chave;
```

LEFT JOIN SQL

Traz todos os resultados da primeira tabela, mesmo que não haja correspondência na segunda tabela.

id_livro	nome	preco	id_autor	id_autor	nome
100	Poeira em alto mar	69.90	1	1	Telésforo da Silva
101	Incêndio na caixa d'agua	54.90	2	2	Fridundino de Oliveira
102	Meu tio é filho único	29.90	3	3	Holofontina Arantes
103	A vovó virgem	39.90	3	3	Holofontina Arantes
104	Os marcianos de Marte	99.90	NULL	NULL	NULL
105	Tsunami na Amazônia	15.90	NULL	NULL	NULL

RIGHT JOIN SQL

Traz todos os resultados da segunda tabela, mesmo que não haja correspondência na primeira tabela.

id_livro	nome	preco	id_autor	id_autor	nome
100	Poeira em alto mar	69.90	1	1	Telésforo da Silva
101	Incêndio na caixa d'agua	54.90	2	2	Fridundino de Oliveira
102	Meu tio é filho único	29.90	3	3	Holofontina Arantes
103	A vovó virgem	39.90	3	3	Holofontina Arantes
NULL	NULL	NULL	NULL	4	Carabino Rodrigues
NULL	NULL	NULL	NULL	5	Aroeiro Ramos

FULL JOIN SQL

Traz todos os resultados da primeira e da segunda tabela, como um SELECT *, porém mostrando as correspondência.

id_livro	nome	preco	id_autor	id_autor	nome
100	Poeira em alto mar	69.90	1	1	Telésforo da Silva
101	Incêndio na caixa d'agua	54.90	2	2	Fridundino de Oliveira
102	Meu tio é filho único	29.90	3	3	Holofontina Arantes
103	A vovó virgem	39.90	3	3	Holofontina Arantes
104	Os marcianos de Marte	99.90	NULL	NULL	NULL
105	Tsunami na Amazônia	15.90	NULL	NULL	NULL
NULL	NULL	NULL	NULL	4	Carabino Rodrigues
NULL	NULL	NULL	NULL	5	Aroeiro Ramos

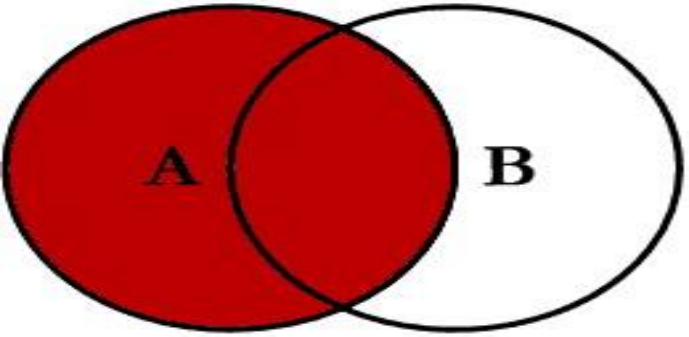
COMANDO SELECT

No **OUTER JOIN,** também é possível excluir as correspondências, ou seja, mostrando somente os resultados que não possuem ligação na outra tabela, utilizando a cláusula **WHERE.** Vamos ver o exemplo com o LEFT JOIN

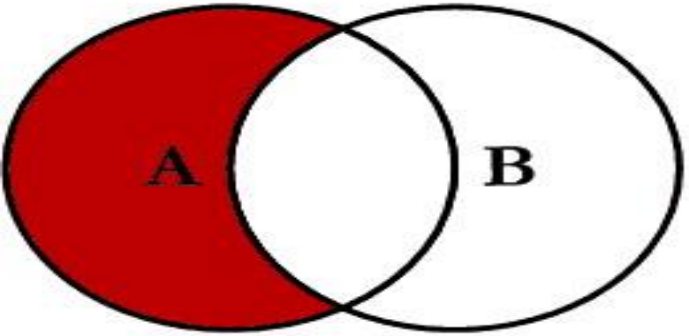
```
SELECT campo FROM tabela1 LEFT JOIN tabela2
ON tabela1.chave = tabela2.chave
WHERE tabela2.chave IS NULL;
```

```
SELECT * FROM Livros LEFT JOIN Autores
ON Livros.id_autor = Autores.id_autor
WHERE Autores.id_autor IS NULL;
```

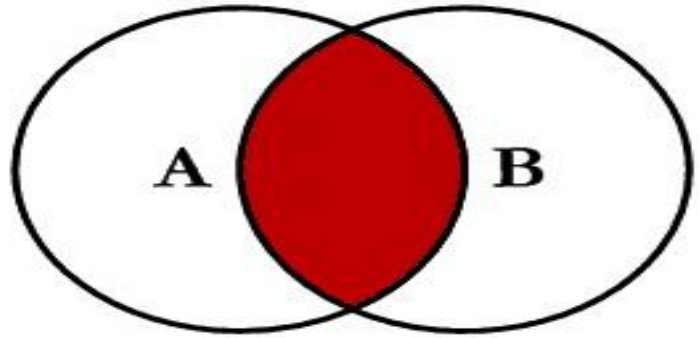
SQL JOINS



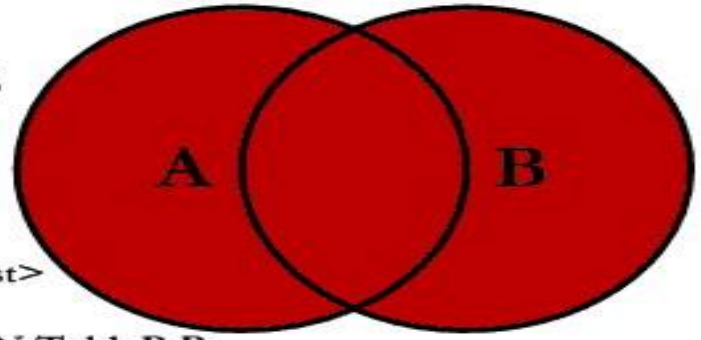

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



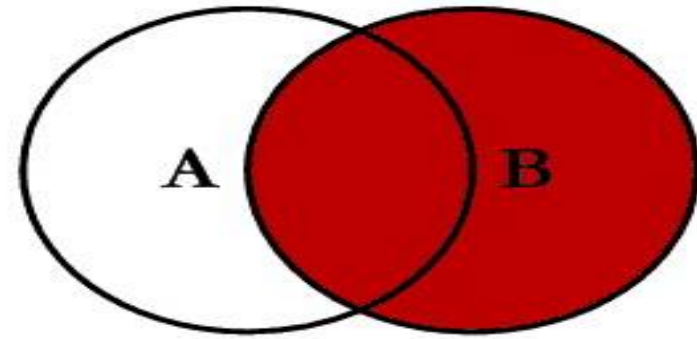

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



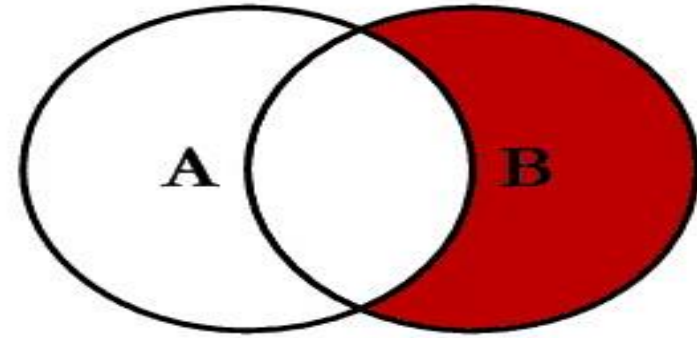

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



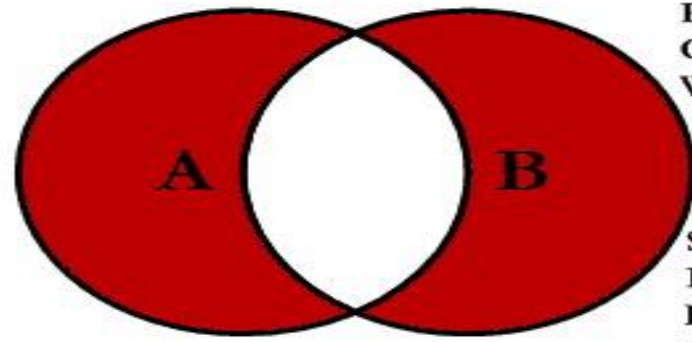

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

COMANDO SELECT INTO

Comando disponível apenas no PostgreSQL. Com ele, é possível criar uma nova tabela com base em um resultado de uma consulta.

Por exemplo, temos uma tabela PRODUTOS com vários campos e queremos selecionar apenas o nome e o preço para então armazenar essa consulta em uma nova tabela chamada TABELA_PRECOS. Observe a sintaxe:

```
SELECT nome_produto, preco
INTO tabela_precos
FROM produtos;
```

GERENCIANDO USUÁRIOS

Os comandos DDL, DML e DQL também são usados para gerenciamento de usuários, em sua criação, edição, exibição e remoção. O MySQL possui o banco de dados “mysql” com tabela ***user*** por padrão, onde ficam armazenados os usuários.

- **CONSULTAR OS USUÁRIOS:** ***SELECT user, host FROM mysql.user***
- **CRIAR UM USUÁRIO:** ***CREATE USER ‘usuário’@’host’ IDENTIFIED BY ‘senha’***
- **DEFINIR UMA SENHA:** ***SET PASSWORD FOR ‘usuário’@’host’ = PASSWORD (‘senha’)***
- **RENOMEAR UM USUÁRIO:** ***RENAME USER nome TO novonome***
- **APAGAR UM USUÁRIO:** ***DROP USER usuario***

DCL - QUEM PODE O QUE?

A DCL contém comandos que controlam o acesso e as operações dentro de bancos de dados e seus componentes (tabelas, colunas, etc). Os dois comandos possíveis são GRANT (conceder permissão) e REVOKE (revogar permissão). Para mostrar as permissões utilizamos o comando SHOW GRANTS, com a sintaxe ***SHOW GRANTS FOR usuário@host.***

- **PERMISSÕES DE DADOS:** insert, update, delete, execute, select
- **PERMISSÕES DE ESTRUTURA:** create, alter, drop, views, trigger, procedure
- **PERMISSÕES ADMINISTRATIVAS:** create user, show databases, shutdown, reload
- **PRIVILÉGIOS SUPERIORES:** all, grant option, usage.

As permissões podem ser dadas em nível global, por banco de dados, por tabela ou por coluna.

DCL - QUEM PODE O QUE?

Para conceder uma permissão utilizamos o comando GRANT, na seguinte sintaxe:

GRANT permissão ON banco.tabela TO usuário;

As permissões compostas são separadas por vírgula. Para conceder todas as permissões, utilizamos **ALL PRIVILEGES.** Quando desejamos afetar todos os bancos de dados ou todas as tabelas, substituímos o nome por um asterisco (*). Há ainda a opção **WITH GRANT OPTION** ao final da sintaxe que permite ao usuário também administrar as permissões dos outros usuários.

GRANT usage ON loja.* TO suporte@localhost;
GRANT all ON *.* TO jonas@localhost WITH GRANT OPTION;
GRANT select, insert, update, delete ON livraria.* TO ana@localhost;
GRANT select(titulo, preco), update(preco) ON livraria.livros TO jonas@localhost;

COMANDOS DCL

Para retirar uma permissão/privilégio, utilizamos o comando REVOKE, na seguinte sintaxe:
REVOKE permissão ON banco.tabela FROM usuário;

Podemos remover todas as permissões utilizando a palavra "**ALL PRIVILEGES**", porém se o usuário contiver a permissão **GRANT OPTION**, é necessário incluí-la também na sintaxe do comando, por exemplo ***REVOKE all privileges ,GRANT OPTION FROM suporte;***

Com os comandos DCL, podemos também criar o usuário ao atribuir permissões, bastando adicionar o complemento ***IDENTIFIED BY 'senha'*** ao final do comando. Ao remover ou conceder permissões, pode ser necessário realizar o comando ***FLUSH PRIVILEGES*** para o MySQL "recarregar" os privilégios de todos os usuários.

DTL - NEGOCIANDO SEGURAMENTE

A DTL contém comandos para transações. Uma transação em banco de dados é vista como um conjunto de operações em que o resultado é único para a transação inteira. Na prática significa que ou tudo dá certo, ou tudo dá errado (mesmo que parcialmente certo). As propriedades de uma transação são:

- **Atomicidade:** Como um AND lógico. Qualquer operação com falha, cancela toda a transação.
- **Consistência:** Uma transação não afeta registros, nem estruturas de tabela. Tudo fica íntegro.
- **Isolamento:** A transação trabalha sozinha! Os dados e operações contidas nela, assim como seu resultado, só ficam acessíveis após o termino da transação.
- **Durabilidade:** As operações da transação são permanentes. Nada é desfeito ou perdido, após seu término, mesmo que haja falhas no sistema.

COMANDOS DTL

Para utilizar transações adequadamente, podemos desativar a implementação automática (chamada 'commit') presente em alguns banco de dados, que faz com que alguns comandos sejam executados imediatamente, como um DELETE ou INSERT. Executamos então a linha ***SET @@autocommit = OFF.*** Os principais comandos da DTL são:

- **START TRANSACTION/BEGIN WORK**: Inicia a transação;
- **COMMIT**: Confirma a transação e "libera" o resultado para acesso.
- **ROLLBACK**: Cancela todas as operações da transação.
- **SELECT @@autocommit:** mostra o status da auto execução.
- **SAVEPOINT**: um ponto lógico, com um nome, para onde podemos limitar o ROLLBACK, com a sintaxe ROLLBACK TO SAVEPOINT <nome>. Permite também um comando RELEASE SAVEPOINT para retirar da memória uma referência a um ponto lógico

ROTINAS - STORED PROCEDURES

As Stored Procedures são comparáveis a um procedimento em lógica de programação ou a uma macro. Trata-se de um bloco de código SQL com determinadas ações gravadas, que pode ser executado a qualquer momento para vários objetivos, inclusive com parâmetros.

A sintaxe simples para criação é:

CREATE PROCEDURE nome (PARAMETROS)
Comandos SQL;

E para executar um procedimento, utilizamos a sintaxe:

CALL nome (PARAMETROS);

ROTINAS - STORED PROCEDURES

Vamos ver um exemplo de uma STORE PROCEDURE implementada para recuperar o nome de um cliente.

```
CREATE PROCEDURE mostrar_nome (id int)
SELECT CONCAT('O nome do cliente é ', nome) FROM clientes WHERE id = id;
```

Para executar essa procedure, podemos utilizar

```
CALL mostrar_nome (3);
```

Para excluir uma procedure do banco de dados, utilizamos também seu nome na sintaxe:

```
DROP PROCEDURE mostrar_nome;
```

ROTINAS - FUNCTIONS

As Functions são comparáveis a uma função em lógica de programação. Sua definição é exatamente como a uma Stored Procedure, porém, assim como na lógica de programação, uma function **retorna** uma resultado para onde foi chamada. Difere também na chamada, sendo utilizada dentro do comando SELECT.

A sintaxe simples para criação é:
CREATE FUNCTION nome (PARAMETROS) RETURNS tipo
Comandos;
RETURN resultado/expressão

Para utilização, colocamos a função como uma “coluna” no comando SELECT.
Para remover uma função, utilizamos a sintaxe ***DROP FUNCTION nome***

ROTINAS - FUNCTIONS

Vamos ver um exemplo de uma FUNCTION implementada para calcular a comissão de um vendedor.

```
CREATE FUNCTION calcula_comissao (vendas double, percentual int) RETURNS double
RETURN vendas *  percentual/100
```

Para utilizar essa função, a invocamos juntamente com o comando SELECT

```
SELECT calcula_comissao(3000, 12);
```

Os parâmetros podem vir de uma consulta SELECT diretamente também.

```
SELECT nome, calcula_comissao(total_vendas, 12) FROM vendas WHERE id = 3;
```

VIEWS

Uma Visão é como uma tabela comum, com todos os comportamentos inclusos, porém ela não existe fisicamente como tabela, sendo apenas virtual. Pode ser vista como um ‘SELECT gravado’, ou seja, traz a vantagem de ter sempre os dados atualizados a cada nova execução. Pode ser vista também como uma ‘nova tabela personalizada’, permitindo um novo SELECT sobre ela.

Sua sintaxe é bem simples, como abaixo:

CREATE VIEW nome AS
Comando SELECT

Para alterar uma view, apenas trocamos CREATE por ***ALTER***. Para excluir, utilizamos ***DROP.***

TRIGGERS

Um trigger é um procedimento invocado automaticamente quando há manipulação de registros em uma tabela do banco de dados, seja exclusão, inserção ou atualização. A sintaxe é:

***CREATE TRIGGER** nome momento operação*
ON** tabela **FOR EACH ROW
Comandos SQL

O Trigger deve ter um nome para identificação, um momento de execução (BEFORE ou AFTER) em relação a operação e a operação propriamente dita, se INSERT, UPDATE ou DELETE. Por exemplo, podemos atualizar um campo DATA/HORA toda vez que um registro for atualizado, ou seja, quando for executado o comando UPDATE. Para diferenciar campos numa trigger de atualização utilizamos as palavras NEW (campo atualizado) e OLD (campo atual).

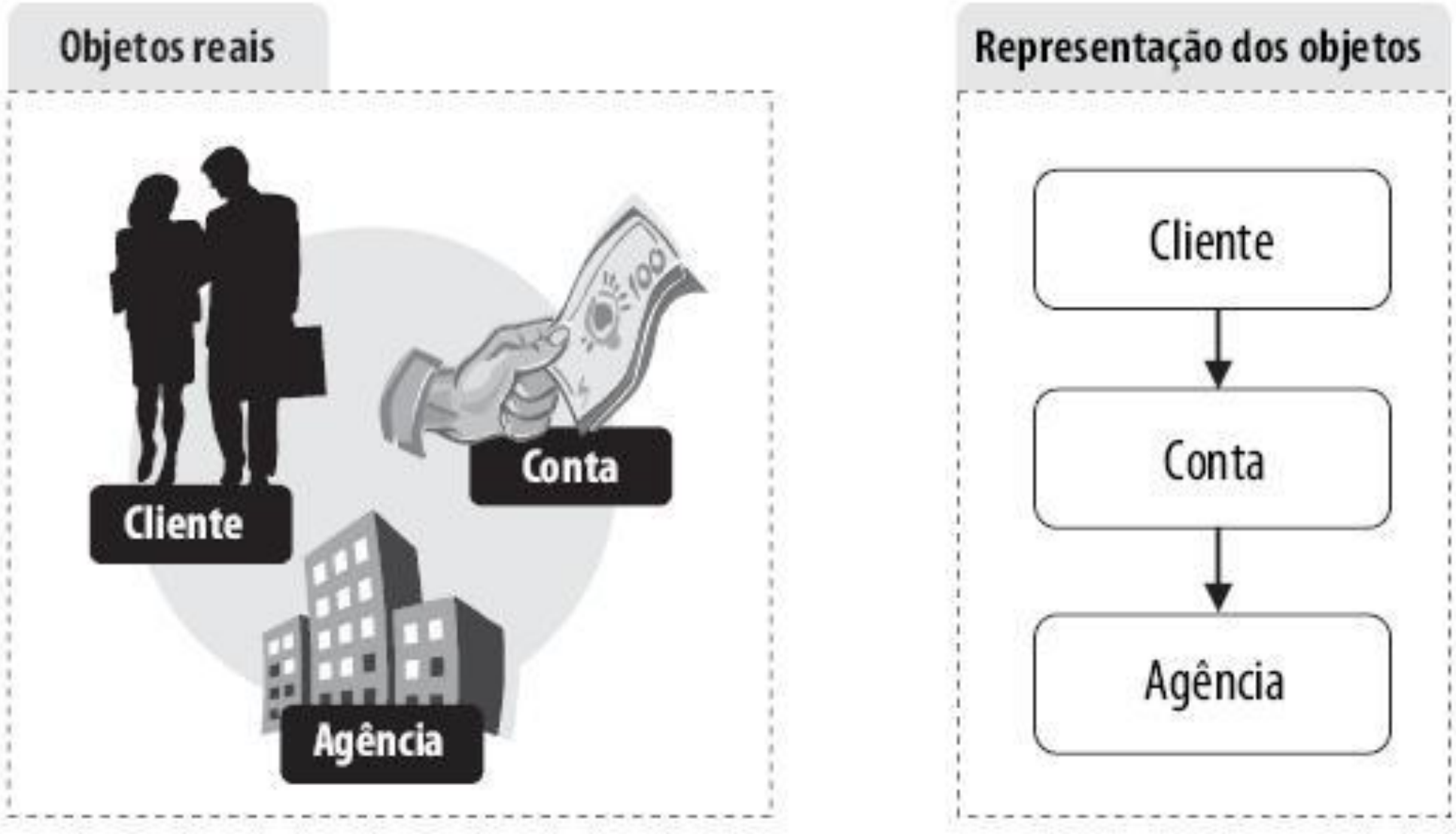
ORIENTAÇÃO A OBJETOS

TUDO É OBJETO!

A orientação à objeto é uma maneira de se pensar e desenvolver programas, é um **paradigma**. Faz referência direta com objetos do mundo real e como eles se relacionam, fazendo uma **abstração**. deixamos assim de pensar em ‘variáveis’ para o software, para pensar em ‘objetos’.

Por exemplo, um sistema bancário não manipula diretamente clientes, contas e cheques. Apenas faz uma representação desses objetos com os atributos e comportamentos do mundo real.

De um modo geral, beneficia a facilidade de programar e manutenção do código.


Objetos reais
Cliente
Conta
Agência
Representação dos objetos
Cliente
Conta
Agência

CLASSE

Ao agrupar caraterísticas únicas de um componente, temos então o conceito de classe. É uma coleção de objetos, sendo a generalização deles. As classes podem relacionar-se entre si, de acordo com a abstração proposta.

Por exemplo, no sistema bancário, a coleção de clientes (classe Cliente) se relaciona com a coleção de contas (classe Conta), onde os “objetos clientes possuem um ou mais objetos contas”.

OBJETO

As classes geram objetos chamados **instâncias,** onde o código é realmente aplicado. Uma classe Cliente, por exemplo, geram vários objetos (‘clientes’) que serão relacionados e trabalhados para o funcionamento do software.

Objetos de uma mesma classe podem também assumirem papéis independentes no sistema. como um todo;

ATRIBUTOS

Classes agrupam características específicas dos objetos chamadas **atributos.** Os atributos podem servir como identificação para os objetos criados e seus valores podem sofrer alterações durante a execução do software.

Por exemplo, uma classe Cliente pode conter atributos como nome, idade, entre outros, significando que cada objeto criado (cliente) carregará esses atributos consigo. Por fim, “Ana Maria” pode ser o valor do atributo ‘nome’ a um objeto específico, gerado da classe Cliente.

MÉTODO

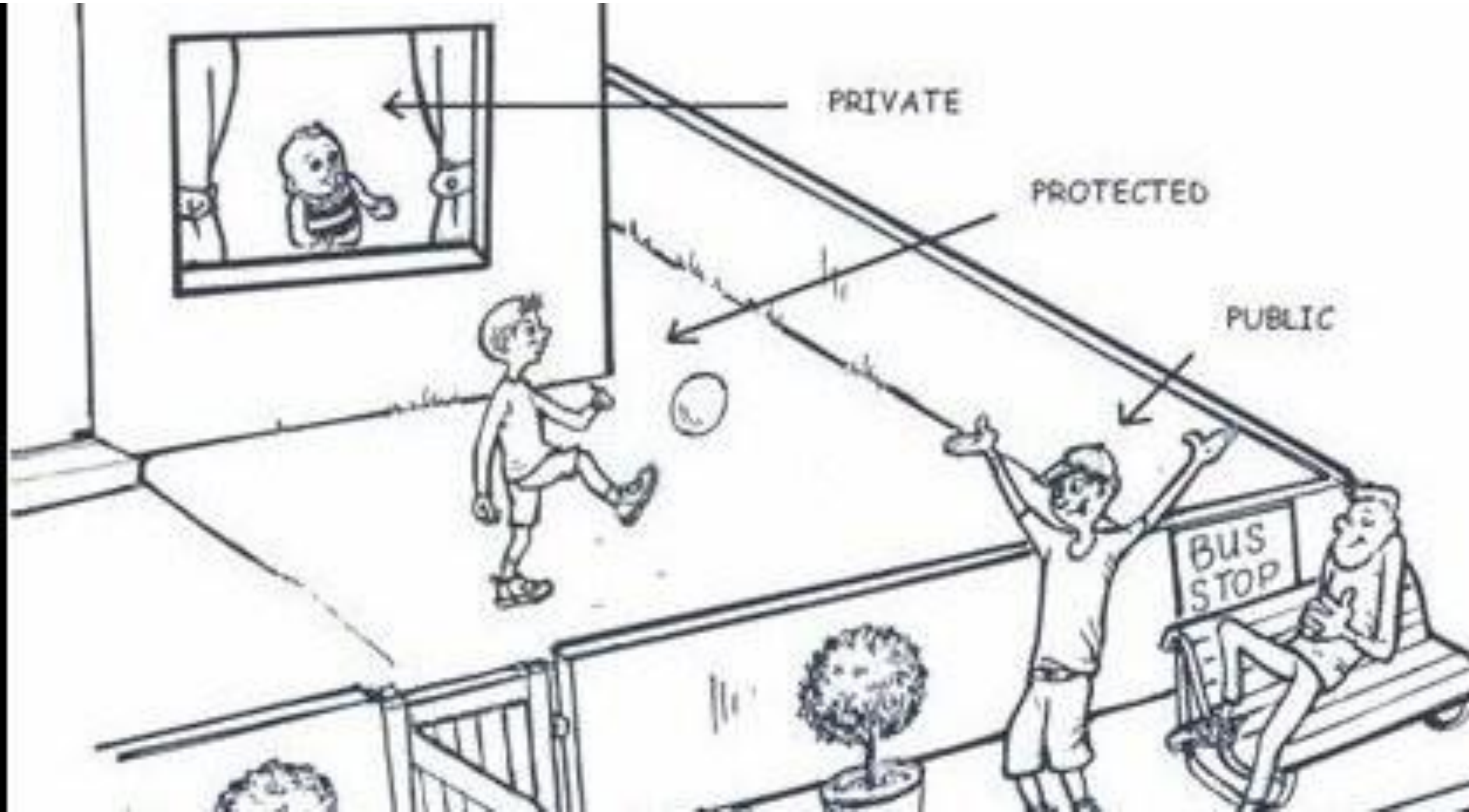
Já os métodos não são características e sim ações e comportamentos, realizados ou sofridos por objetos de uma classe. Os métodos servem em suma para manipular os atributos dos objetos e são geralmente identificados por verbos.

Por exemplo, uma classe Animal pode ter métodos como, andar, fazer som, crescer, pular, etc. Um método ‘andar’ pode alterar por exemplo o atributo ‘localização’, enquanto um método ‘crescer’, pode alterar um atributo ‘altura’.

TIPO DE ACESSO

Um método ou classe pode ter diferentes tipos de acesso, que determina o controle de uso. Basicamente, temos 3 tipos de acesso para orientação a objeto.

- **Público**: um método ou classe pública pode ser aproveitado dentro do código em qualquer ponto do programa, isto é, em qualquer arquivo e qualquer linha.
- **Privado**: um método ou classe privada pode apenas ser utilizado no próprio arquivo que ele foi declarado. Isso protege, por exemplo, de uso indevido quando há distribuição de código.
- **Protegido**: um método ou classe protegida comporta-se como privada, mas pode se tornar acessível através de **herança.**


PRIVATE
PROTECTED
PUBLIC
BUS STOP

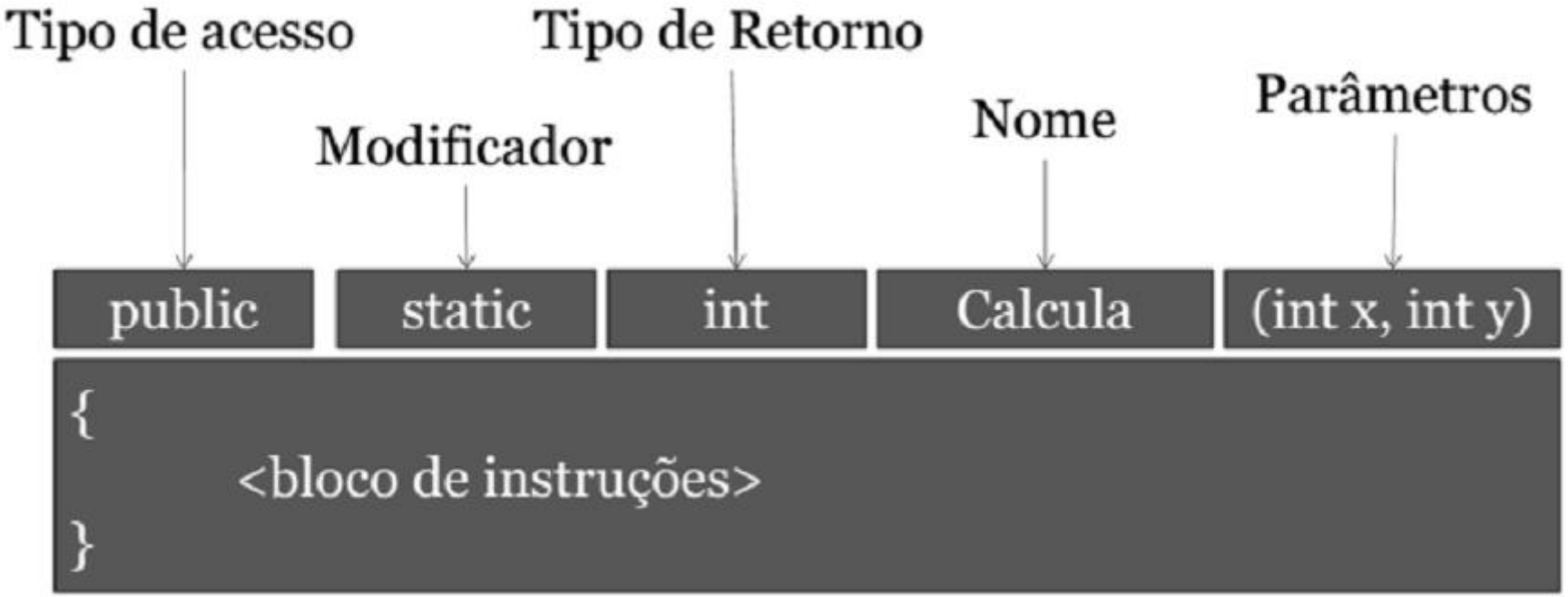
MODIFICADORES DE MÉTODO

Um método ou classe também pode se comportar de diferentes maneiras, através dos modificadores. Isso ajuda a controlar o funcionamento, principalmente em casos de redistribuição de código.

- **Abstrato**: um método ou classe abstrata não possui corpo, isto é, não possui “código”, contendo apenas a declaração. Quando ele for utilizado efetivamente, deve então ser implementado.
- **Estático**: um método estático é aquele que não é atrelado a nenhum objeto, funcionando por si através da Classe.
- **Final**: um método ou classe final não pode mais ser reutilizada, isto é, não pode ser herdada.

ASSINATURA DE MÉTODO

Um método deve ser declarado para existir dentro do sistema, como um todo. Essa declaração chamamos de assinatura e tem a seguinte sintaxe.



TIPOS DE MÉTODO

Temos 4 tipos de métodos, definidos pelo seu comportamento. Cada tipo de método tem características distintas, de corpo, parâmetros e modificadores.

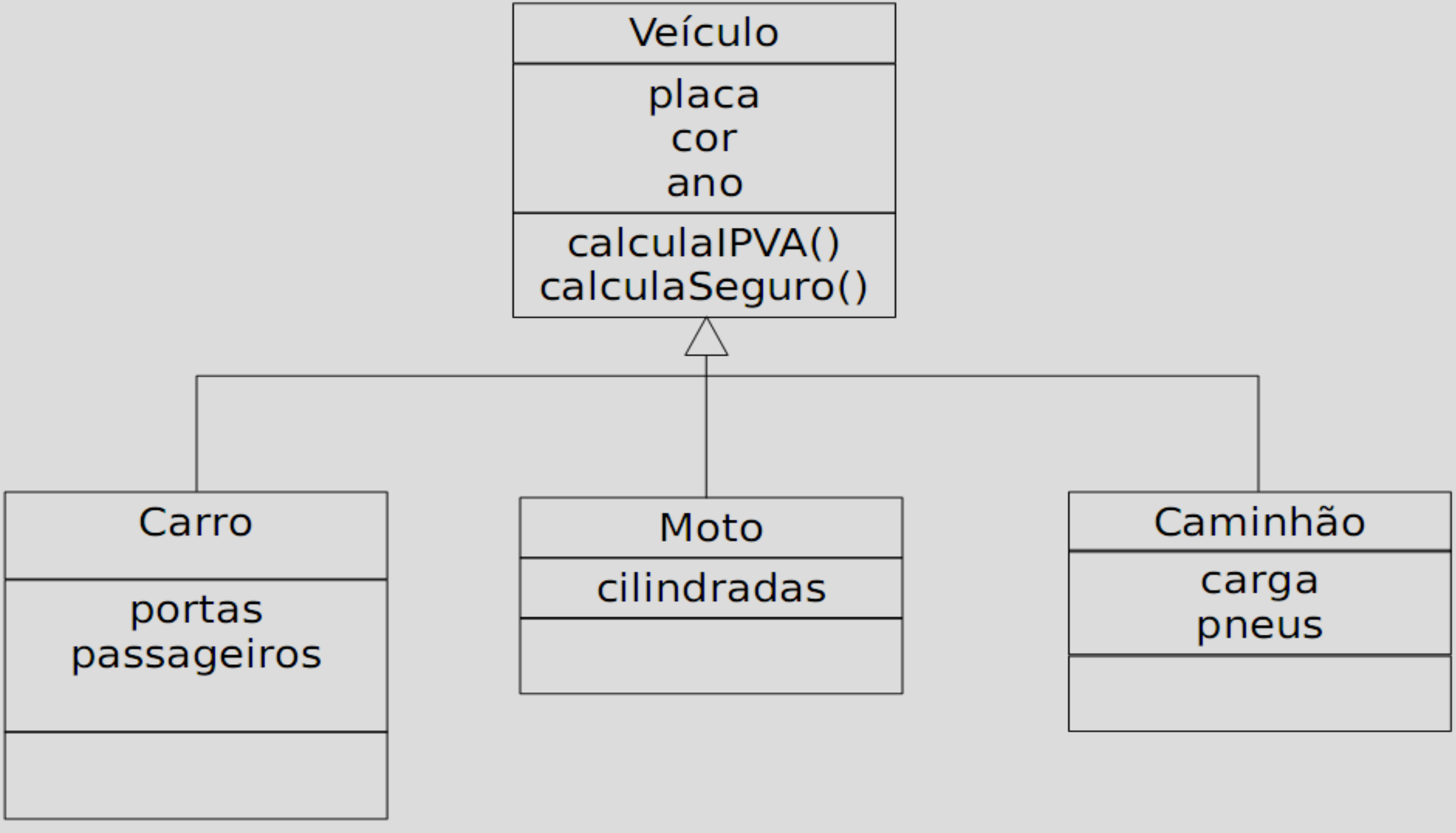
- **Construtor**: este é o método que gera um novo objeto, podendo inicializar os atributos com valores definidos ou com valores padrão. Inclusive podemos definir diferentes métodos construtores para diferentes parâmetros.
- **Mutante**: método que irá modificar o valor de um atributo. Não possui retorno.
- **Acessor**: método que irá selecionar o valor de um atributo, podendo ser exibido na tela ou utilizado em outros processamentos.
- **Principal**: método que será executado por padrão ao inicializar o programa.

HERANÇA

Quando uma classe recebe características e métodos de uma outra classe superior, chamamos esse processo de **herança**. É uma espécie de 'especialização' de uma classe mais genérica. Tem como principal vantagem a reutilização do código, pois o mesmo comportamento pode ser atribuído a várias classes.

Por exemplo, uma classe Animal pode conter atributos e métodos genéricos a todos os animais. E uma classe Mamífero pode herdar o conteúdo da classe Animal, para que não seja necessário recodificar novamente partes comuns entre elas.

A classe herdeira é referenciada como classe-filha ou subclasse, sendo a outra, a classe-mãe ou superclasse. A classe herdeira também pode ter seus próprios métodos e atributos ou então reescrever métodos da classe-mãe.


Veículo
placa
cor
ano
calculaIPVA()
calculaSeguro()
Carro
portas
passageiros
Moto
cilindradas
Caminhão
carga
pneus

ENCAPSULAMENTO

O conceito de encapsulamento é complexo e consiste em proteger o código, no que corresponde ao seu funcionamento interno, permitindo ao usuário o acesso somente aos métodos para manipular a classe. O objetivo é evitar que o usuário manipule os atributos direta e livremente.

Por exemplo, quando o usuário solicita ao sistema uma lista de produtos, ele não precisa saber como o sistema irá recuperar esta lista. O usuário deve ter acesso apenas ao resultado, isto é, o acesso ao funcionamento interno é restrito, devendo o usuário ter acesso somente ao que está especificado no código, sem manipulação livre.

ACELERAR

FREAR

BUZINAR

POLIMORFISMO

É uma propriedade que ocorre quando um mesmo método comporta-se de maneira diferente em várias partes do código. É pertinente à herança de classes, podendo os métodos herdados serem sobrescritos.

Por exemplo, uma classe veículo que contenha um método chamado ‘acelerar’. Ao ser herdado, este método irá se comportar de diferentes formas em cada veículo específico.


Mamífero
Macaco
Homem
Baleia
Homem X
Especialização
Generalização

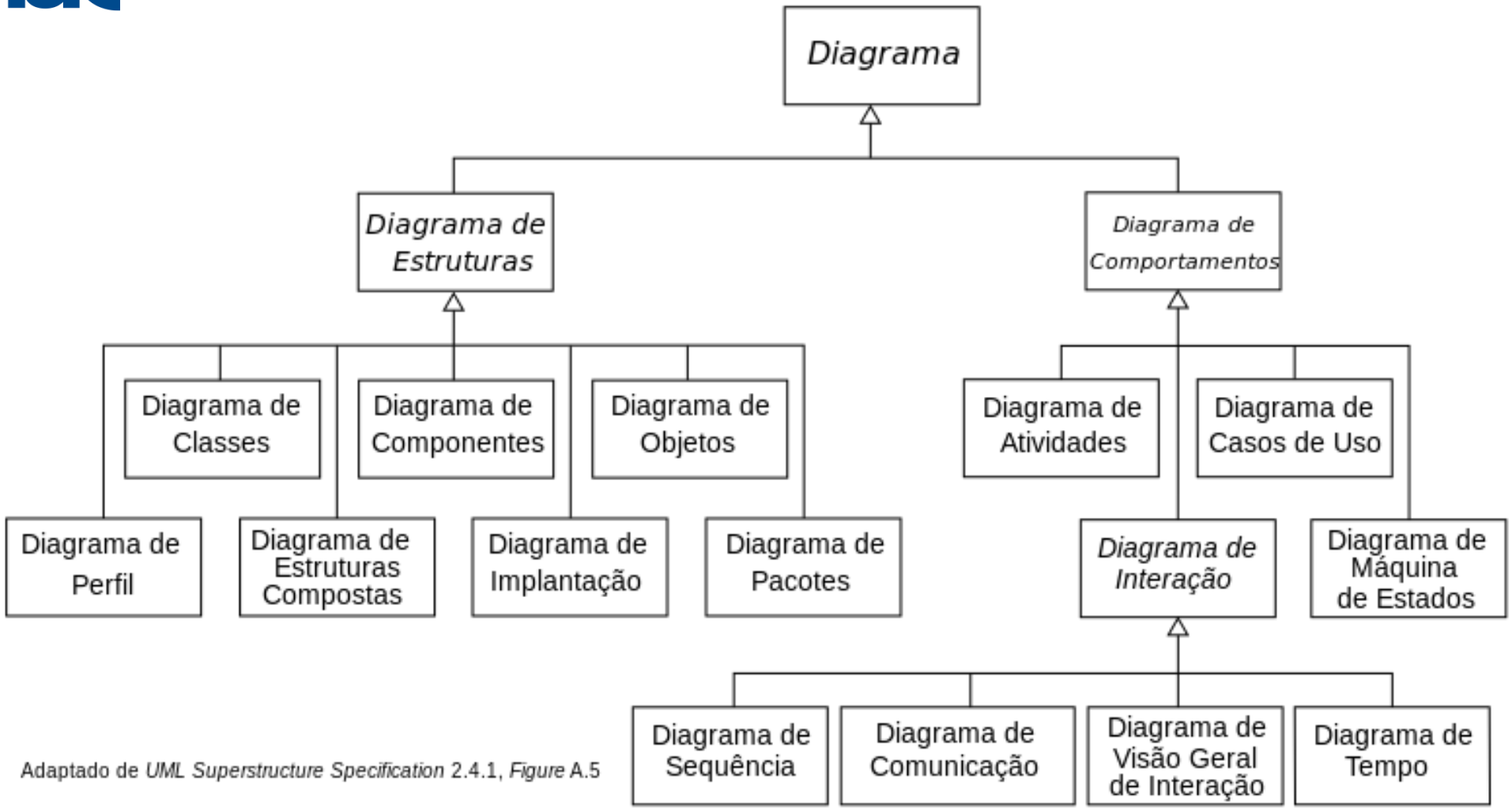

Agora.
"Fale!"
Quack!
Quack!"
"Au! Au!"
"Miau!"

UML

A UML é um conjunto de notações padronizadas para representar graficamente um software. É chamada de linguagem de modelagem. Com os modelos UML, pode-se representar várias fases do projeto. Uma das formas amplamente utilizadas são os diagramas da UML com várias finalidades que auxiliam na análise de problemas orientado a objetos.

A UML é muitas vezes, erradamente, confundida como uma linguagem de programação.

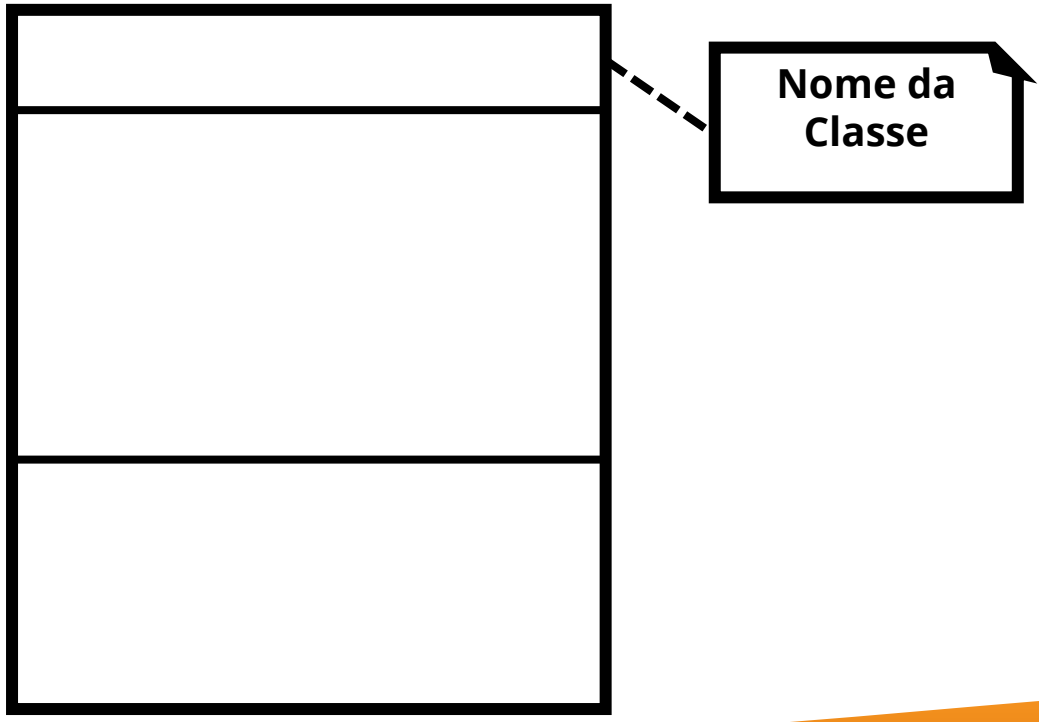
VISÃO GERAL DA UML



Adaptado de *UML Superstructure Specification* 2.4.1, *Figure* A.5

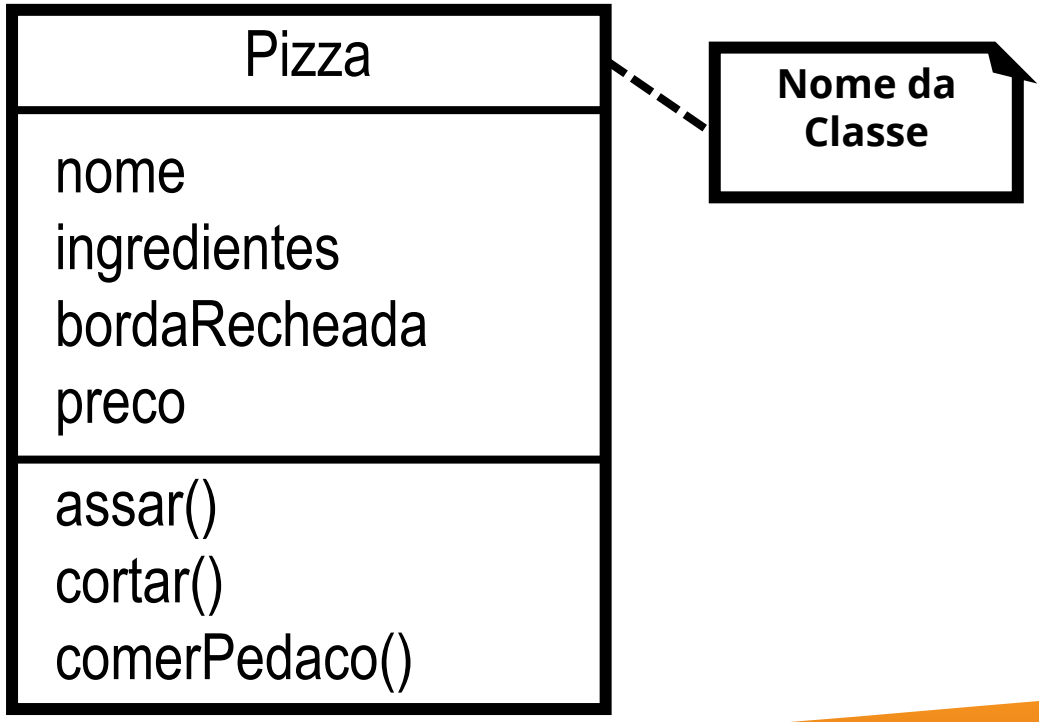
NOTAÇAÕ DE DIAGRAMA DE CLASSE

Um retângulo preto representa um classe. Ele é dividido em três partes. Na parte superior fica o nome da classe. No meio, ficam descritos os atributos da classe. E na parte inferior ficam as operações (métodos) possíveis.



NOTAÇAÕ DE DIAGRAMA DE CLASSE

Um retângulo preto representa um classe. Ele é dividido em três partes. Na parte superior fica o nome da classe. No meio, ficam descritos os atributos da classe. E na parte inferior ficam as operações (métodos) possíveis.



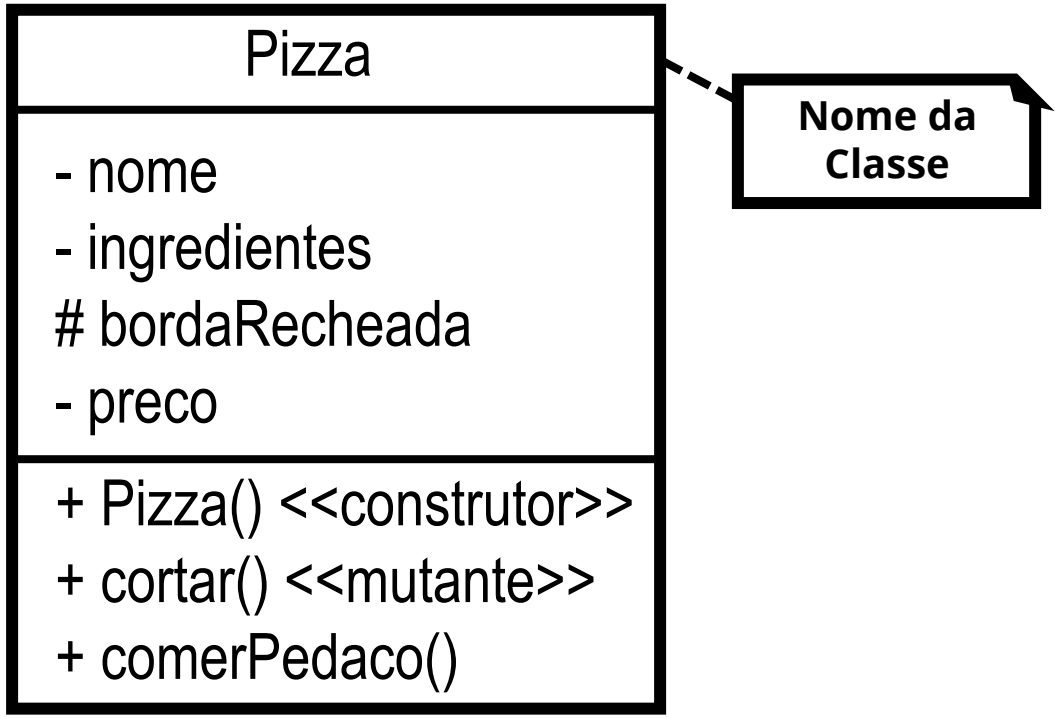
NÍVEIS DE ACESSO E ESTEREÓTIPOS

Na notação UML, podemos representar os níveis de acesso de atributos e operações:

- (hífen): nível privado;
+ (adição): nível público;
# (tralha): nível protegido;

Para encapsulamento, é comum termos atributos privados (ou protegidos) e métodos públicos. Para fins de visibilidade, a única notação reconhecida para a classe é “abstrata”, onde o nome fica em itálico.

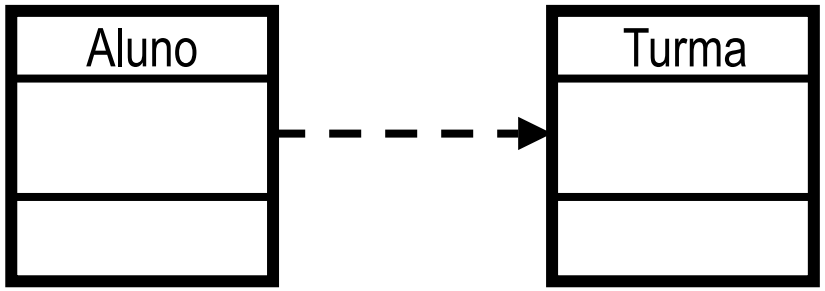
Já os estereótipos ajudam a refinar a notação, com mais descrições sobre os componentes, facilitando a visualização. São representados por sinais de menor e maior duplos << >>



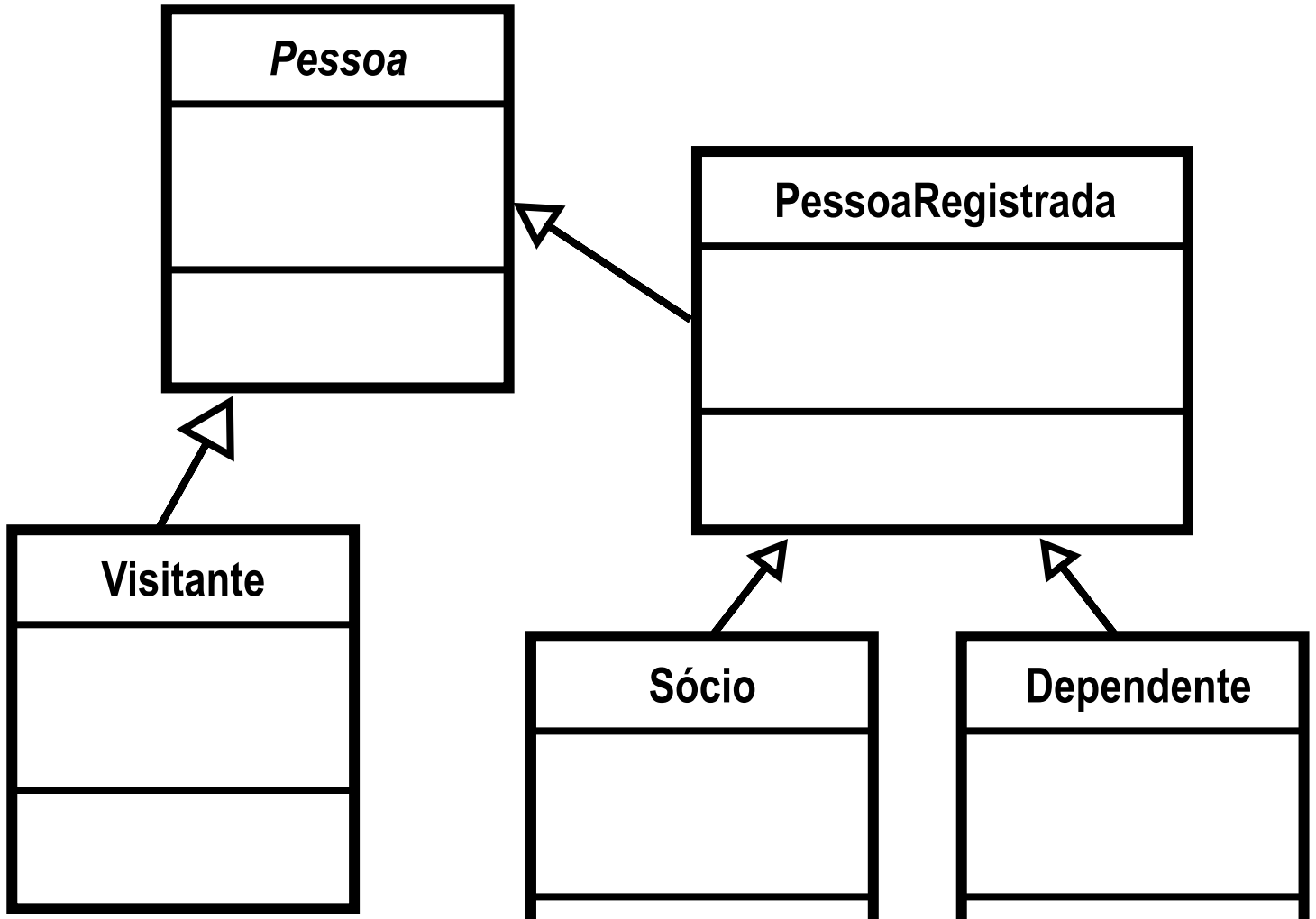
LIGANDO CLASSES

Para fazer sentido em sistemas, classes devem ser relacionadas para que possam "conversar". Esse relacionamento pode acontecer de três maneiras, representadas na UML.

- **Generalização**: como o nome sugere, vai do geral ao específico, caracterizando uma herança ('É-UM'). Permite também que um objeto da superclasse substitua qualquer objeto de qualquer subclasse, sem alterar o funcionamento do sistema. É representada por um seta vazada e fechada.
- **Dependência**: representa a mais simples forma de relacionamento, indicando que uma classe depende de outra para conseguir gerar objetos.



DEPENDÊNCIA



GENERALIZAÇÃO

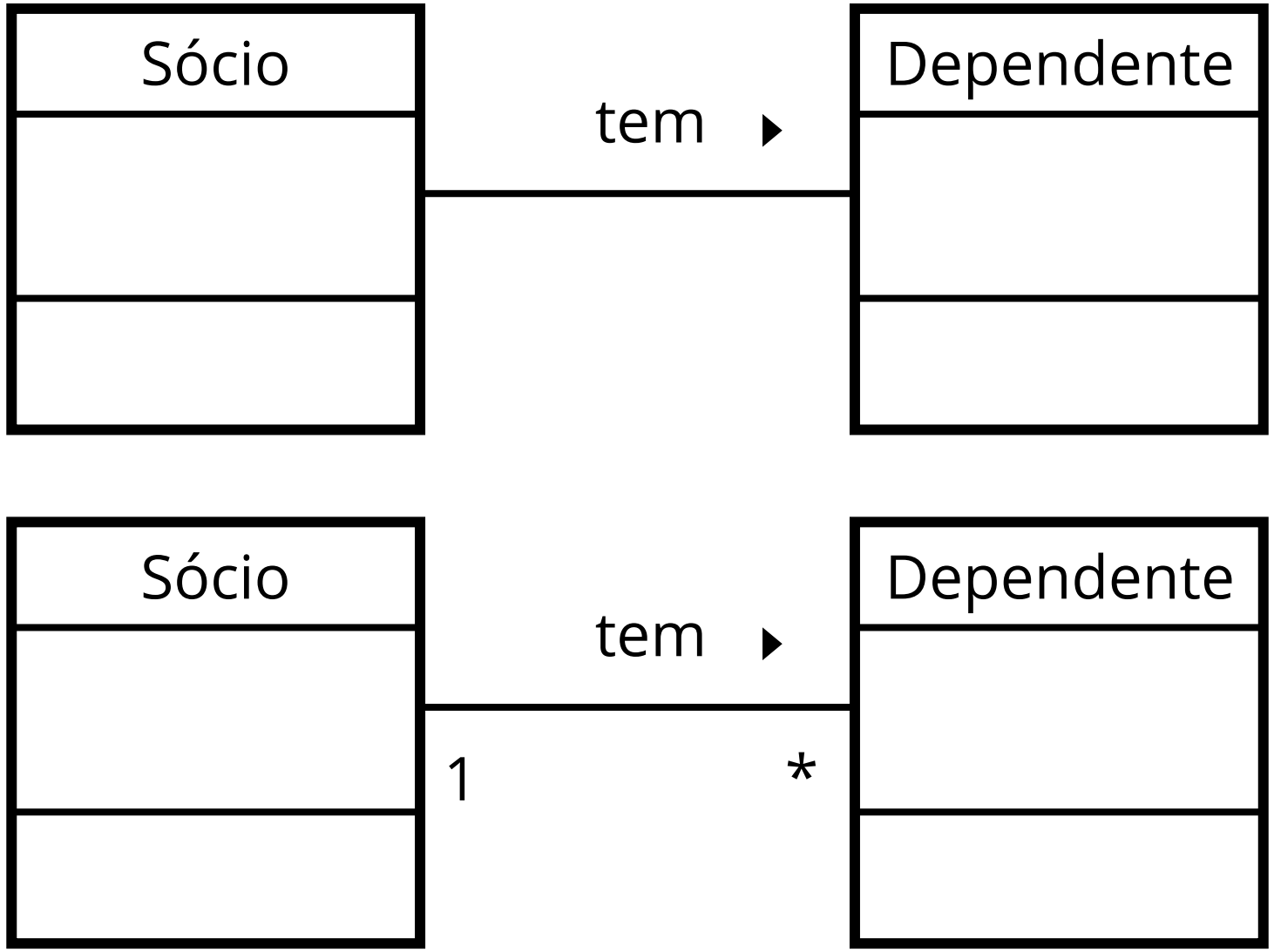
ASSOCIAÇÃO

A associação é a terceira forma de relacionamento entre classes, sendo a mais ampla. Quando uma classe faz parte da outra (‘TEM-UM’), temos uma associação. É representada por uma linha, com uma seta, indicando a direção da associação. Toda associação ainda deve conter um nome, para facilitar o entendimento. Ainda pode, opcionalmente, conter uma descrição para o papel de cada classe na associação.

Ainda temos a multiplicidade, que representa quantos objetos de uma classe são associados com tantos outros da outra classe. Pode-se indicar multiplicidade com um número (quantos objetos exatamente), uma lista (um intervalo) ou um asterisco (vários objetos).

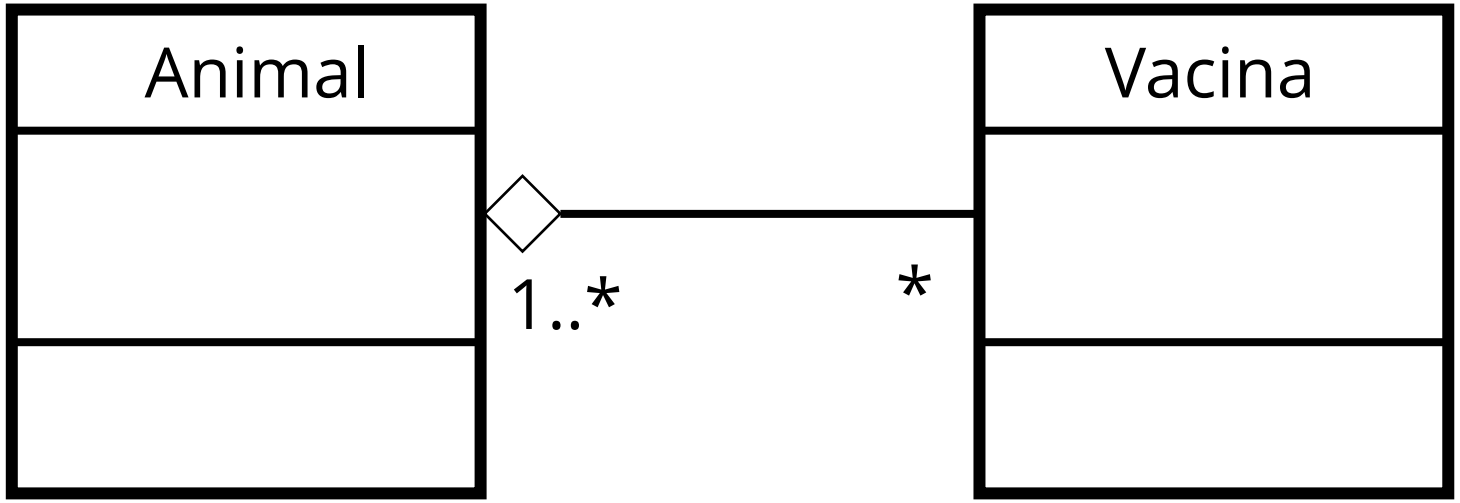
A notação ao lado indica que a classe Sócio é associada (‘TEM-UM’) com a classe Dependente. A essa associação deu-se o nome de TEM.

Na notação de multiplicidade ao lado, está indicado que 1 objeto Sócio pode conter vários objetos Dependentes. Chamamos essa associação de **UM-PARA-MUITOS.** Observe que, caso tivéssemos uma visão do lado da classe Dependente, essa associação caracterizaria UM-PARA-UM, isto é, um objeto Dependente poderia conter apenas um objeto Sócio.



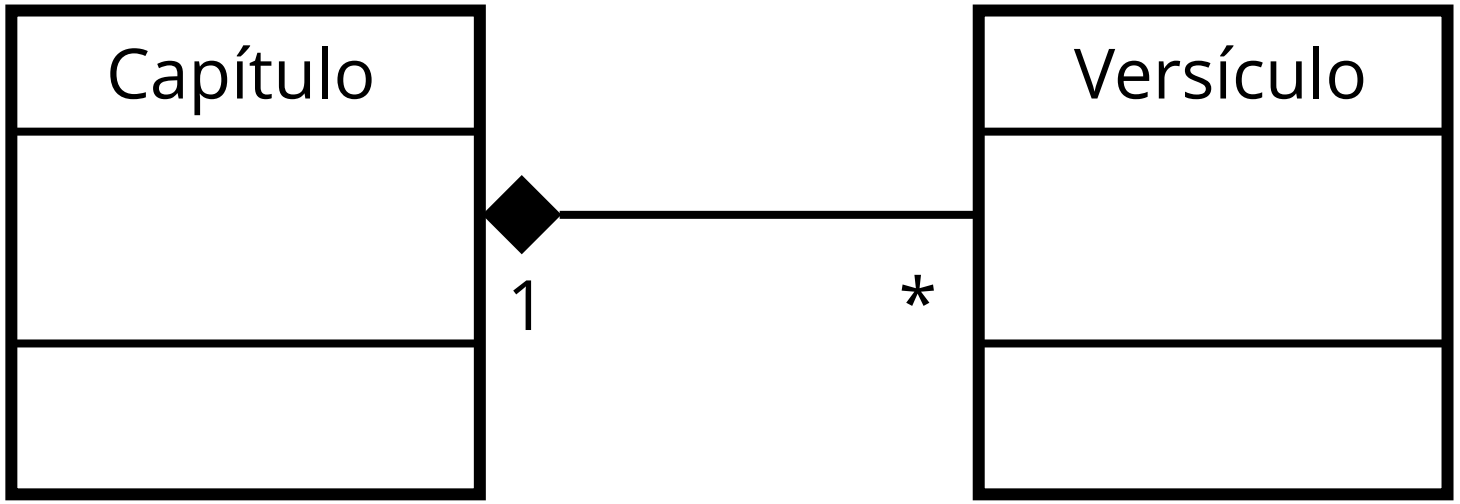
ASSOCIAÇÃO - AGREGAÇÃO

Um tipo específico de associação onde as classes apenas se complementam, sem causar dependência entre elas. Trata-se de uma relação do tipo **TODO-PARTE**. A independência entre as classes permite que um objeto seja excluído sem afetar o outro.



ASSOCIAÇÃO - COMPOSIÇÃO

Em uma associação do tipo composição, as classes não são independentes, pois um compõe a “existência” do outro. Isto significa que caso um objeto seja excluído, o outro também desaparecerá.



DIAGRAMAS DE CASO DE USO

Um caso de uso pode ser encarado como uma ação dentro do sistema. Um cadastro, um acesso, uma validação, etc. São sempre vistos a partir do ponto de vista do usuário. Descobrir as ações do sistema, ajuda a entender melhor as necessidades e possibilidades dos usuários.

Um caso de uso extenso e detalhado dificulta o entendimento do sistema. Isto é trabalho para a implementação. Um caso de uso deve representar uma abstração de forma geral. Nem tudo é caso de uso. Trate as partes mais críticas do sistema, deixando de lado as tarefas rotineiras.

A FIGURA DO ATOR

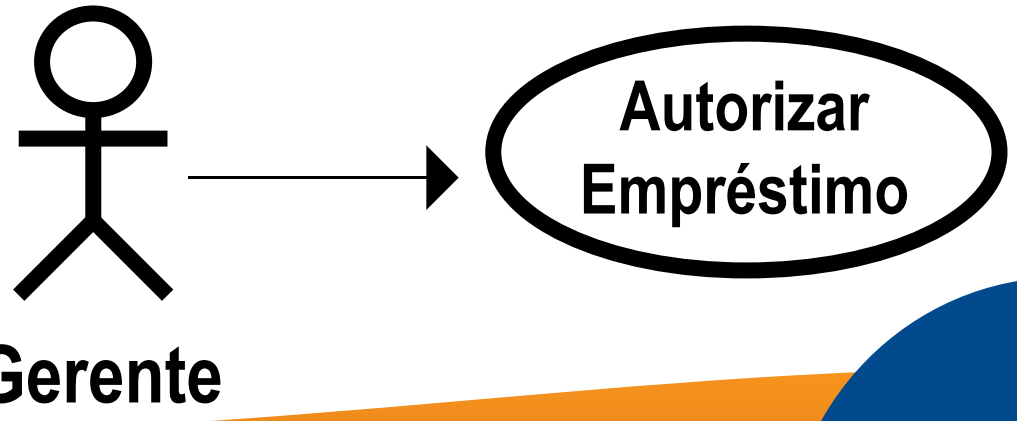
Os casos de uso partem sempre de atores. Os atores são responsáveis por aquela ação específica, formando um cenário para o caso de uso. Mas atenção! Os atores não são somente os usuários. As ações podem vir do próprio sistema ou de um sistema terceiro. Com algumas perguntas, podemos identificar os atores:

- Quem são os usuários básicos?
- Há outro sistema em conjunto (relógio de ponto, biometria, etc.)?
- Há banco(s) de dado(s)?

NOTAÇÃO E PAPEL DO ATOR

Para formular os casos de uso, é preciso definir as ações de cada ator no sistema. Alguns casos de uso podem ser combinados, outros podem ser separados. Há casos de uso que podem variar, por exemplo, um pedido pode ser feito por e-mail ou por telefone. Nestes casos, o melhor caminho é a abstração, simplificando os dois casos em apenas um.

Em UML, a ligação entre um caso de uso e um ator é representado por uma linha com uma seta fechada e preenchida na ponta.



SE TEM ATOR, TEM CENÁRIO!

Imagine abrir um conta no banco. São necessários alguns passos e cumprimento de algumas tarefas para isso. Chamamos esse conjunto de etapas de cenário. Durante as etapas, podem ocorrer problemas ou então sair tudo como planejado. Chamamos isso de **caminho principal** (caminho “sem erros”) e caminho alternativo (possibilidades de erro).

Os cenários podem requisitar informações iniciais antes de executar suas tarefas. São chamadas de pré-condições. Sem o cumprimento destas, o caso de uso não inicia, isto é, as ações não ocorrem. Em alguns casos, as ações podem retornar informações ao serem executadas. Ou o caso de uso como um todo pode ter um “resultado final”. A isto damos o nome de pós-condições.

RELACIONAMENTO ENTRE CASOS DE USO

Há dois tipo de relacionamento entre casos de uso, de inclusão ou de extensão. Um relacionamento de **inclusão** determina que um caso de uso OBRIGATORIAMENTE inclua outro caso de uso antes ou durante sua execução.

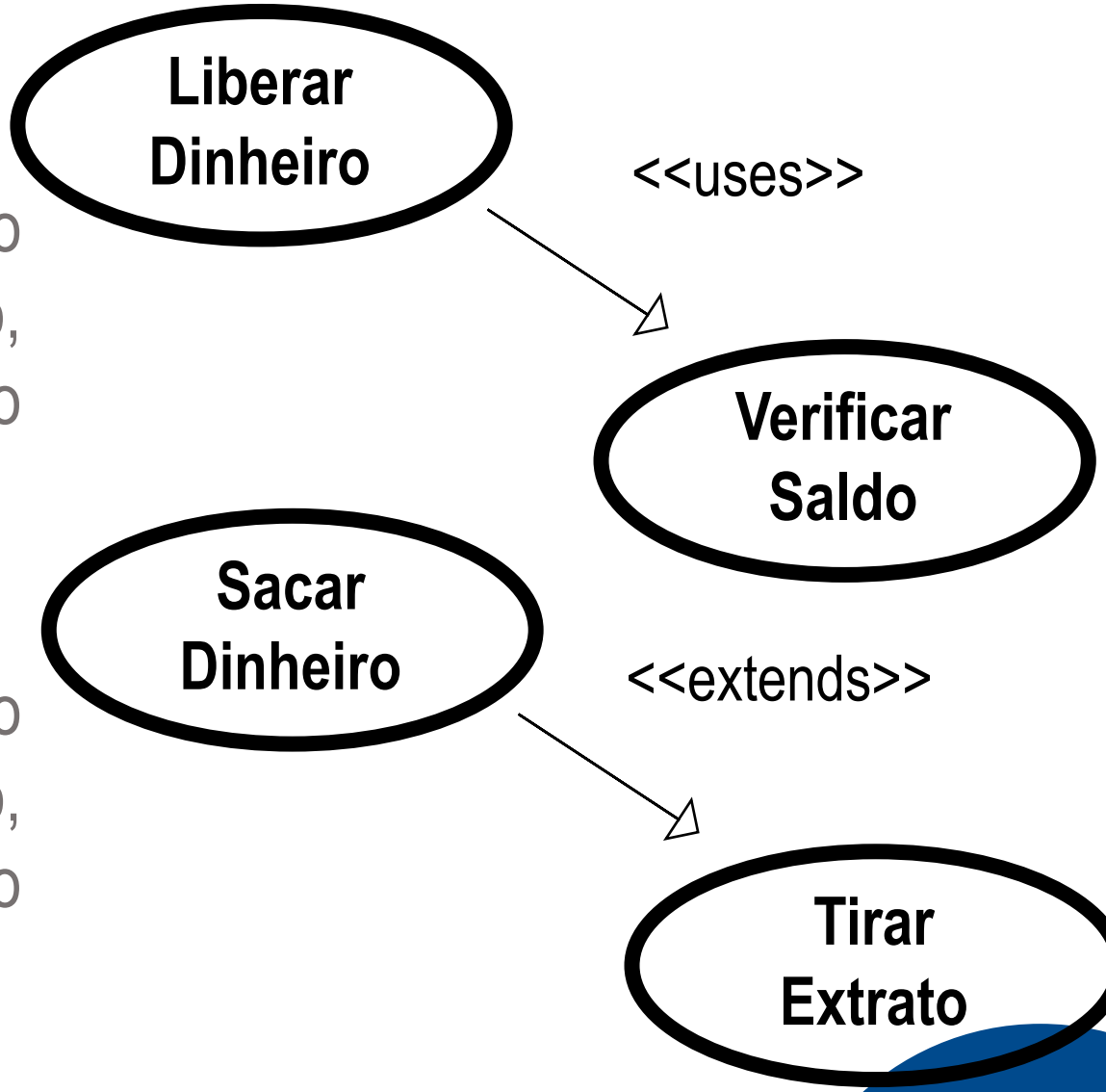
Já um relacionamento de **extensão**, indica que um caso de uso, pode OPCIONALMENTE estender (ir para) para outro caso de uso antes, durante ou depois de sua execução.

INCLUSÃO

Imagine a situação acima onde o ator seja o Caixa Eletrônico. Para especificar a inclusão, utilizamos um estereótipo UML identificado como **USES**.

EXTENSÃO

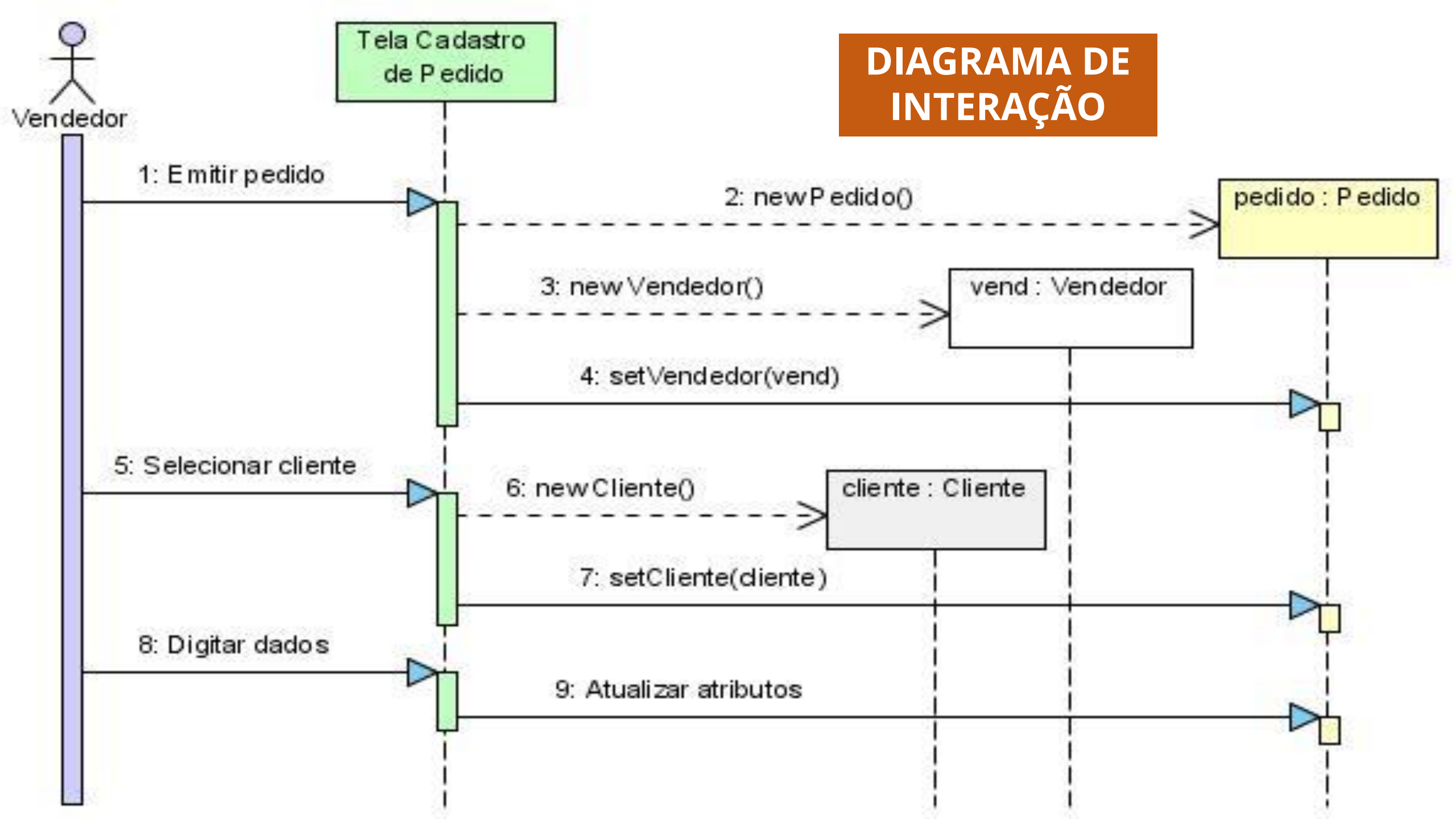
Imagine a situação acima onde o ator seja o Cliente. Para especificar a extensão, utilizamos um estereótipo UML identificado como **EXTENDS**.

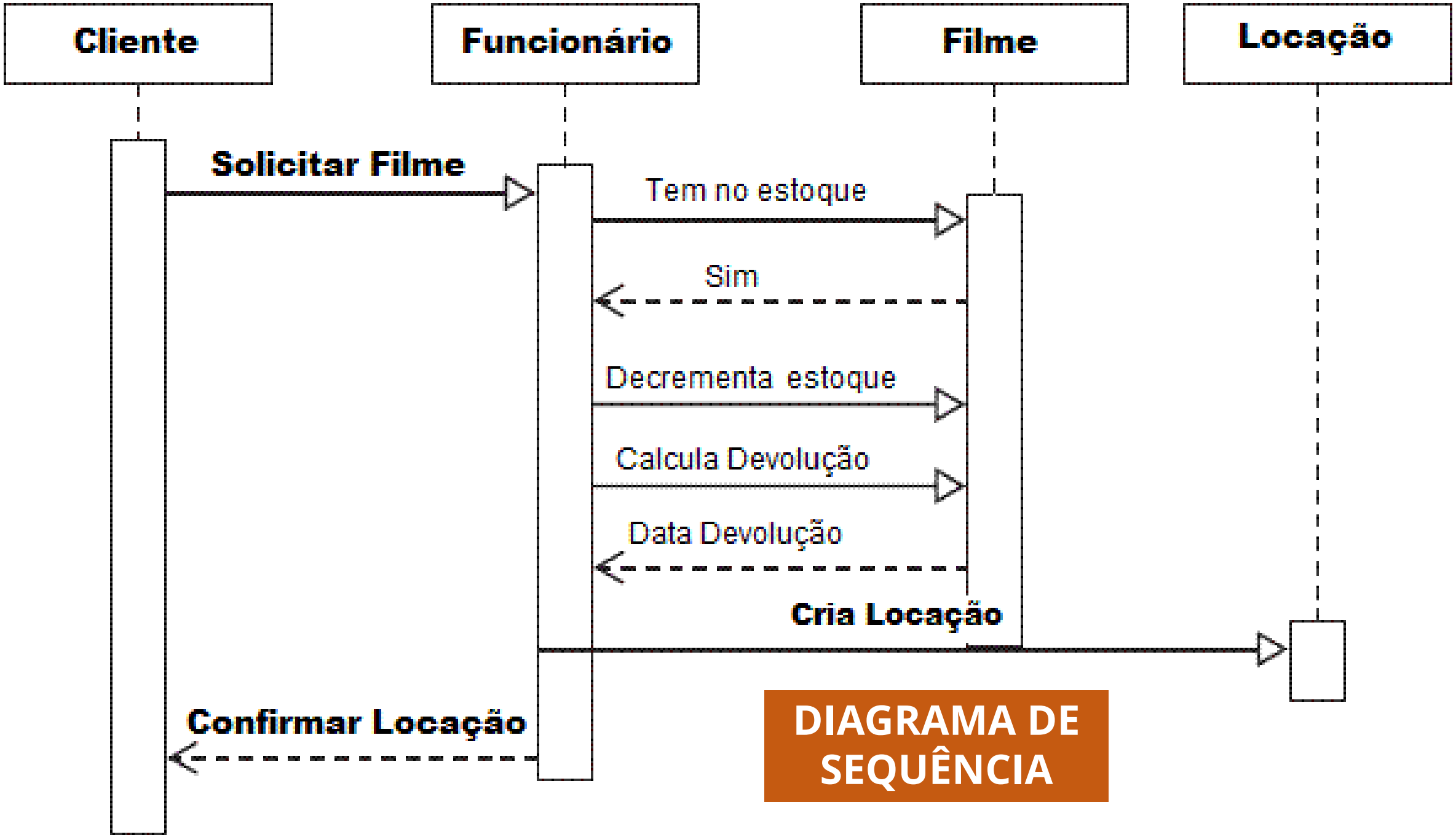


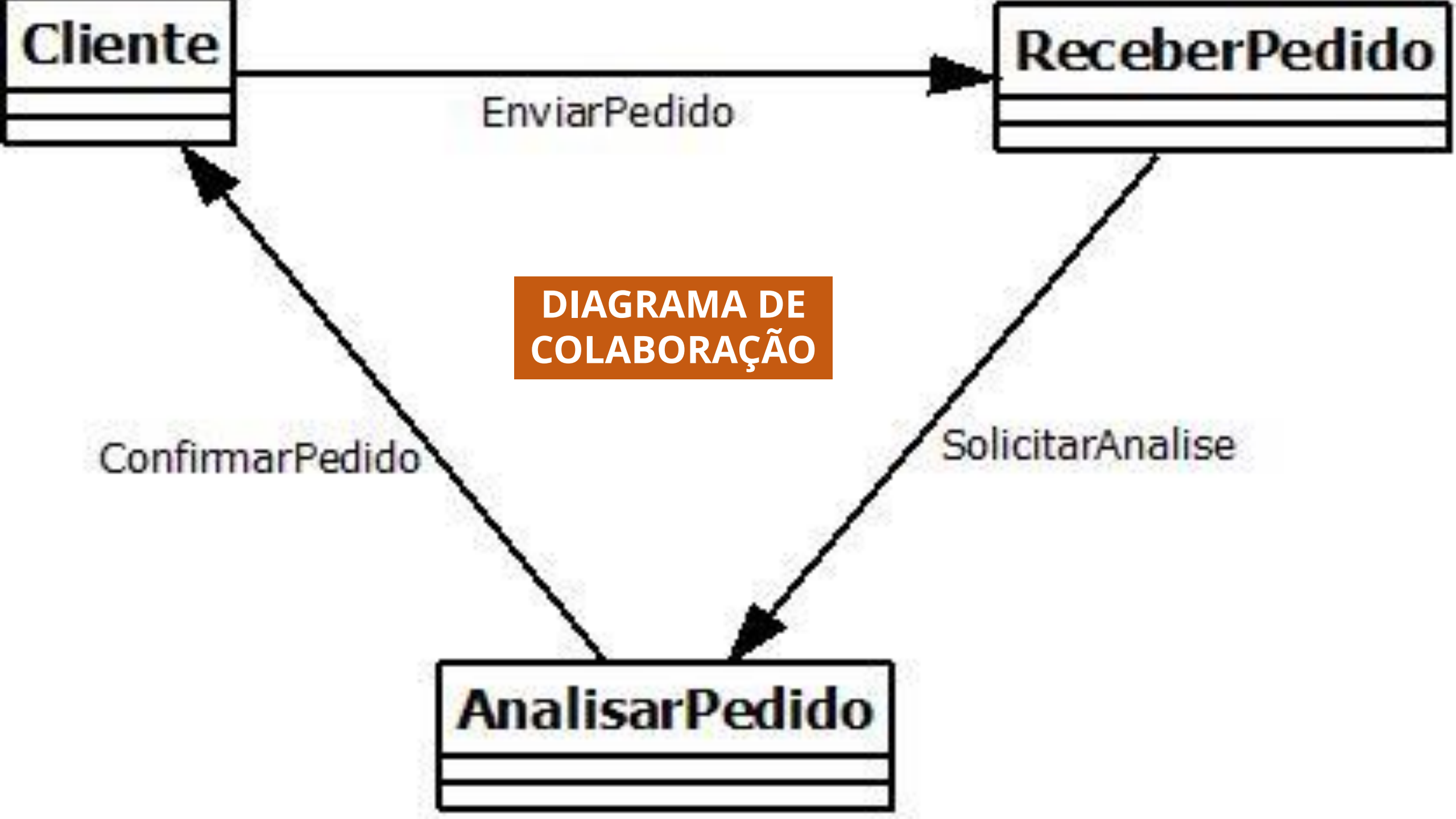
OUTROS DIAGRAMAS

- **Diagrama de Interação**: Os diagramas de interação detalham mais a fundo a relação entre os objetos do sistema, descrevendo as ações que ocorrem dentro dessa relação. Tem uma abordagem mais profunda em relação ao diagrama de caso de uso, com indicações mais genéricas.
- **Diagrama de Sequência:** detalham a sequência dos acontecimentos de um processo do sistema durante todo o lifecycle.
- **Diagrama de Colaboração**: Tem foco no relacionamento entre os atores no cenário. Não há relações com o tempo, apenas com as ações

DIAGRAMA DE INTERAÇÃO




Cliente
Funcionário
Filme
Locação
Solicitar Filme
Tem no estoque
Sim
Decrementa estoque
Calcula Devolução
Data Devolução
Cria Locação
Confirmar Locação
DIAGRAMA DE SEQUÊNCIA


Cliente
ReceberPedido
EnviarPedido
DIAGRAMA DE COLABORAÇÃO
ConfimarPedido
SolicitarAnalise
AnalisarPedido

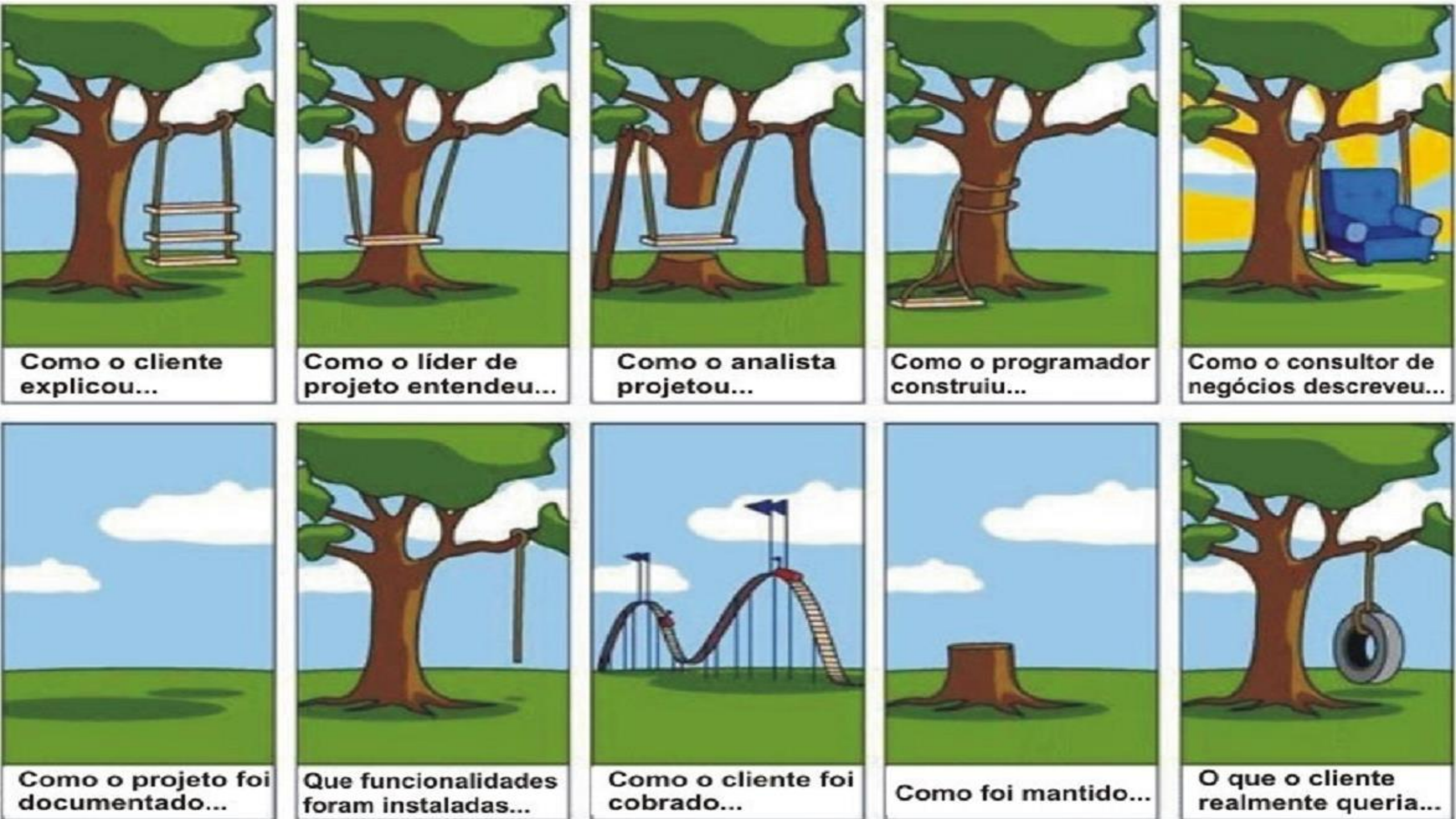
ANÁLISE E TESTE DE SOFTWARE

ANÁLISE ORIENTADA A OBJETOS

A atividade de análise é essencial para a construção de um software. Análise da viabilidade para implantação, análise da complexidade do software, análise de mercado, análise das tecnologias disponíveis. Tudo é válido. A partir da análise, começa-se efetivamente a pensar no desenvolvimento.

A maioria dos erros em software são originários de uma análise fraca. É a fase que necessita de mais atenção e ao mesmo tempo, é a mais ignorada. O cliente não sabe o que é possível fazer, pois ele não analisa, apenas pede. E ainda pode pedir errado. Ou então, não sabe o que pedir.

Analisar, pensando desde o início no paradigma dos objetos, pode facilitar a compreensão e a construção.


Como o cliente explicou...
Como o líder de projeto entendeu...
Como o analista projetou...
Como o programador construiu...
Como o consultor de negócios descreveu...
Como o projeto foi documentado...
Que funcionalidades foram instaladas...
Como o cliente foi cobrado...
Como foi mantido...
O que o cliente realmente queria...

ANÁLISE ORIENTADA A OBJETOS

A atividade de análise é essencial para a construção de um software. Análise da viabilidade para implantação, análise da complexidade do software, análise de mercado, análise das tecnologias disponíveis. Tudo é válido. A partir da análise, começa-se efetivamente a pensar no desenvolvimento.

A maioria dos erros em software são originários de uma análise fraca. É a fase que necessita de mais atenção e ao mesmo tempo, é a mais ignorada. O cliente não sabe o que é possível fazer, pois ele não analisa, apenas pede. E ainda pode pedir errado. Ou então, não sabe o que pedir.

Analisar, pensando desde o início no paradigma dos objetos, pode facilitar a compreensão e a construção.

QUESTIONÁRIOS

OBSERVAÇÕES IN LOCO

CASOS DE USO

ENTREVISTAS

PESQUISAS

CENÁRIOS

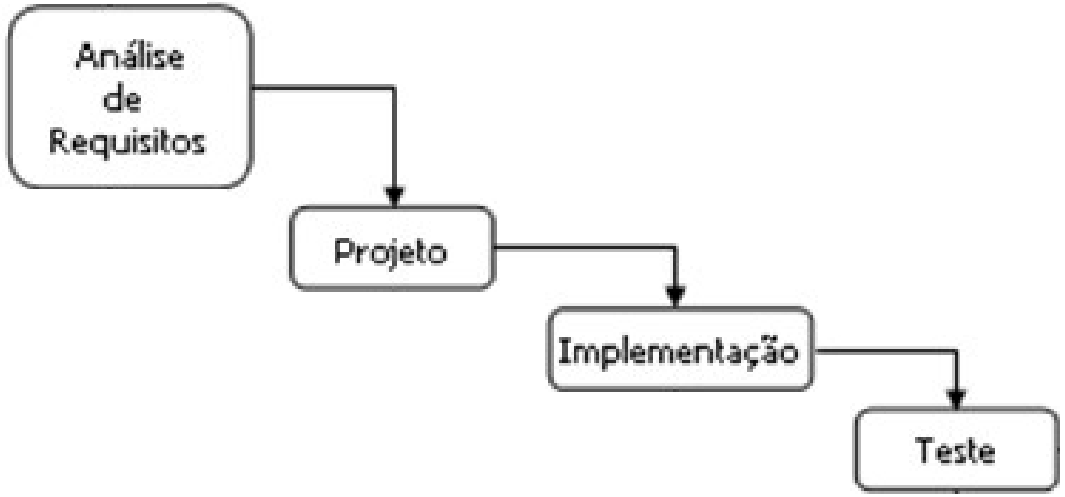
PROTOTIPAÇÃO

Várias ferramentas de análise são possíveis, pois vários problemas podem surgir:

- Cliente especifica requisitos confusos;
- Projeto mal elaborado;
- Conhecimento da equipe;
- Tecnologia difícil ou insuficiente;
- Testes insuficientes

METODOLOGIA EM CASCATA

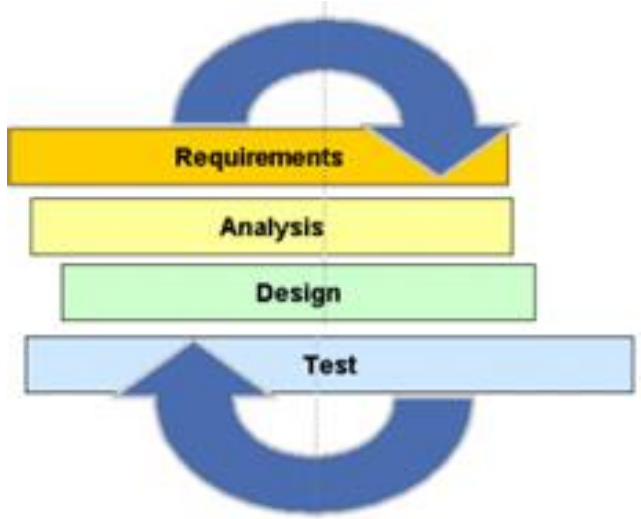
A metodologia em cascata é concebida em etapas necessárias para completar o desenvolvimento do software. Cada etapa é bem definida e só será avançada quando estiver totalmente concluída. Ao avançar de etapa, não é permitido voltar para os passos anteriores pois assume-se que eles estejam corretos. É um processo que tenta evitar mudanças no software, durante o desenvolvimento.



METODOLOGIA ITERATIVA

Os passos são os mesmo de um processo em cascata, mas diferencia-se pela revisão constante de todas as etapas, durante o desenvolvimento. Se uma coisa não está indo bem, o melhor a fazer é retornar as etapas anteriores e recomeçar. O processo iterativo favorece a mudança constante do projeto e evolução dos requisitos. Não serve somente para correção de erros. Isto aumenta a confiabilidade do software, diminui seu custo e também o tempo de entrega.

A construção do software dá-se na verdade por um conjunto de várias iterações. É um processo incremental. A cada nova iteração tem-se um avanço na qualidade e capacidade do software.



E A ETAPA DE TESTE?

“A atividade de teste nunca termina. Apenas é transferida do projetista para seu cliente. Um software de qualidade não contém erros. Um testador profissional pode realizar testes mais completos e portanto descobrir o maior número de erros possíveis antes que os testes do cliente se iniciem.”

“Errar é humano, mas pra realmente estragar tudo, você precisa de um computador” Paul R. Ehrlich

POR QUE TESTAR?

- **Engenharia de Software**: testar é importante para aumentar a qualidade do software ou serviço, conforme diretrizes da Engenharia de Software.
- **Pessoas são propensas a erros**: atividade de desenvolvimento software ou serviço são executadas por pessoas, que por mais dedicadas e capacitadas, cometem falhas. A etapa de teste de software busca justamente identificar essas falhas.
- **Importância acadêmica e prática**: o teste de software continua a ser um tópico estudado e desenvolvido, para atender cada vez mais a indústria e serviços que prezam pela qualidade de software.
- **Valor Agregado:** software testado é software mais valioso.

PRA QUE TESTAR?

O objetivo de testes é revelar defeitos no software. Um teste bem sucedido sempre irá revelar um erro ainda não descoberto. Se não, porque você testou?

Importante notar que testes revelam a presença de defeitos, mas não sua ausência.

Somente o teste exaustivo seria capaz de mostrar que um software é livre de defeitos, contudo, esse teste é impossível. Por isso devemos priorizar testes, o que chamamos de **Caso de Teste,** considerando funcionalidades e situações.

CASO DE TESTE = ENTRADAS + SAÍDA ESPERADA

10 DESASTRES FAMOSOS

1. Lançamento Mariner 1 (1962)

O foguete Mariner 1 com uma sonda espacial para Vênus foi desviado de sua rota de voo pretendida logo após o lançamento. O Controle da Missão destruiu o foguete 293 segundos após o lançamento.

US$ 18 milhões (US$ 180 milhões corrigidos)

Por quê? Um programador transcreveu incorretamente uma fórmula manuscrita em código de computador, desconsiderando uma barra sobrescrita. $\dot{\bar{R}}_n$

10 DESASTRES FAMOSOS

2. Hartford Coliseum (1978)

Apenas algumas horas depois de milhares de fãs terem saído do Hartford Coliseum, o teto de treliça de aço desmoronou sob o peso da neve molhada.

US$ 90 milhões (US$ 370 milhões corrigidos)

Por quê? O programador do software CAD assumiu incorretamente que os suportes do telhado enfrentariam apenas uma compressão pura. Quando um dos suportes se soltou da neve, desencadeou uma reação em cadeia.

10 DESASTRES FAMOSOS

3. Alarme Nuclear Soviético Falso (1983)

O sistema soviético de alerta indicou falsamente que os EUA haviam lançado cinco mísseis balísticos. Felizmente, o oficial de serviço soviético raciocinou que se os EUA estivessem realmente atacando teriam lançado mais de cinco mísseis, e considerou falso alarme.

Quase a 3ª Guerra Mundial!

Por quê? Um bug no software soviético não conseguiu filtrar falsas detecções de mísseis causadas pela luz do sol refletindo nas nuvens.

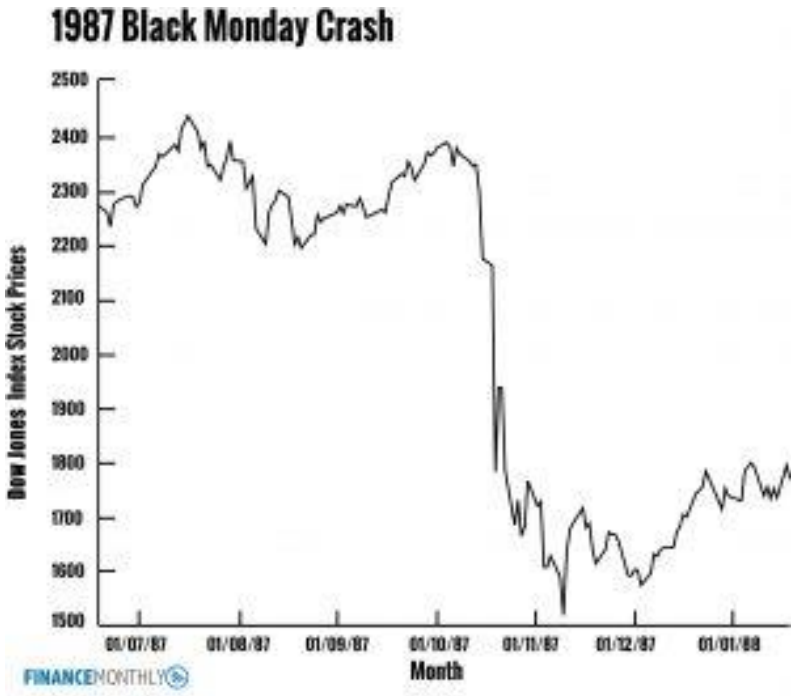
10 DESASTRES FAMOSOS

4. Crash em Wall Street (1987)

19/10/1987 foi o “Black Monday”, o índice Dow Jones despencou 508 pontos, perdendo 22,6% de seu valor.

US$ 500 bilhões em um dia!

Por quê? Os computadores compram e vendem ações automaticamente com base em algoritmos definidos por empresas de negociação de ações. Quando o clima de um mercado muda, esses programas de computador reagem de acordo com um conjunto de equações matemáticas. Porém, uma saída planejada resultou em um excesso de vendas.



10 DESASTRES FAMOSOS

5. Falha nas linhas telefônicas da AT&T (1990)

Um único switch em um dos 114 centros de comutação da AT&T sofreu um pequeno problema mecânico e desligou o centro. Após retornar, ele enviou uma mensagem para outros centros de comutação, o que fez com que eles desligassem e derrubassem toda a rede da AT&T por 9 horas.

75 milhões de chamadas e 200 mil reservas de voos perdidas

Por quê? Uma única linha de código com bug, após uma atualização de software implementada para acelerar as chamadas causou um efeito cascata que desligou a rede.

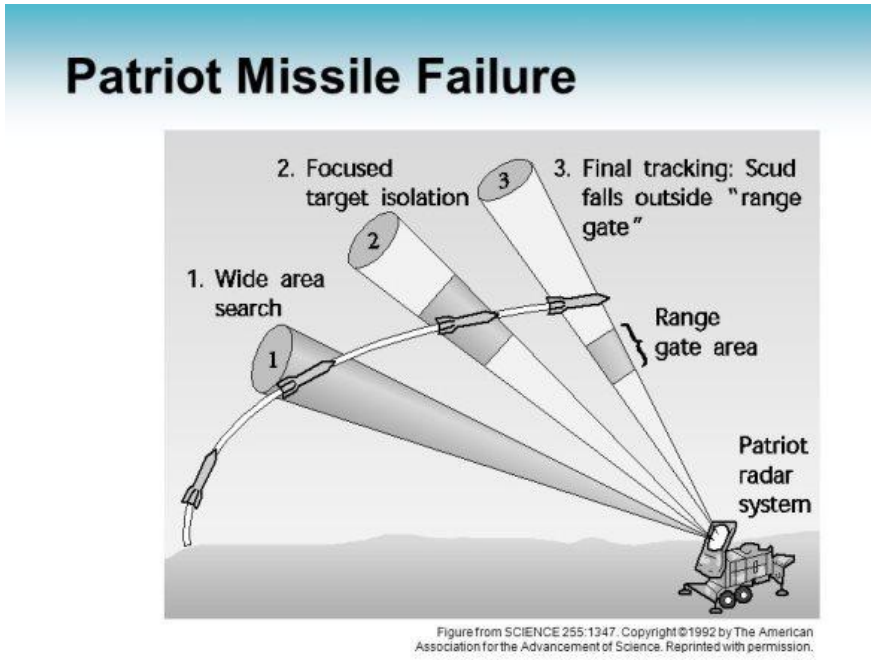
10 DESASTRES FAMOSOS

6. Falha do sistema Patriot (1991)

Durante a primeira Guerra do Golfo, um sistema americano de mísseis Patriot na Arábia Saudita não conseguiu interceptar um novo míssil Scud iraquiano. O míssil destruiu um quartel do exército americano.

28 soldados mortos e 100 feridos.

Por quê? Um erro de arredondamento de software calculou incorretamente o tempo, fazendo com que o sistema Patriot ignorasse o míssil Scud de entrada.



10 DESASTRES FAMOSOS

7. Falha de Divisão do Pentium (1993)

O famoso chip Pentium da Intel ocasionalmente cometeu erros ao dividir números de ponto flutuante em um intervalo específico. Ex: dividir 4195835.0 / 3145727.0 produziu 1,33374 em vez de 1,33382, um erro de 0,006%. Estimativa de 5 milhões de chips defeituosos em circulação.

US$ 475 milhões + queda na credibilidade.

Por quê? O divisor na unidade de ponto flutuante Pentium tinha uma tabela de divisão defeituosa, faltando cerca de cinco entre mil entradas, resultando nesses erros de arredondamento.

10 DESASTRES FAMOSOS

8. Ariane 5 (1996)

O foguete europeu Ariane 5 foi intencionalmente destruído alguns segundos após o lançamento em seu primeiro voo. Também foi destruída a carga de quatro satélites científicos para estudar como o campo magnético da Terra interage com os ventos solares.

US$ 500 milhões.

Por quê? O desligamento ocorreu quando o computador de orientação tentou converter a velocidade de foguete lateral de 64 bits para um formato de 16 bits. O número era muito grande e ocorreu um erro de estouro. Quando o sistema de orientação foi desligado, o controle passou para uma unidade redundante idêntica, que também falhou porque estava executando o mesmo algoritmo.

10 DESASTRES FAMOSOS

9. Satélite Mars Climate (1998)

Depois de uma jornada de 286 dias da Terra, o Mars Climate Orbiter disparou seus motores para entrar em órbita ao redor de Marte. Os motores dispararam, mas a espaçonave caiu muito na atmosfera do planeta, provavelmente causando a colisão em Marte.

US$ 125 milhões.

Por quê? O software que controlava os propulsores da Orbiter usava unidades imperiais (libras de força), em vez de unidades métricas (Newtons), conforme especificado pela NASA.

10 DESASTRES FAMOSOS

10. Bug do Milênio (1999)

O desastre de um é a fortuna de outro. As empresas gastaram bilhões em programadores para consertar uma falha no software legado. Embora não tenham ocorrido falhas significativas no computador, a preparação para o bug Y2K teve um impacto significativo em termos de custo e tempo em todas as indústrias que usavam computadores.

US$ 500 bilhões.

Por quê? Para economizar espaço de armazenamento do computador, o software legado geralmente utilizava o ano em apenas dois dígitos, como “99” para 1999. O software também interpretou “00” como 1900 em vez de 2000, então com a chegada do ano 2000, temia-se um colapso geral.


A HISTÓRIA DO
BUG DO MILÊNIO

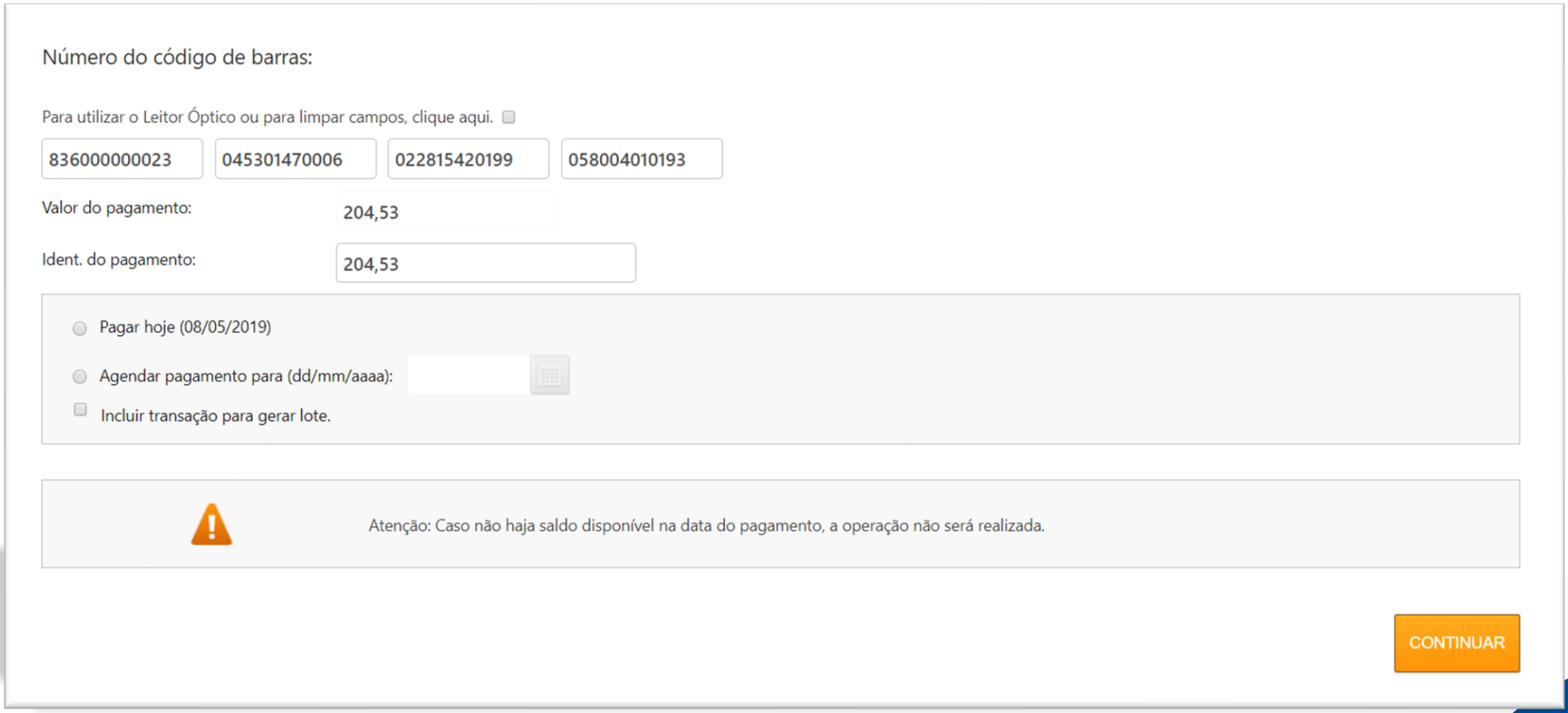
FALHAS EM SOFTWARES DIVERSOS

Certa vez, foi detectado um bug excêntrico na Caixa. Um usuário ao tentar pagar um boleto, erra a senha. Ao tentar novamente (se for poucos segundos depois), mesmo acertando a senha o sistema nega o pagamento, indicando que já há uma transação de pagamento para tal boleto!

Após o erro, o sistema volta para a tela inicial,
com o usuário precisando preencher tudo novamente.

Qual o erro?

Essa mensagem só deveria aparecer quando a
transação de pagamento realmente iniciar, isto é, se realmente ocorreu o pagamento.


Número do código de barras:
Para utilizar o Leitor Óptico ou para limpar campos, clique aqui.
836000000023
045301470006
022815420199
058004010193
Valor do pagamento:
204,53
Ident. do pagamento:
204,53
Pagar hoje (08/05/2019)
Agendar pagamento para (dd/mm/aaaa):
Incluir transação para gerar lote.
Atenção: Caso não haja saldo disponível na data do pagamento, a operação não será realizada.
CONTINUAR

Confira os dados informados, digite abaixo a assinatura eletrônica de sua conta, selecione CONFIRMAR e aguarde o comprovante.

Representação numérica do código de barras:	836000000023 045301470006 022815420199 058004010193
Conta de débito:	0041 / 001 . 00072038-2
Data de débito:	08/05/2019
Descrição:	Conta de Energia Elétrica ou Gás
Valor:	204,53
Identificação do pagamento:	204,53

Digite sua assinatura eletrônica para efetuar a operação.

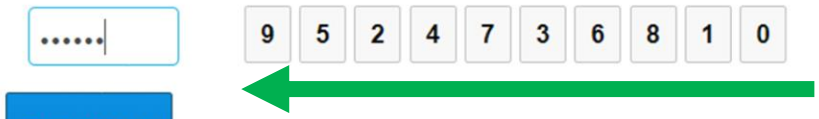
RETORNAR

CONFIRMAR

Confira os dados informados, digite abaixo a assinatura eletrônica de sua conta, selecione CONFIRMAR e aguarde o comprovante.

Representação numérica do código de barras:	836000000023 045301470006 022815420199 058004010193
Conta de débito:	0041 / 001 . 00072038-2
Data de débito:	08/05/2019
Descrição:	Conta de Energia Elétrica ou Gás
Valor:	204,53
Identificação do pagamento:	204,53

Digite sua assinatura eletrônica para efetuar a operação.

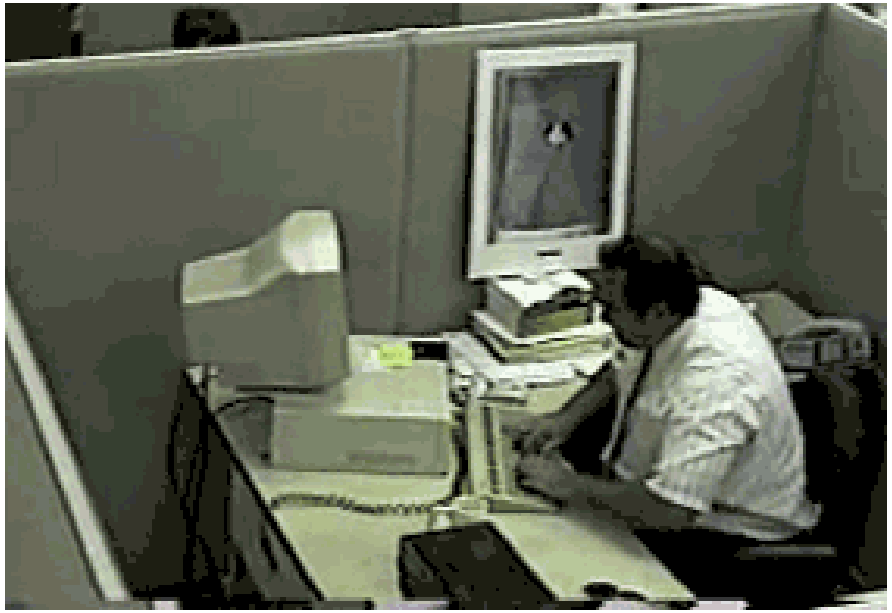


Digitando a senha corretamente!

RETORNAR

CONFIRMAR

Houve mais de uma tentativa de realizar esta transação em menos de 28 segundos, por favor cheque seu extrato para ver se a mesma já foi efetivada.



Solicitado por:
ELAINE CRISTINA DE SOUZA GOMES

Órgão do Solicitante:
Campus Garanhuns

Data da Solicitação:
02/08/2016

Número da PCDP:
002235/16

Nome do Proposto:
MAYARA DALLA LANA
(javascript:void(0);)

Tipo de Proposto:
Servidor

Período da Viagem:
16/08/2016 a 20/08/2016

Motivo da Viagem:
Nacional - Congresso

Viagem:
Nacional

Posição da PCDP no Fluxo:
Clique aqui (#)

Histórico:
Clique aqui ()

Justificativas:
Clique aqui (javascript:void(0);)

Bilhetes:
Clique aqui ()

Encaminhamentos:
Clique aqui ()

Viagem em Grupo:
Não

Curso Ministrado por Escola de Governo:
Não

Detalhes da PCDP:
Clique aqui (#)

Descrição do Motivo da Viagem:
Jogos Olímpicos e Paralímpicos de 2016 - Simultânea. Participação e apresentação de trabalho científico no III Reforest- Simpósio Nacional de Restauração Florestal.

RESERVAS PENDENTES DE EMISSÃO

		Localizador	Companhia Aérea	Trecho	Tentativas de Emissão	Última Tentativa
	☐	ME3JHQ	AZUL LINHAS AÉREAS BRASILEIRA	Recife (PE) - 16/08/2016 09:16 / Belo Horizonte (MG) - 16/08/2016 11:45	139	05/08/2016 11:06

Motivo da não emissão:
; nested exception is: java.net.SocketTimeoutException: Read timed out: AxisFault faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.userException faultSubcode: faultString: java.net.SocketTimeoutException: Read timed out faultActor: faultNode: faultDetail: {http://xml.apache.org/axis/}stackTrace:java.net.SocketTimeoutException: Read timed out at java.net.SocketInputStream.socketRead0(Native Method) at java.net.SocketInputStream.socketRead(SocketInputStream.java:116) at java.net.SocketInputStream.read(SocketInputStream.java:170) at java.net.SocketInputStream.read(SocketInputStream.java:141) at

Elmano Ramalho Cavalcanti
Último acesso: 10/12/2018 11:36
Sua sessão expira em 19min54s

- Agenda Médica Eletrônica
- Agenda Médico Família
- Alterar Data de Pagamento
- Atualização Cadastral
- Cartão de Identificação
- Consulta de Autorizações
- Dados Cadastrais (PIN-SS)
- Declarações
- Demonstrativo de Pagamento
- Demonstrativo para Imposto de Renda
- Emissão de 2ª Via de Boleto

CASSI
Caixa de Assistência dos Funcionários do Banco do Brasil

Erro no processamento!

CASSI

Ocorreu um erro no processamento!

[15995] Data/Hora: 2018-12-10T12:20:06

Favor entrar em contato com a CASSI ou tentar novamente mais tarde!

Agora você usará o seu CPF em vez de email para acessar os serviços online da CASSI. A senha já cadastrada continua válida. Se precisar obter acesso ou recuperar senha, use os botões que correspondem a estas opções.

Object reference not set to an instance of an object.

CPF

Informe seu CPF

Senha

••••••••••

Dúvidas no acesso

- Nunca acessei a área logada. O que fazer?
- Não consegui acesso. O que fazer?
- Esqueci minha senha. O que fazer?
- Não consigo recuperar minha senha. O que fazer?

Exception at /almoxarifado/relatorio/nota_fornecimento_pdf/user/69718/

Deprecated String Exception: 'xml parser error (bogus < or &) in paragraph beginning\n\'<para align="left" fontSize="8\"

Request Method:	GET
Request URL:	https://suap.ifpe.edu.br/almoxarifado/relatorio/nota_fornecimento_pdf/user/69718/
Exception Type:	Exception
Exception Value:	`Deprecated String Exception: 'xml parser error (bogus < or &) in paragraph beginning\n\'<para align="left" fontSize="8\''`
Exception Location:	/usr/lib/python2.5/site-packages/reportlab/platypus/paragraph.py in _setup, line 548
Python Executable:	/usr/bin/python
Python Version:	2.5.2
Python Path:	['/usr/lib/python2.5/site-packages/pip-1.2.1-py2.5.egg', '/usr/lib/python2.5', '/usr/lib/python2.5/plat-linux2', '/usr/lib/python2.5/lib-tk', '/usr/lib/python2.5/lib-dynload', '/usr/local/lib/python2.5/site-pa '/usr/lib/pymodules/python2.5', '/var/www', '/var/www/suap']
Server time:	Sex, 16 Nov 2018 17:13:18 -0300

Traceback Switch to copy-and-paste view

/usr/lib/python2.5/site-packages/django/core/handlers/base.py in get_response

```
86.                 response = callback(request, *callback_args, **callback_kwargs)
```

▶ Local vars

/var/www/suap/almoxarifado/relatorio.py in nota_fornecimento_pdf

```
648.     cabecalho = pdf.table(cabecalho, a=['l', 'l'],w=[32, 138], grid=0)
```

▶ Local vars

/nfs/suap/djtools/pdf.py in table

```
204.                 cell = para(cell, **para_args)
```

▶ Local vars

/nfs/suap/djtools/pdf.py in para

```
125.                     caseSensitive=caseSensitive)
```

▶ Local vars

/usr/lib/python2.5/site-packages/reportlab/platypus/paragraph.py in __init__

```
523.         self._setup(text, style, bulletText, frags, cleanBlockQuotedText)
```

▶ Local vars

Pesquisador

Avisos e Pendências
Propostas e Pedidos
Relatório Técnico e Prestação de Contas
Seu Currículo Lattes
Consultoria Ad hoc
Termos de Concessão
Gerenciamento de Projetos
Informar CPF
Solicitação de Mudanças
Execução do Projeto
Substituição de Bolsista PDJ

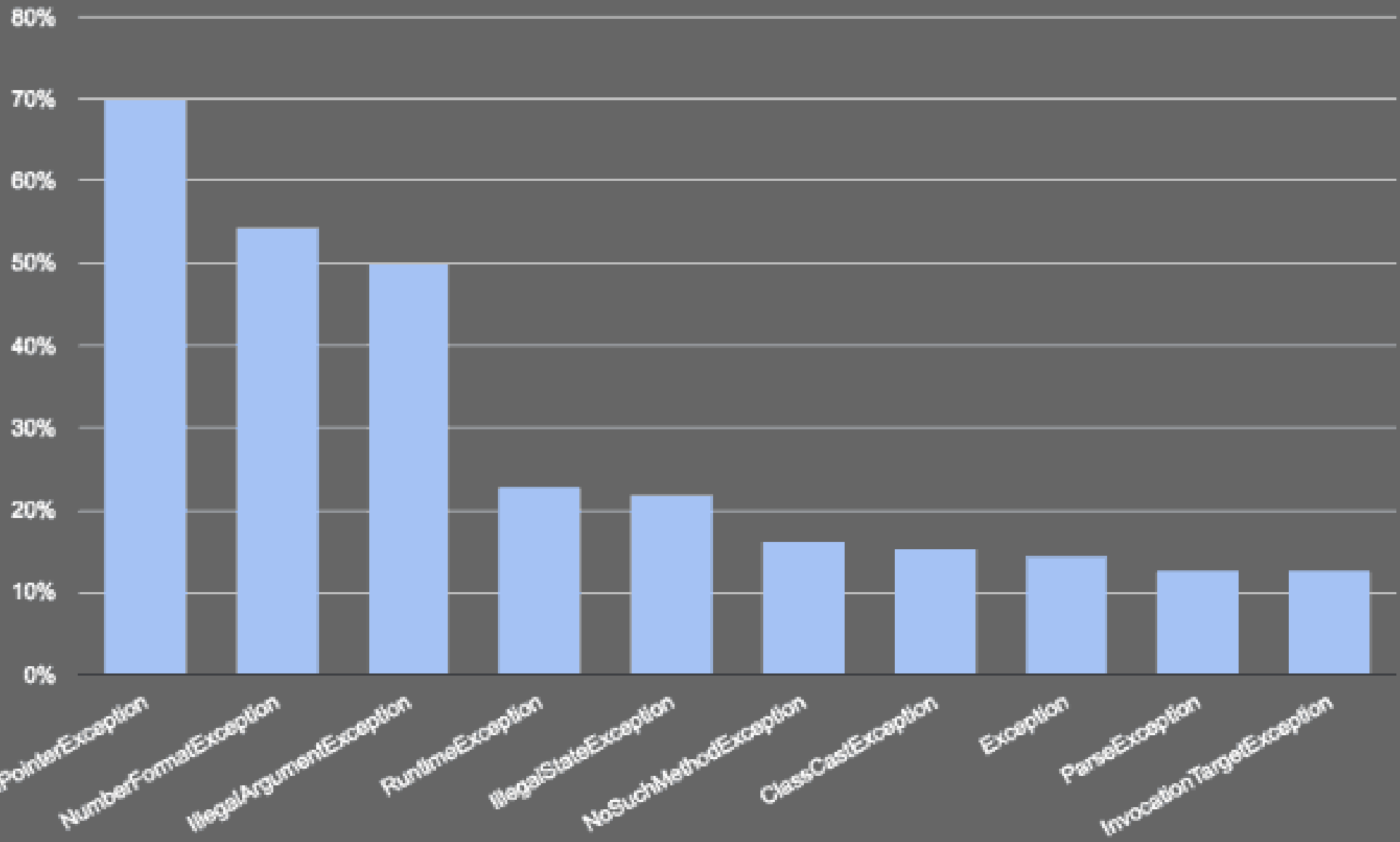
Definir como página inicial

Ocorreu um erro imprevisto.

- java.lang.NullPointerException

Para voltar a tela anterior, clique aqui.

Top 10 Exception Types by Frequency in 1,000+ Applications



Menu Completo

- Meu menu
- Conta-corrente
- Confirmações Pendentes
- Poupança
- Pagamentos
- Transferências
- Cartões
- Empréstimo

Problema na execução de sua solicitação...
S054C - Erro na Consulta - BASE_ (G999-805)

RETORNAR

Plataforma Lattes

Notícias

HTTP Status 503 - Servlet jsp is currently unavailable

type Status report

message Servlet jsp is currently unavailable

description The requested service (Servlet jsp is currently unavailable) is not currently available.

JBoss Web/7.0.13.Final

Acesso direto

Currículo Lattes

- Buscar currículo
- Atualizar currículo
- Cadastrar novo currículo

Diretório de Instituições

- Buscar instituição
- Atualizar instituição
- Cadastrar instituição

Diretorio dos Grupos de Pesquisa

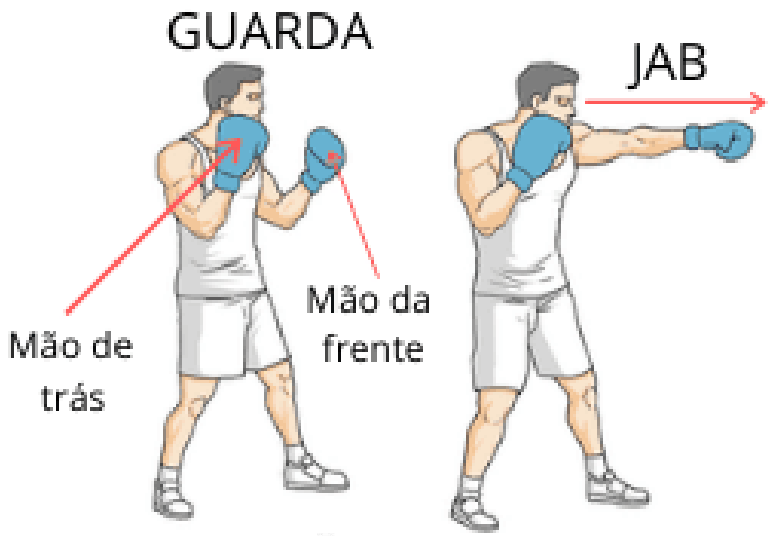
- Acessar o portal do Diretório

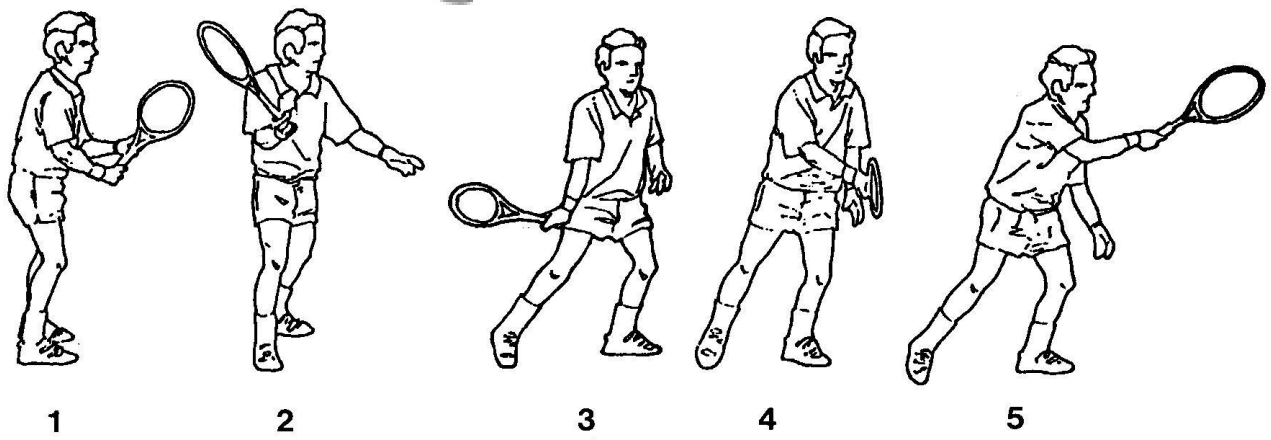
Painel Lattes

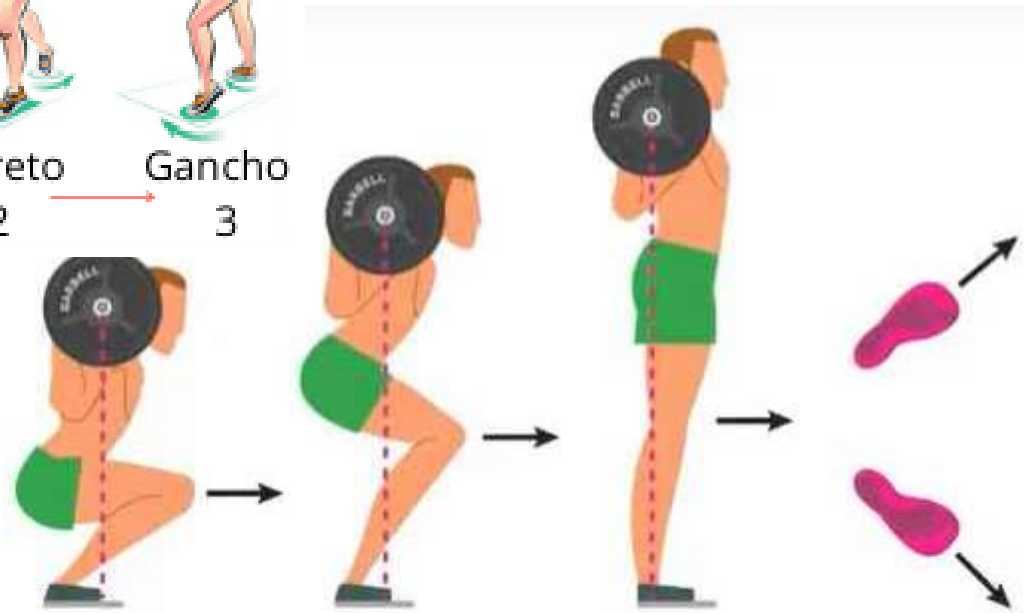
- Distribuição Geográfica
- Comparativo de Instituições
- Evolução na formação
- Todos os gráficos


BUGS
BUGS EVERYWHERE

POR QUE UTILIZAR TÉCNICAS DE TESTE?

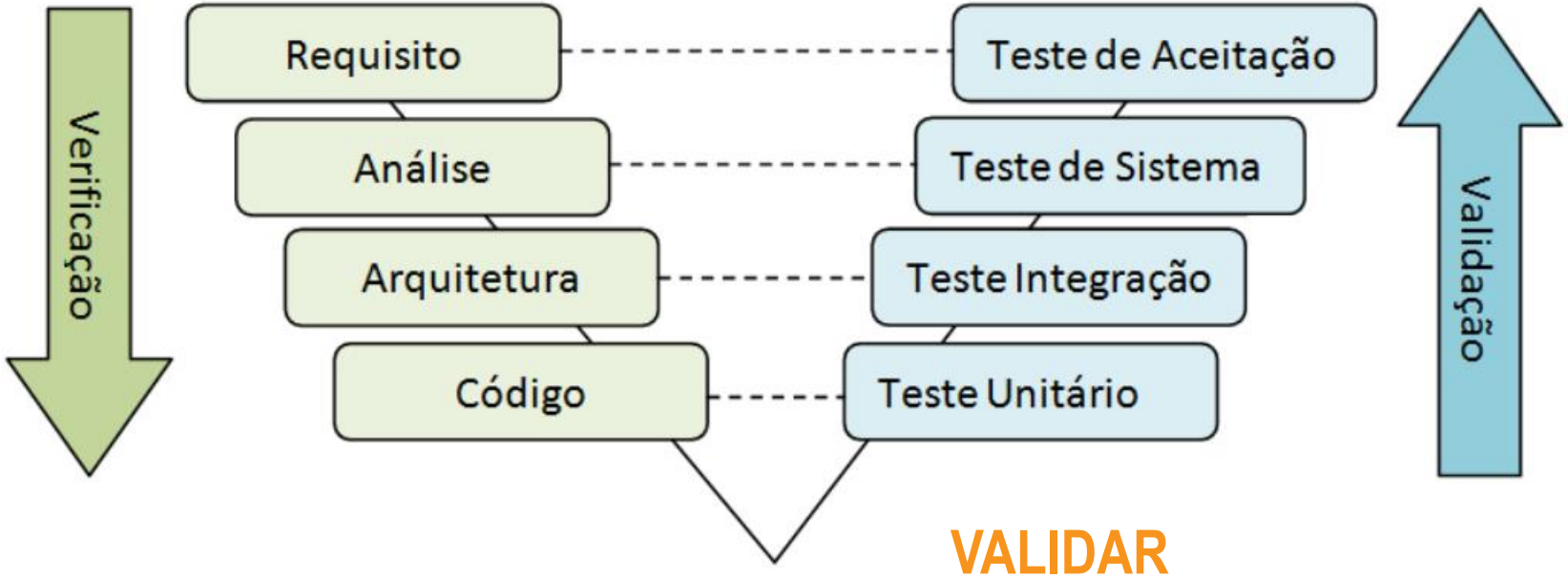






VALIDAR E VERIFICAR

Este é um tópico da Engenharia de Software para prover conhecimento de modo que o desenvolvimento do software atenda aos requisitos e também as necessidades dos usuários.



VERIFICAR

Estamos construindo certo o produto?

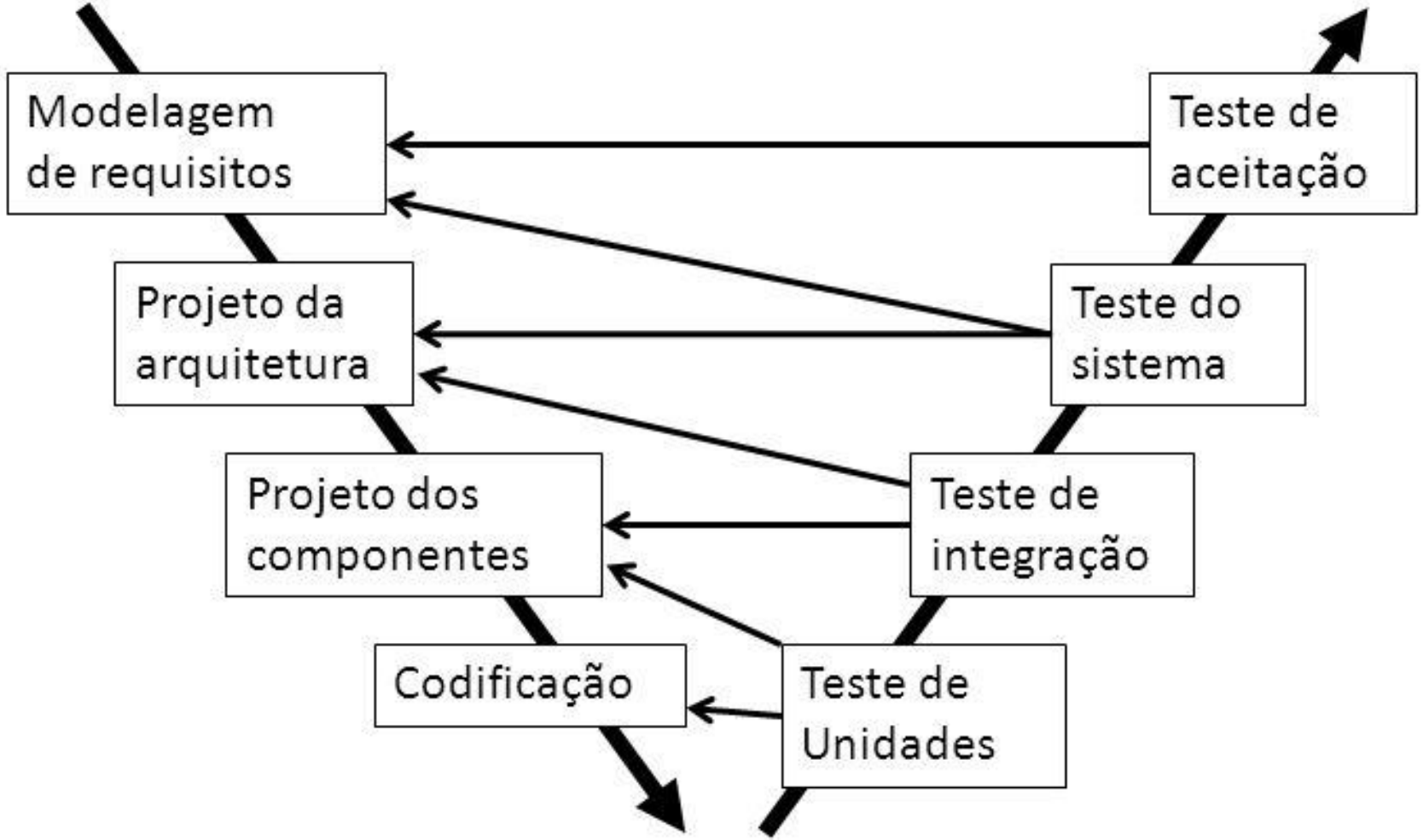
O software aos requisitos especificados?

VALIDAR

Estamos construindo o produto certo?

O software é o que usuário deseja?

QUANDO TESTAR?

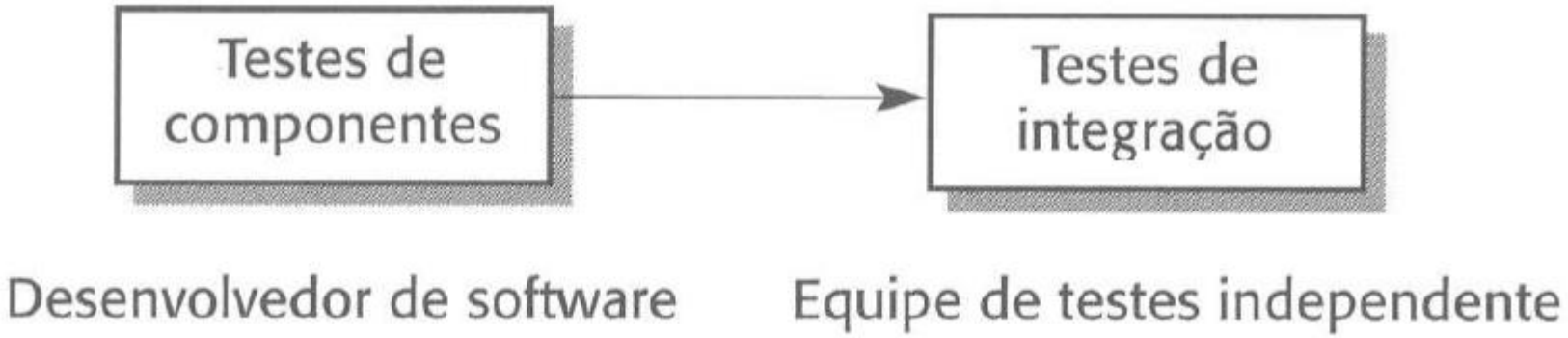


PROCESSOS DE TESTE

O processo de teste de software inicia-se no planejamento e definição dos testes, passando pela execução e encerrando-se com o relatório de teste.

Teste de Componentes: usualmente os programadores assumem a responsabilidade pelo teste de seu código. Isso faz com o que o teste dependa da experiência do programador.

Teste de Integração: uma equipe independente deve fazer o teste. Os testes são feitos baseados na especificação do sistema.



VIABILIDADE DE TESTES

Testar o software por completo é impossível, até mesmo porque não sabemos todas as variáveis envolvidas no sistema. Para que fosse possível, deveríamos testar todos as possibilidades de entradas e avaliar todas as saídas possíveis.

Imagine um software que realize a soma de 2 números inteiros

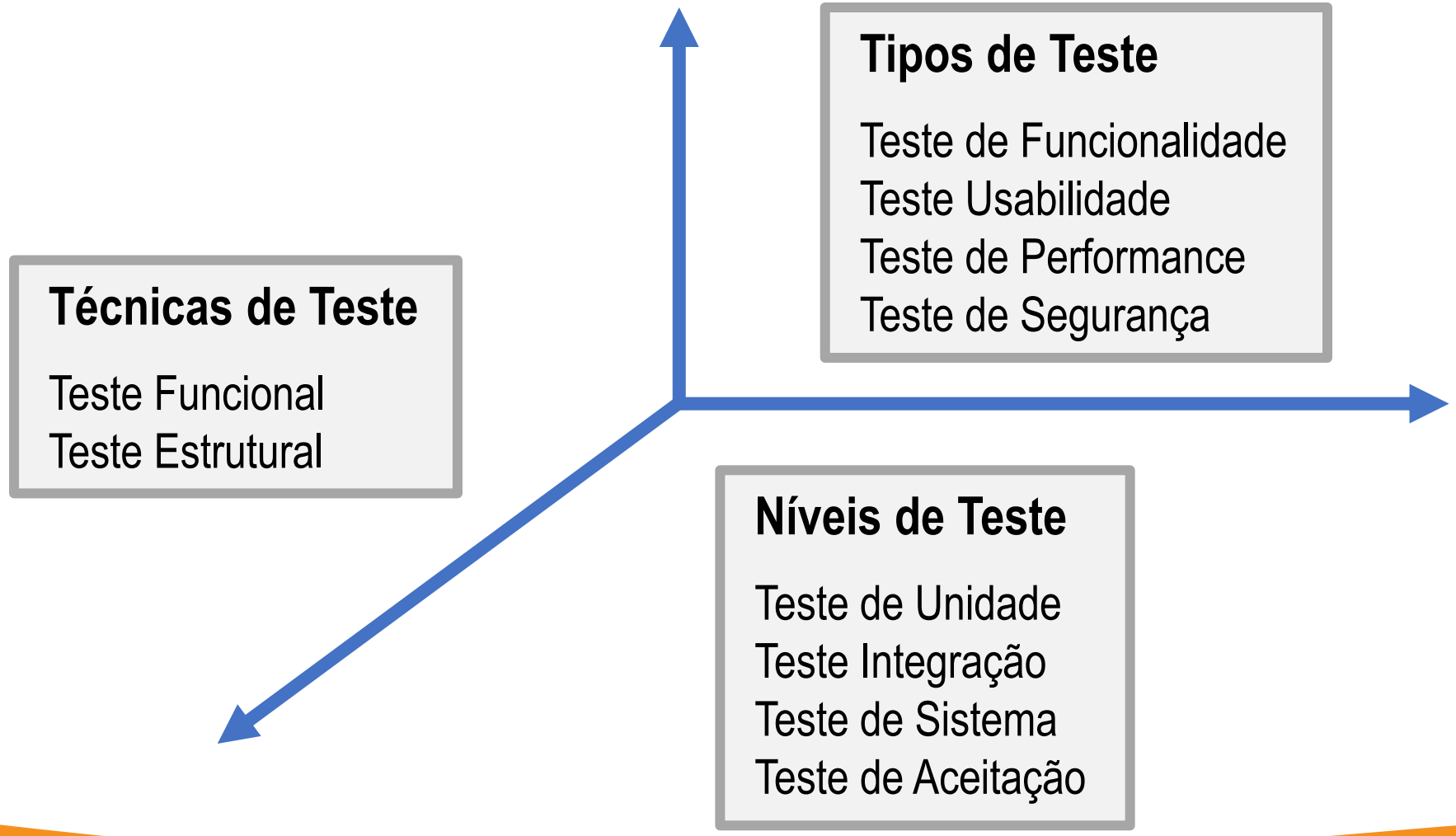
Para **garantir** que o software não tem nenhum erro, devemos testar todas as possibilidades de entrada de número inteiro.

Tamanho do tipo INT: 2.147.483.648

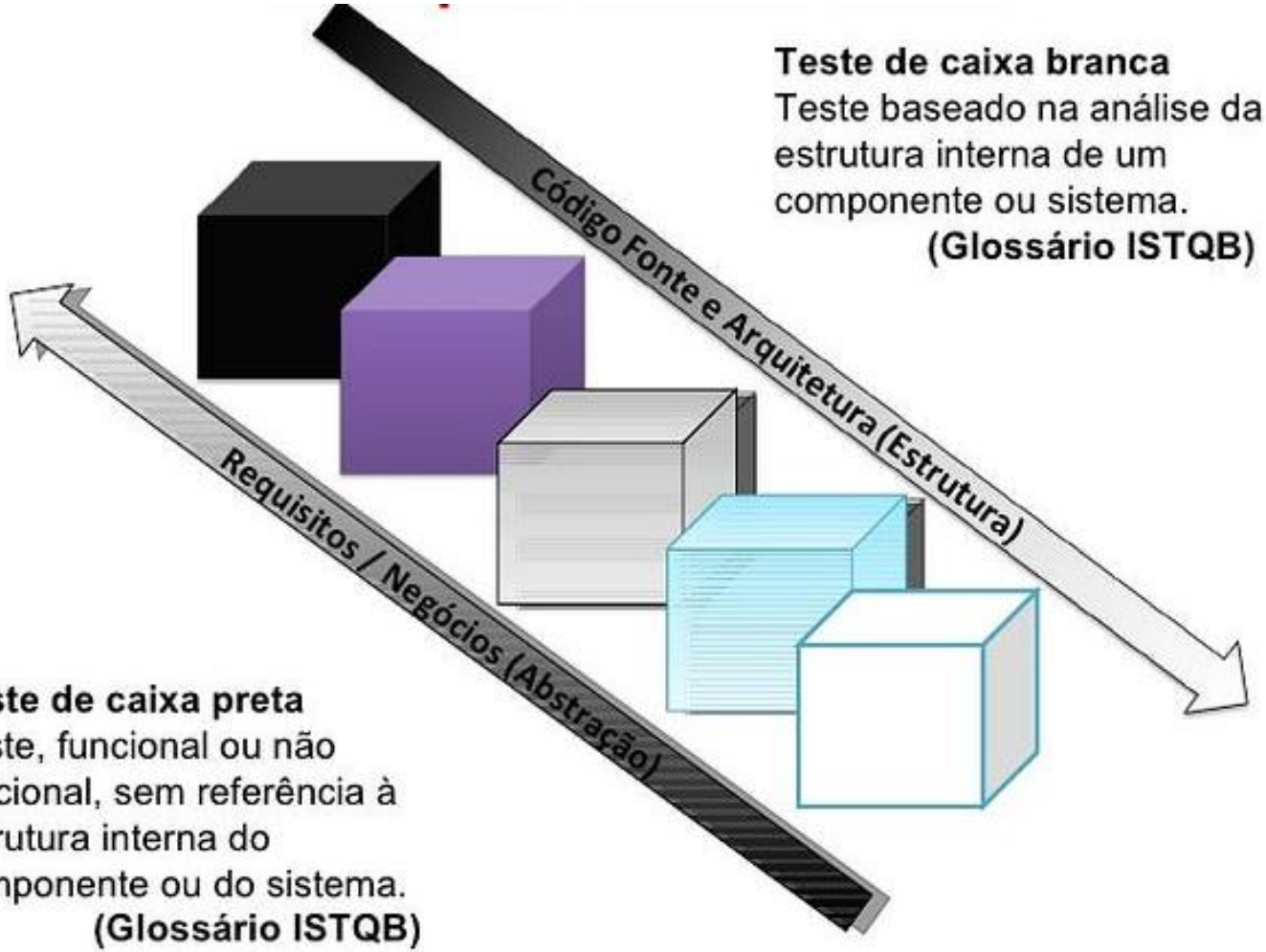
Possibilidades de cada campo: 4.294.967.296

Possibilidades Absolutas: 18 quintilhões

VISÃO GERAL - TESTE DE SOFTWARE



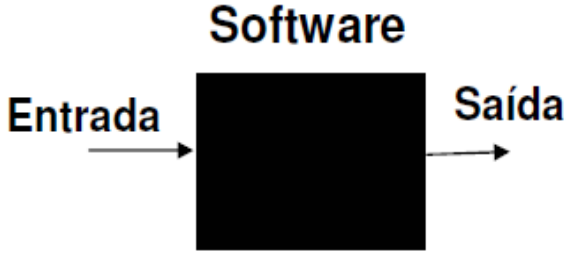
TESTES ESTRUTURAIS E FUNCIONAIS



Teste de caixa branca
Teste baseado na análise da estrutura interna de um componente ou sistema.
(Glossário ISTQB)

Teste de caixa preta
Teste, funcional ou não funcional, sem referência à estrutura interna do componente ou do sistema.
(Glossário ISTQB)

TESTE DE CAIXA PRETA



Quando não temos acesso ao código fonte, os testes são feitos em cima das entradas e saídas. As técnicas de caixa preta testam apenas a funcionalidade, em outras palavras, verifica se o resultado de um por exemplo está correto ou se uma mensagem foi exibida corretamente. O código, diretamente, não é testado, nem mesmo é acessado durante os teste de caixa preta.

- **Valores de Borda**: utilizar valores extremos **válidos** como nulos ou valores exorbitantes, etc;
- **Sintaxe**: testar entradas **inválidas** do sistema, por exemplo valores com ponto flutuante em campos inteiros;
- **Transição de estado**: testar a mudança de comportamento ou status de um componente, por exemplo, em um sistema de passagens, escolher assentos de um voo/ônibus e depois cancelá-los;

PARTICIONAMENTO EM CLASSES DE EQUIVALÊNCIA

Outra técnica de Caixa-Preta largamente utilizada em teste. Busca dividir o domínio de entrada em classes, projetando casos de testes para exercitar tanto as classes válidas quanto inválidas.

Por exemplo, um sistema que avalie se uma idade digitada é maior ou menor de idade, podendo receber números de 0 a 200. Quais são as classes **inválidas** e quais são **válidas**?

VALORES LIMITES

Com base no problema anterior, podemos também utilizar a técnica de Valor Limite. Consiste em testar os limites inferiores e superiores, com valores iguais, imediatamente acima e imediatamente abaixo. Inclusive podemos aplicar nas classes de equivalências.

1º Caso de Teste: -1 (Saída: valor inválido)

2º Caso de Teste: 0 (Saída: menor de idade)

3º Caso de Teste: 1 (Saída: menor de idade)

4º Caso de Teste: 199 (Saída: maior de idade)

5º Caso de Teste: 200 (Saída: maior de idade)

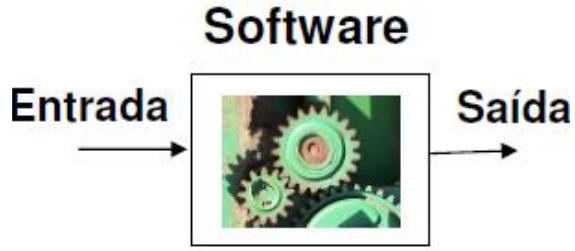
6º Caso de Teste: 201 (Saída: valor inválido)

TESTE DE STRESS

Executar o sistema além da carga máxima projetada. Os testes de estresse frequentemente provocam a revelação de defeitos no sistema. Durante os testes de estresse o sistema não deve falhar de maneira catastrófica, pois isso pode provocar corrupção de dados ou a perda inesperada de serviços. As exceções de stress devem ser idealmente tratadas no nível de código.

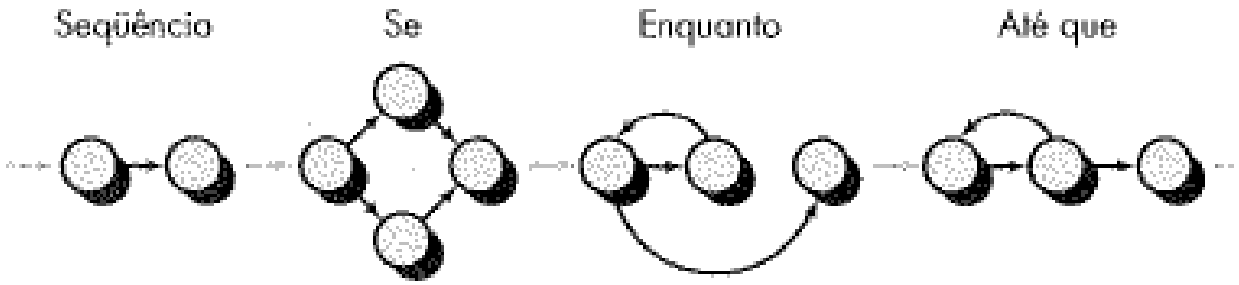
O teste de stress é largamente utilizado, por ser particularmente importante, em sistemas distribuídos que podem apresentar uma grande degradação à medida que a rede se torna sobrecarregada.

TESTE DE CAIXA BRANCA



Técnicas de caixa branca são baseadas na estrutura interna do programa e o testador precisa ter grande habilidade com programação, pois o objetivo é testar diretamente o código. É o mais trabalhoso e caro dos testes, por isso a exigência de grande capacidade da equipe testadora.

- **Fluxo de dados**: testa a lógica diretamente, em suas diferentes estruturas, observando os valores armazenados na memória. valores extremos válidos como nulos ou valores exorbitantes, etc;
- **Laços**: teste de laços em vários graus de complexidade, aninhados ou não.;

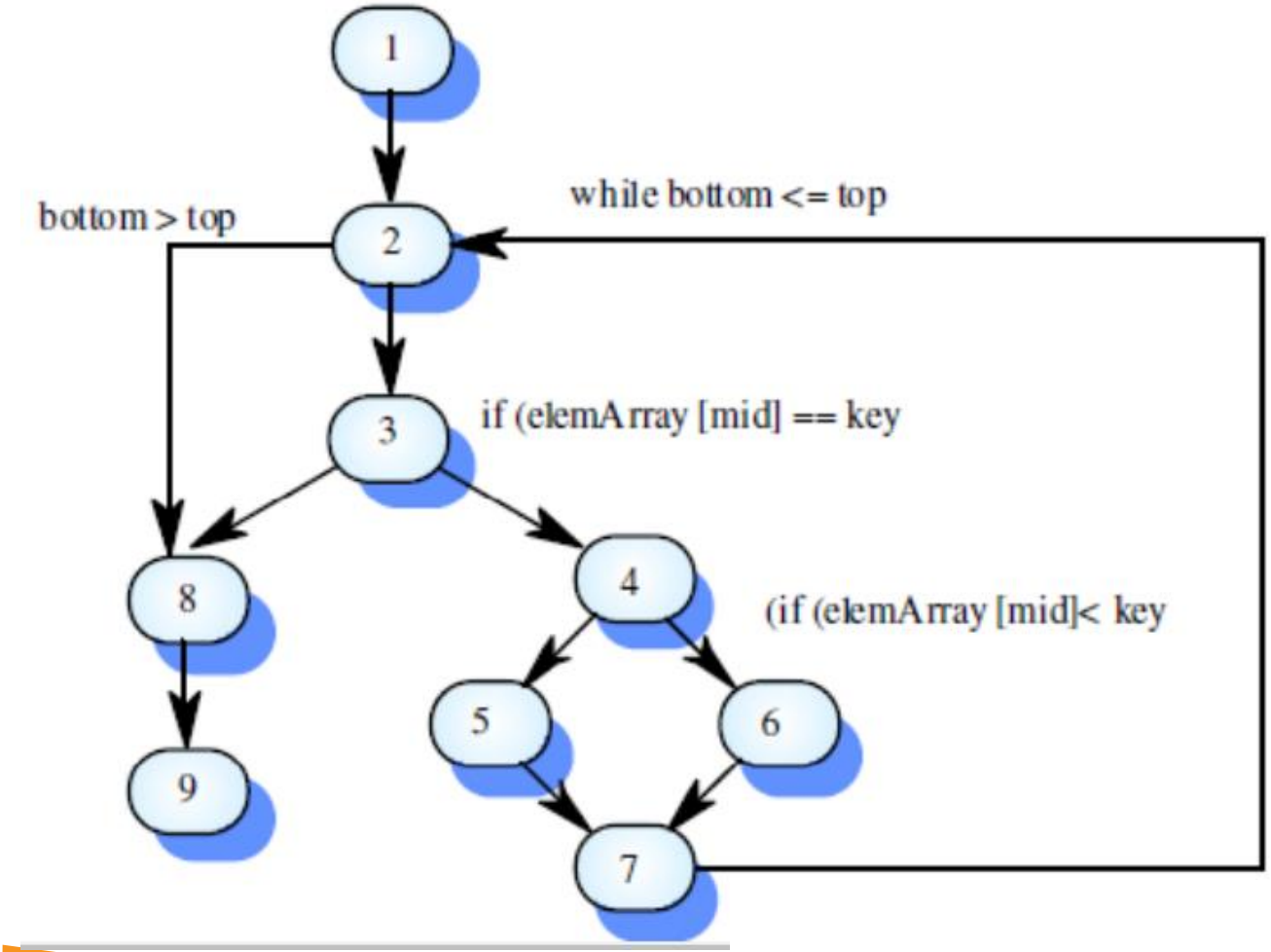


TESTE DE CAMINHO

É um teste de Caixa-Branca com o objetivo de garantir que o conjunto de testes é tal que cada caminho do programa é executado pelo menos uma vez. O ponto de partida para o teste de caminho é um grafo de fluxo do programa que mostra nós, representando decisões de programa e em arcos representando o fluxo de controle. Declarações condicionais são portanto nós no grafo de fluxo.

Um grafo de fluxo é uma ferramenta importante que mostra os caminhos lógicos que o software pode seguir. Cada ramo em uma declaração condicional é mostrado como um caminho independente e os loops são indicados por setas fazendo o loop de volta para o nó de condição do loop.

TESTE DE CAMINHO



Observe os caminhos possíveis
1,2,8,9
1,2,3,4,5,7,2
1,2,3,4,6,7,2
1,2,3,4,5,7,2,8,9
1,2,3,4,6,7,2,8,9

Casos de Teste devem ser propostos de forma a executar todos os caminhos. Algum analisador dinâmico pode ser utilizado para verificar se todos os caminhos foram executados e quais foram os resultados.

NÍVEIS DE TESTE

O nível de teste define o escopo, objetivo e executores do teste. Pode ser feito, do ponto de vista do desenvolvimento e do sistema, a nível local, regional ou global.

Teste Unitário: O tipo de teste de menor nível. Testa isoladamente cada componente (geralmente uma classe ou método) do sistema. Os testes são feitos diretamente no código e podem abranger diversas partes como a interface (passagem de dados), estruturas de dados e tratamento de exceções.

Teste de Integração: Testes de unidade não garantem o funcionamento pleno do sistema, já que testa apenas um componente por vez. O teste de integração visa integrar módulos isolados que já foram testados com sucesso para verificar a interoperabilidade do código. Obrigatório para interação de novos módulos.

NÍVES DE TESTE

Teste de Sistema: São complexos pois baseiam-se em testes globais, de como o sistema como um todo comporta-se. Inclui-se nesta etapa testes de carga (stress), de desempenho, recuperação e segurança. Pode-se também aplicar técnicas de ações aleatórias (para garantir que o sistema não funcione somente em determinada ordem) e teste de Banco de dados vazio (comum quando o desenvolvimento está no início).

Teste de Aceitação: É o teste de uma funcionalidade específica, visando atingir o nível de aceitação mínimo para ser liberada. É feito sempre no final de uma iteração, como forma de finalização daquela etapa. Recomenda-se que os testes de aceitação sejam feitos por um grupo de usuários finais restritos.

E COMO ESCREVER CASOS DE TESTE?

A escrita de Casos de Teste é uma das grandes habilidade de um profissional de qualidade de software. Os casos devem ser gerados de forma a abranger o máximo possível do sistema aliada a alta capacidade de encontrar uma falha. Há algumas técnicas que podem ajudar a formular os casos.

FATOS E PONTOS-CHAVE SOBRE TESTE DE SOFTWARE

- **Teste é uma atividade muito custosa**. Gasta-se muito tempo, esforço e dinheiro para fazer teste de software, sendo a etapa mais dispendiosa do desenvolvimento de software. Por isso deve ser planejada com técnicas e gerenciada para ser eficiente.
- **Teste bem sucedido descobre falhas**. Por isso, os testes devem ser os mais completos e abrangentes possíveis, estando bem planejados, para aumentar a chance de descobrir falhas. Um teste de caminho feliz (happy path) não deve ser parâmetro para métricas de qualidade de software. está no início).
- **Nenhum software é 100% testado**. Não é possível testar um software em sua totalidade, pois são inúmeras as possibilidades de entradas e ações do usuário.

FATOS E PONTOS-CHAVE SOBRE TESTE DE SOFTWARE

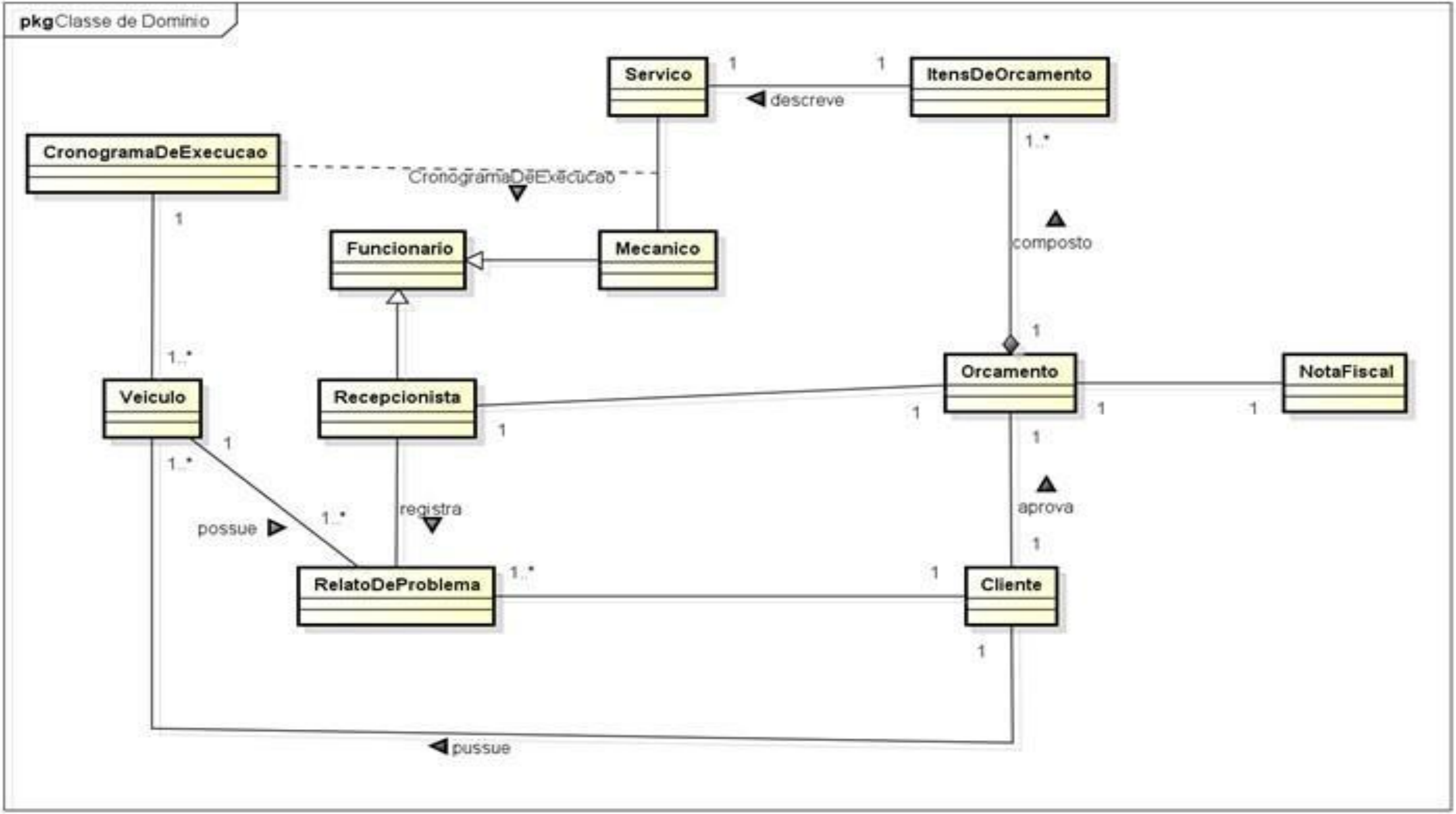
- **Teste as partes mais críticas**. Como não há tempo e viabilidade de testar um software por completo, os casos de teste devem ser direcionados para partes críticas, importantes e comumente utilizadas, em detrimento de partes com uso ocasional ou raro.
- **As técnicas mudam durante o processo**. O tipo, nível e técnicas de teste serão definidos, executados e adaptados ao longo do processo de desenvolvimento, em diferentes momentos, com as condições possíveis atuais. Não é ideal “viciar” o software em apenas um tipo de teste.

PROJETANDO SOLUÇÕES

Visto todas as ferramentas para entendimento do problema, podemos então projetar a solução. Um ótimo começo e projetar o domínio do problema. O domínio abrange todas as classes de representação, parecido com as soluções UML. Tudo deve ser incluso e tudo deve ser inicialmente tratado como classe.

Mais tarde, iremos refinando, adicionando, retirando. É comum ainda a solução estar um pouco confusa.

Exercício mental: como projetar um sistema de uma oficina mecânica?


pkgClasse de Dominio
Servico
ItensDeOrcamento
descreve
CronogramaDeExecucao
CronogramaDeExecucao
Funcionario
Mecanico
composto
Orcamento
NotaFiscal
Veiculo
Recepcionista
possue
registra
aprova
RelatoDeProblema
Cliente
pussue

DISTRIBUA AS RESPONSABILIDADES

Com todos os elementos do sistema mapeado, vamos separar as responsabilidades que irá se tornar na prática, os métodos. Precisamos então descobrir o propósito de cada objeto e “encaixá-lo” no sistema. Uma boa ferramenta para isso são os cartões **CRC (Class Responsibility Card)**.

São fichas de arquivo, onde organizaremos as responsabilidades dos objetos de nosso sistema. Você é limitado de forma intencional pelo tamanho do cartão, por isso mantenha a abstração. A baixa tecnologia é proposital para promover a interação.

Classes associadas

Classe: Conta Corrente	
Responsabilidade	**Colaboração**
Saber o seu saldo	Cliente
Saber seu cliente	Histórico de Transações
Saber seu número	
Manter histórico de transações	
Realizar saques e depósitos	

atributos

métodos

ANÁLISE INTENSA

Com os cartões CRC, surgirão mais claramente as interações entre os objetos e os relacionamento entre as classes. É o momento de definir as dependências, agregações, composições e generalizações do seu projeto. Analise também onde pode haver heranças e polimorfismos futuros.

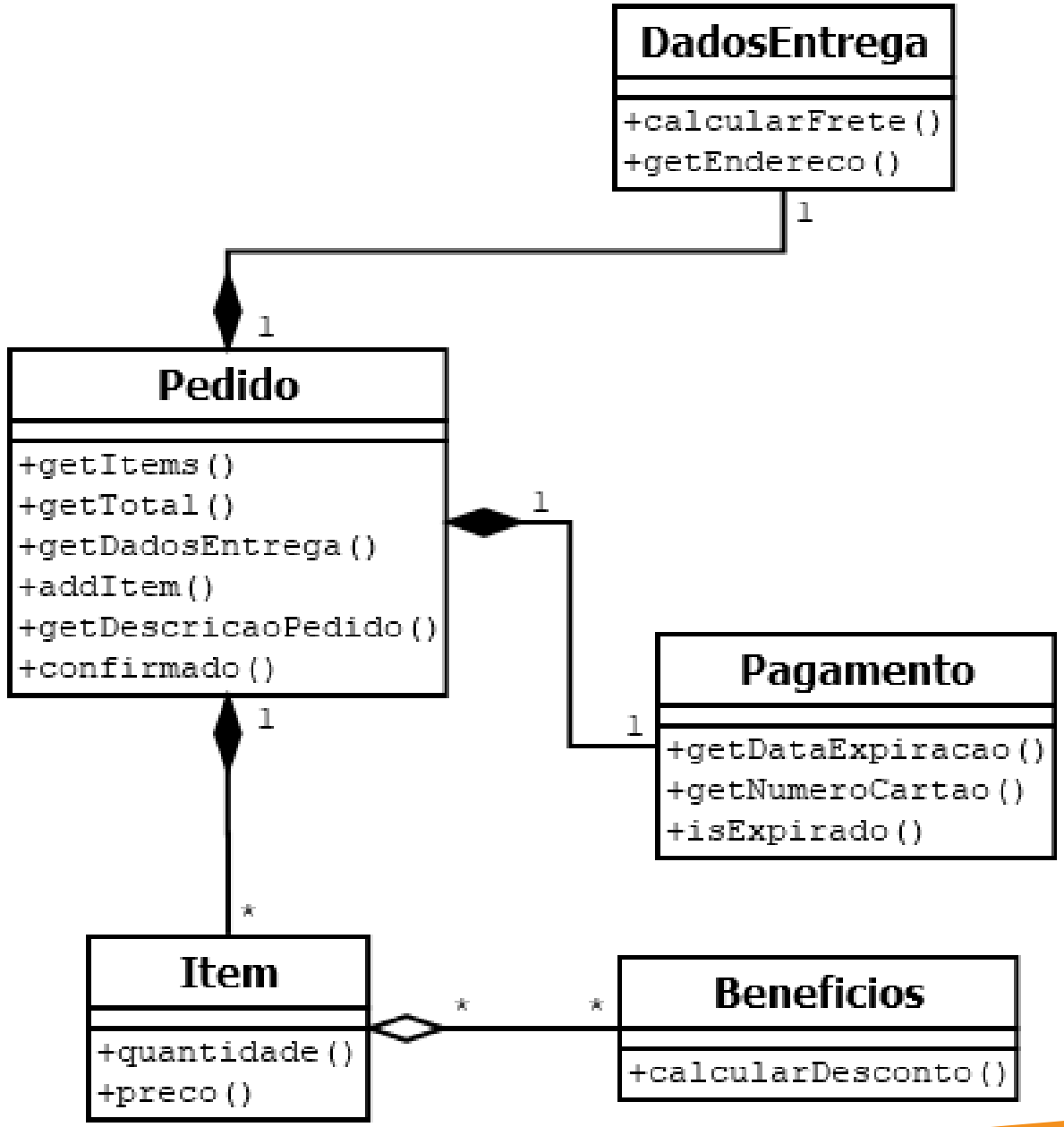
Com todos os planejamentos definidos, o passo final da projeção é desenhar o diagrama de classe e interação para repassá-las ao programador para implementação. Pode haver outros diagramas auxiliares também como o de atividades e colaboração.

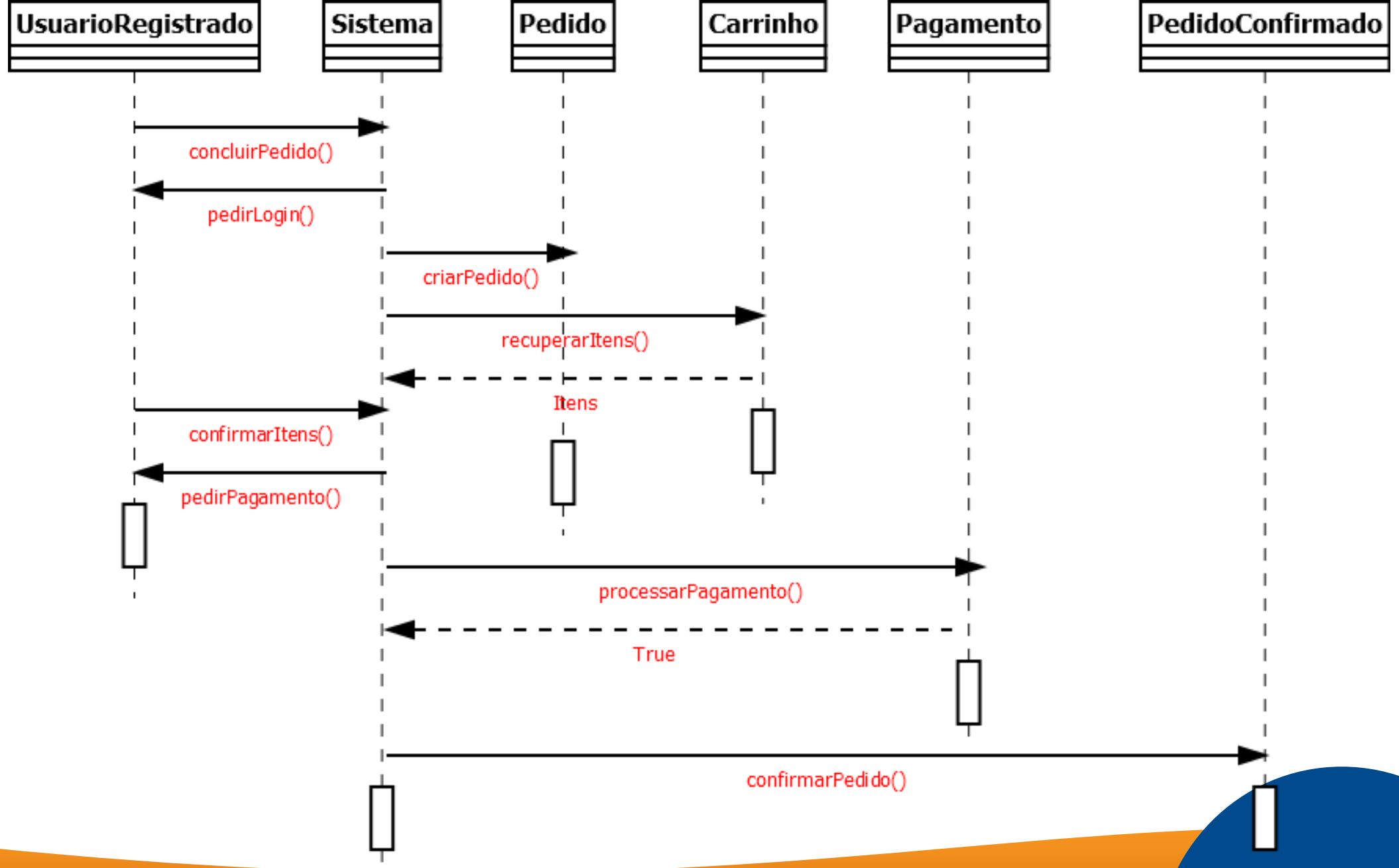
Vamos ver um exemplo de análise e construção de uma loja online, com o cenário da etapa do pedido.

- Pedido
  - O usuário registrado passa para a totalização e pagamento.
  - O usuário registrado fornece informações de entrega.
  - O sistema mostra o total do pedido.
  - O usuário registrado fornece informações de pagamento.
  - O sistema autoriza o pagamento.
  - O sistema confirma o pedido.
  - O sistema envia um e-mail de confirmação.
- Condições prévias
  - Um carrinho de compras não vazio.
- Condições posteriores
  - Um pedido no sistema.
- Alternativa: cancelar pedido
  - Durante os passos 1 a 4, o usuário opta por cancelar o pedido. O usuário volta para a home page.
- Alternativa: a autorização falhou
  - No passo 5, o sistema falha em autorizar as informações de pagamento. O usuário pode introduzir novamente as informações ou cancelar o pedido.

Usuário Registrado	
fornece informações de entrega	Informações de Entrega
fornece informações de pagamento	Pagamento

Funcionário	
recupera informações de entrega e pagamento	Informações de Entrega
introduz o pedido	Carrinho de Compras Pedido
exibe o pedido	Mostra Pedido
autoriza o pedido	Terminal de Pagamento
confirma o pedido	Pedido


DadosEntrega
+calcularFrete()
+getEndereco()
1
1
Pedido
+getItems()
+getTotal()
+getDadosEntrega()
+addItem()
+getDescricaoPedido()
+confirmado()
1
1
Pagamento
+getDataExpiracao()
+getNumeroCartao()
+isExpirado()
1
*
Item
+quantidade()
+preco()
*
*
Beneficios
+calcularDesconto()


UsuarioRegistrado
Sistema
Pedido
Carrinho
Pagamento
PedidoConfirmado
concluirPedido()
pedirLogin()
criarPedido()
recuperarItens()
Itens
confirmarItens()
pedirPagamento()
processarPagamento()
True
confirmarPedido()

POO COM JAVA

O QUE E PORQUE?

Java é uma linguagem de programação orientada a objetos, surgida em 1995, integrante da plataforma de mesmo nome. Originalmente pertencia a Sun Microsystems, porém a empresa foi vendida para a Oracle Corp em 2008.

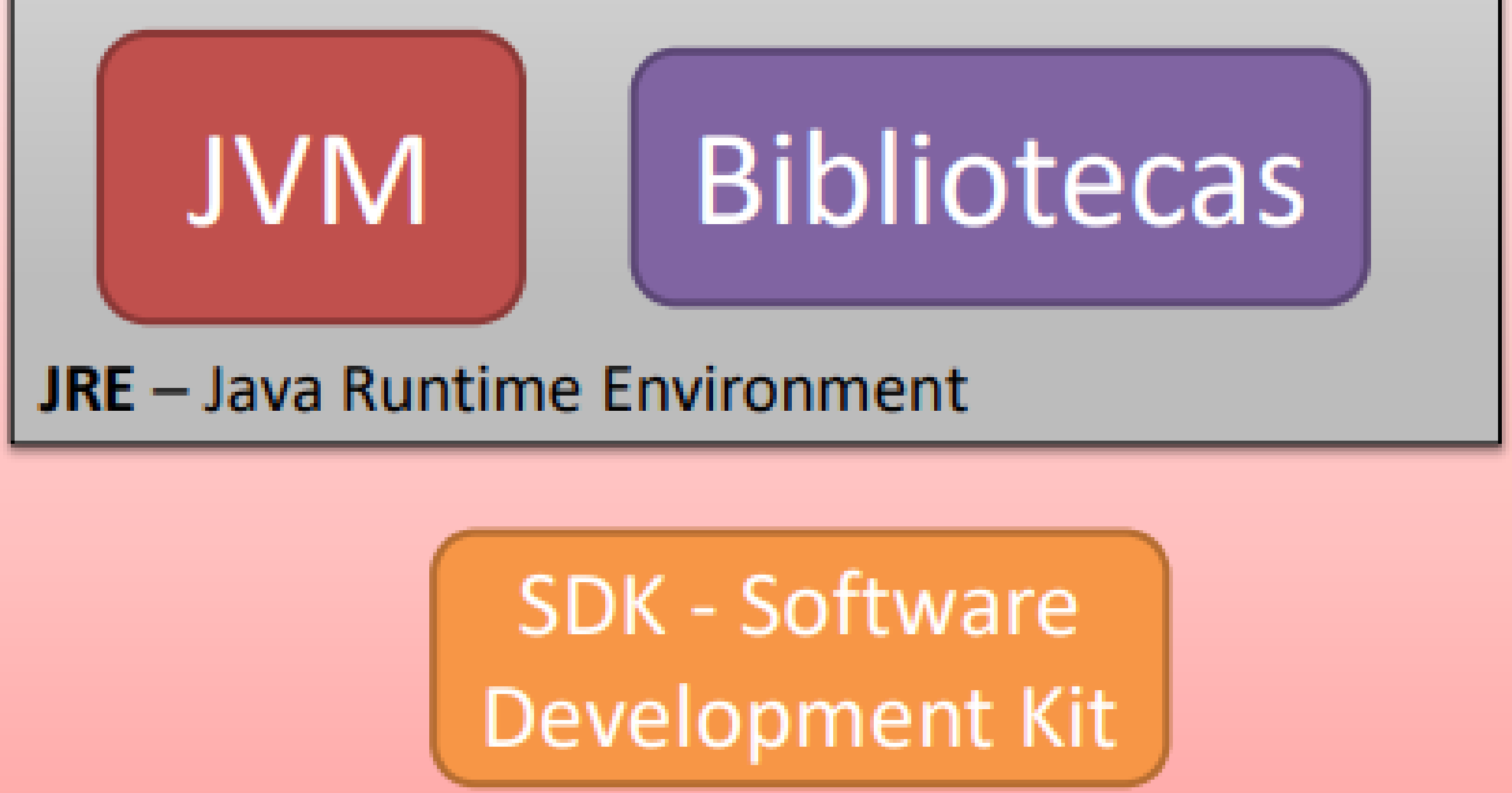
Tem compatibilidade quase universal para os equipamentos, isto é, pode ser executado em praticamente qualquer plataforma. Possui também uma vasta biblioteca pronta, que auxilia no desenvolvimento de aplicações, seja para desktops, web, mobile e sistemas embarcados.

Possui 3 plataformas principais.

PLATAFORMAS E VERSÕES

- **Java SE:** o Java Standard Edition é a base da plataforma, com o ambiente de execução, as bibliotecas e todas as ferramentas para o desenvolvimento.
- **Java EE:** o Java Enterprise Edition é uma edição para desenvolvimento de aplicações comerciais e empresariais, como ERPs, e também aplicações para a Internet, como servlets e aplicações JSP.
- **Java ME:** o Java Micro Edition é uma edição para dispositivos móveis e softwares embarcados.

É importante diferenciar 2 termos comuns nos primeiros contatos com Java. A **JRE** é o pacote com o ambiente de execução, e a **JDK** é o kit de desenvolvimento.


JVM
Bibliotecas
JRE – Java Runtime Environment
SDK - Software Development Kit
JDK – Java Development Kit

CONHEÇA A JVM

A JVM é o grande segredo da linguagem Java. A **Java Virtual Machine** é o “servidor” que permite executar qualquer aplicação Java, sendo compatível sua instalação com qualquer plataforma. Ela consegue ler o código Java (chamado de Bytecode) e “traduzir” para a plataforma de destino para correta execução.

Importante também diferenciar o código original escrito (chamado código-fonte) do código compilado (Bytecode). O código-fonte é legível por humanos, o Bytecode apenas a JVM conseguirá ler.

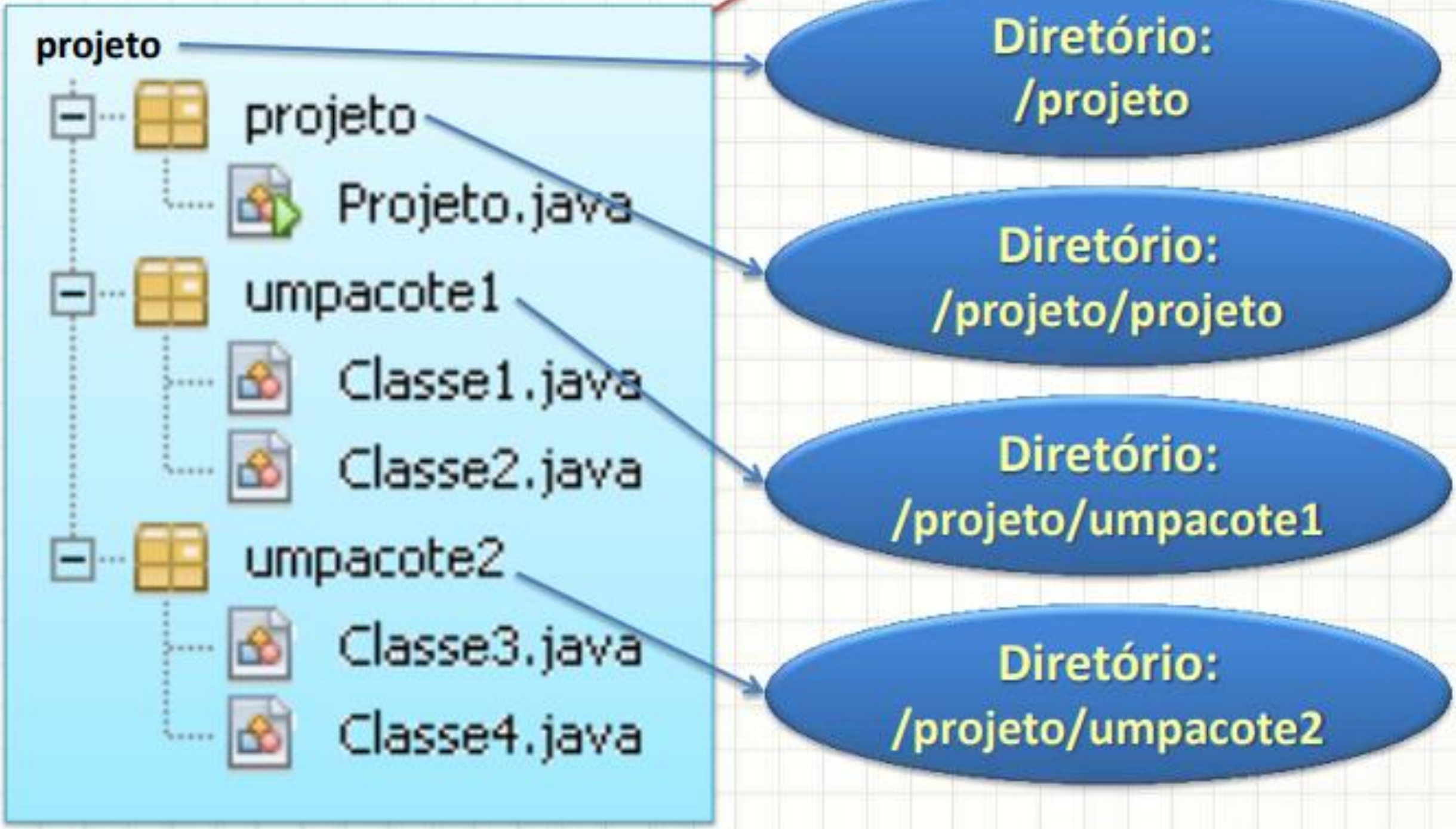
A JVM é instalada junto com o pacote JRE.

ESTRUTURA DO CÓDIGO

O código Java se organiza primeiramente em um **projeto**, como se fosse uma grande caixa**.** Dentro do projeto, criamos diversos **pacotes** que se assemelham a pastas. E dentro dos pacotes, colocamos as **classes**, como se fosse folhas. O projeto não vem especificado no código escrito, já o pacote e a(s) classe(s) são obrigatórias.

Para o pacote, usamos a palavra **package** seguida do nome do pacote**.** Geralmente é a primeira linha do código. Já para as classe utilizamos a palavra **class,** precedida por seu modo de acesso (público, privado ou protegido).

O arquivo é sempre salvo com a extensão .java. Este é o seu código-fonte original.


projeto
projeto
Projeto.java
umpacote1
Classe1.java
Classe2.java
umpacote2
Classe3.java
Classe4.java
Diretório:
/projeto
Diretório:
/projeto/projeto
Diretório:
/projeto/umpacote1
Diretório:
/projeto/umpacote2

Arquivo: **Projeto.java**

Indica o pacote
SEMPRE a primeira coisa!

```
package projeto;

public class Projeto {

    public static void main(String[] args) {

        // Código do método

    }
```

Conteúdo da Classe

E AS CLASSES?

Uma classe pode ser análogo a um pequeno programa. Então o software em Java será composto de vários programas, isto é, várias classes. Um classe pode pedir a outra para executar determinada tarefa ou informar/registrar determinados valores.

As “tarefas” que uma classe pode fazer são os **métodos**. Os valores são armazenados nos **atributos**.

As relações entre as classes são controladas pelos tipos de acesso. O método público pode ser acionado por qualquer classe. O privado, apenas pela classe onde o método está inserido. O mesmo vale para atributos e até para as próprias classes.

Ainda tem o tipo **protegido**, que restringe o escopo ao **pacote**.

PHP

LINGUAGEM DINÂMICA

O PHP é uma das muitas linguagens server-side para desenvolvimento de aplicações. Pode acessar banco de dados e também ser orientado a objetos. É possivelmente a linguagem mais “madura” que existe, sendo utilizado na web já há vários anos. Vejamos algumas de suas características:

- Licença gratuita e servidores de baixo de custo;
- Dinâmica e suporta Orientação à Objetos;
- Case-Sensitive;
- Conexão e funções nativas para diversos bancos de dados;
- Roda praticamente em qualquer servidor;
- Escalável para diversas aplicações;

PHP BASICS

- **DELIMITADORES**: Marcam o código PHP em meio ao código HTML ou em uma página separada. Todo o código PHP deve ficar entre os delimitadores para evitar erros de sintaxe.

```
<?php CÓDIGO... ?>        <? CÓDIGO... ?>        <% CÓDIGO... %>
```

- **SAÍDA**: Pode-se utilizar o comando **echo** ou o comando **print** para produzir saídas do sistema. Uma diferença entre eles é que o comando print possui a variação printf que permite formatações, como número de casas decimais, entre outras e print_r que funciona semelhante a tag <pre> do HTML.

```
echo "Olá" ou echo ("Olá")        print "Olá" ou print ("Olá")
```

PHP BASICS

- **COMENTÁRIOS**: Comentários não são processados e servem para realizar “anotações” no código. Pode-se utilizar comentários de uma ou várias linhas.

```
// Comentário de uma linha
# Comentário de uma linha
/* Comentário de várias linhas
Comentário de várias linhas */
```

- **VARIÁVEIS**: Não é necessário declarar o tipo de uma variável em PHP, bastando declarar seu nome e o valor. O símbolo utilizado para variáveis é o cifrão ($). Respeitando a lógica, valores literais devem vir entre aspas. O símbolo de atribuição em PHP é o sinal de igual (=). Já para constantes, utilizamos a função define, que recebe como parâmetro o nome da constante e seu valor

```
$nome $idade
define (“PI”, 3.14)
```

PHP BASICS

- **VARIÁVEIS VARIÁVEIS**: Recurso para criar variáveis de forma dinâmica, onde o nome desta será o valor de outra variável. No exemplo abaixo, criamos uma variável $nome com o valor “Jurema” e, logo após, uma variável $Jurema com o valor “Creuza”

```
$nome = “Jurema”;          $$nome = “Creuza”
```

- **OPERADORES ARITMÉTICOS**: +, -, *, /, %, .
- **OPERADORES DE ATRIBUIÇÃO**: =, +=, -=, *=, /=, %=, .=
- **OPERADORES LÓGICOS**: and, or, xor, !, &&, ||
- **OPERADORES DE COMPARAÇÃO**: ==, !=, <, >, <=, >=
- **OPERADORES DE INCREMENTO**: ++
- **OPERADORES DE DECREMENTO**: --

PHP BASICS - TIPOS

- **INT:** O PHP suporta dados do tipo inteiro nas bases decimal, octal e hexadecimal. Para declaração de valores na base octal, o número deve vir precedido de 0 (zero). Para a base hexadecimal, o número deve vir precedido de 0x.

```
$octal = 0312;        $hex = 0x22;
```

- **DOUBLE/FLOAT:** Aceitam todos os valores reais, podendo também ser representando através de notações E (e ou e-). O ponto flutuante separa as casas decimais através do símbolo de ponto (.) e não da vírgula(,).

```
$numero = 2.561;        $numero = 4e3;
```

- **BOOLEAN:** Os valores booleanos em PHP são representados pelas palavras reservadas `true` e `false`. Ainda admite-se o uso do valor 0 (zero) para representar false e de qualquer outro valor (negativo ou positivo) para representar o valor true.

PHP BASICS - ESTRUTURAS DE DADOS

- **ARRAY:** Podemos criar vetores em PHP, declarando os índices explicitamente ou através da palavra reservada array. Os índices são representados entre colchetes ([ ]) e podem ser valores inteiros (arrays numéricos) ou literais (arrays associativos). Ao criar vetores com a palavra array, os índices são implícitos por padrão, mas podem ser explícitos com o uso de ponteiros (numéricos ou literais).

```
$dados[1] = "Telésforo";    $dados[2] = 23;
$dados["nome"] = "Telésforo";    $dados["idade"] = 23;
$dados = array ("Telésforo", "23");
$dados = array ("nome" => "Telésforo", "idade" => "23");
```

PHP BASICS

- **LISTAS:** Semelhantes a vetores, cria uma lista de variáveis que recebem valores a partir de um array. Na prática constituem um array associativo, com outra sintaxe. São criadas com a palavra reservada list.

```
$dados = array ("Telésforo", "23");     list ($nome, $idade) = $dados;
```

- **COERÇÃO:** Quando há um cálculo feito entre um número e um valor literal, considera-se a string somente quando começar com números.

```
$numero = 1+"1a";     $numero = 1+"a222";
```

PHP BASICS

- **CASTING:** Faz uma conversão explícita do tipo de dados, colocando-se o tipo desejado entre parênteses, antes do valor ou variável a ser convertida. Ao converter valores flutuantes para inteiros, as casas decimais são truncadas.

```
$inteiro = 20;
(double) $inteiro //20.0
```

```
$decimal = 20.4;
(int) $decimal //20
```

- **SET/GET TYPE:** A função settype força a conversão dos valores. Recebe como parâmetro o valor ou variável a ser convertida e o tipo desejado. Já a função gettype retorna o tipo do valor ou variável desejada.

```
$inteiro = 20;
settype ($inteiro, double);
```

```
gettype ($inteiro);
```

PHP BASICS – ASPAS/APÓSTROFO

Na linguagem PHP, podemos utilizar aspas ou apóstrofos para escrever valores literais. A diferença é que com apóstrofos, o valor é exatamente igual ao escrito. Já com aspas, o valor é processado, no caso de variáveis ou cálculos.

```
$variavel = 'Jurema';
echo '$variavel';
echo "$variavel";
```

PHP BASICS - ESTRUTURA IF

Testa uma ou mais condições e executa o bloco de comandos daquela que for verdadeira. Quando não há chaves, somente a primeira linha do bloco é executada.

```
if (condição) { comandos; }
```

```
if (condição) { comandos; }
else { comandos; }
```

```
if (condição) { comandos; }
elseif (condição) { comandos; }
else { comandos; }
```

PHP BASICS - EXPRESSÃO CONDICIONAL

Chamado também de operador ternário, tem a mesma função de um bloco IF/ELSE, mas com sintaxe diferente, muito parecida com a função SE do Microsoft Excel.

```
condição? VERDADEIRO : FALSO
media>7? “APROVADO”: “REPROVADO”
```

O exemplo acima funciona semelhante a um bloco IF/ELSE com a estrutura abaixo:

```
if (media>7) { echo “APROVADO”; }
else {echo “REPROVADO”}
```

PHP BASICS - ESTRUTURA SWITCH

Testa uma ou mais condições e executa o bloco de comandos daquela que for verdadeira. Só faz operações de igualdade. Ao final de cada case inserimos o comando break para interromper a execução do switch.

```
switch (variavel)
{
        case valor1: comandos; break;
        case valor2: comandos; break;
        default: comandos; break;
}
```

PHP BASICS - ESTRUTURA FOR

Repete blocos de código, ou ele todo, um determinado número de vezes. Utiliza uma variável de controle para contar as repetições e uma expressão lógica VERDADEIRA como condição de parada.

```
for ($var=INICIO; CONDIÇÃO DE PARADA; INCREMENTO/DECREMENTO)
{ comandos; }
```

```
for ($var=1; $var<6; $var++)
{ echo "Olá Mundo"; }
```

PHP BASICS - ESTRUTURA WHILE

Repete códigos infinitamente, até que a condição imposta seja falsa. Como a condição é verificada no início do bloco, é possível que o código não seja executado nenhuma vez, caso a condição seja falsa de imediata. Para evitar isso, podemos utilizar um "truque" que é especificar uma condição booleana.

```
while (condição)
{ comandos; }
```

```
while (true)
{ comandos; }
```

PHP BASICS - ESTRUTURA DO-WHILE

Funciona exatamente como a estrutura while, isto é, repete infinitamente um bloco até que a condição imposta seja falsa. A diferença é que a condição é verificada no final do bloco, o que garante que o código seja executado, ao menos uma vez.

```
do
{ comandos; }
while (condição);
```

PHP BASICS - ESTRUTURA FOREACH

Utilizado para ler arrays, associando cada valor contido na coleção em uma variável simples, controlada através de uma iteração automática. O funcionamento é semelhante a estrutura FOR, porém sem a variável de controle.

```
foreach ( $array as $variavel )
{ comandos; }
```

Se o Array contiver índices textuais, é possível associar uma variável para eles na estrutura foreach.

```
foreach ( $array as $indice => $valor )
{ comandos; }
```

PHP BASICS - CONTROLE DE FLUXO

Quando trabalhamos com estruturas de repetição no PHP (for, while, do-while, foreach), existem 2 comandos para alterarmos o funcionamento do loop.

- **break**: quando executado, interrompe imediatamente uma estrutura de repetição. Pode também ser usado na estrutura switch

```
for ($var=1;$var<10;$var) { break; }
```

- **continue**: quando executado, força a próxima iteração na estrutura de repetição (como se reiniciasse o bloco).

```
for ($var=1;$var<10;$var) { continue; }
```

PHP BASICS - TRATANDO ERROS

Na linguagem PHP, podem ocorrer erros de 2 tipos: **WARNING** (não interrompem a execução do código) ou **FATAL** (o processamento é inte
Podemos ocultar a mensagem de erro com o operador @
verá nenhuma mensagem de erro, caso ocorra.

```
; Even when display_errors is on, errors that occur dur
; sequence are not displayed.  It's strongly recommende
; display_startup_errors off, except for when debugging
display_startup_errors = Off
```

```
@$alunos[2]; // evitar mostrar erro de ponto nulo
```

Existem outras formas de tratar erros em PHP. É possível também desativar a exibição de erros por uma diretiva, assim como registrar em logs os possíveis erros.

```
; server, your database schema or other information.
display_errors = On
```

PHP BASICS - REQUIRE / INCLUDE

Servem para reaproveitar um código ou uma página já existente. A diferença é que, em caso de erro, o include gera um **warning**, isto é, não interrompe a execução da página. Já o require gera um **fatal error**, interrompendo a execução da página. Podemos utilizar a expressão “once” para evitar a geração de erros.

require (endereço ou nome da página)
include (endereço ou nome da página)

require_once (endereço ou nome da página)
include_once (endereço ou nome da página)

ORIENTAÇÃO A OBJETOS EM PHP

A orientação à objeto em PHP segue os mesmo conceitos do paradigma em outras linguagens. Classes, métodos, atributos, objetos e outros conceitos permanecem com suas definições clássicas, alterando apenas a sintaxe.

O que muda são algumas palavras reservadas e particularidades da linguagem PHP, como tipos de acessos, tipos de dados e introdução dos chamados “métodos mágicos”.

CLASSE EM PHP

A declaração de classes em PHP é feita com a palavra reservada **class**. Não é necessário declarar a classe como **public**, pois isso já é feito implicitamente, mas pode ser útil ao visualizar o código de forma mais clara.

```
class Animal
{


}
```

ATRIBUTO EM PHP

Os atributos são como variáveis comuns do PHP, declarados implicitamente como **private**, seguindo o conceito de encapsulamento. Pode também ser declarado como **protected**, permitindo acesso interno e através de herança de classes.

```
class Animal
{
    private $nome;
    private $idade;
}
```

MÉTODO EM PHP

Os métodos são assinados como uma função (**function**), não necessitando do tipo de retorno, podendo ou não receber parâmetros. Os métodos mutantes utilizam a palavra **this** para referenciar o objeto e seu atributo alterado. Os métodos acessores utilizam a palavra **return** para retornar a propriedade acessada, juntamente com **this** para referenciar o objeto.

```
function setNome($nome)
{
    this->nome = $nome;
}
```

```
function getNome
{
    return this->nome;
}
```

MÉTODOS MÁGICOS EM PHP

Os métodos mágicos PHP ajudam a implementar funções pré-definidas, acelerando o desenvolvimento. A assinatura dos métodos mágica inicia-se com dois **underlines** (__) seguindo de seu nome.

```
function __construct( )
{
      echo "Objeto Criado";
}
```

```
function __destruct ( )
{
 echo "Objeto Destruído";
}
```

MÉTODOS MÁGICOS EM PHP

Os métodos mágicos PHP ajudam a implementar funções pré-definidas, acelerando o desenvolvimento. A assinatura dos métodos mágica inicia-se com dois **underlines** (__) seguindo de seu nome.

```
function __set( $atrib, $valor)
{
      this->$atrib = $valor;
}
```

```
function __get ( $atrib )
{
      return this->$atrib;
}
```

OBJETOS EM PHP

Os objetos são manipulados da mesma forma que outras linguagens, podendo invocar métodos de sua classe. Ao instanciar o objeto, o método **__construct** é invocado implicitamente. Para referenciar métodos e atributos, utilizamos o símbolo **->**

```
$bicho = new Animal;
$bicho->setNome("Astrogildo");
$bicho->setIdade(3);


$bicho->getNome();
```

HERANÇA EM PHP

A herança em PHP é feita com a palavra reservada **extends**. A subclasse passa então a acessar todos os atributos e métodos da classe herdada.

```
class Cao extends Animal
{
    private $pedigree;
}
$cachorrinho = new Cao;
$cachorrinho->setNome(“Rex”);
```

SUPERGLOBAIS

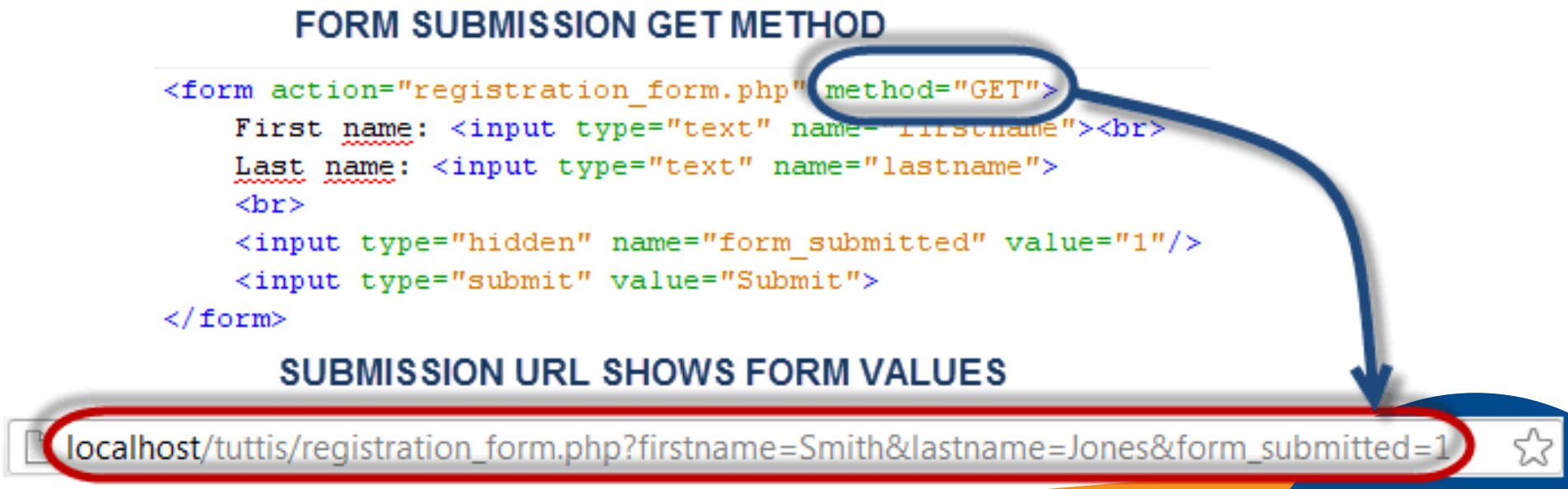
Uma **superglobal** é uma estrutura pré-definida da linguagem PHP, na forma de array associativo, onde as posições são definidas por strings. Por exemplo, quando um formulário é enviado para uma página PHP, os dados enviados podem ser tratados utilizando superglobais. Vejamos algumas delas:

- **GET / POST / REQUEST**: recebem dados não-binários de formulários;
- **FILES:** recebem arquivos binários enviados por formulários;
- **SESSION:** armazena dados em uma sessão, acessível por qualquer página;
- **SERVER / ENV:** armazena informações do servidor e do ambiente;
- **COOKIE:** manipula informações de cookies na máquina local;

SUPERGLOBAIS - $_GET

Quando os dados são transmitidos via GET (definido no atributo method do formulário), a transmissão é explícita, com os parâmetros e valores aparecendo na URL.

Pelo lado do servidor, quem recebe os dados é a superglobal $_GET, que na verdade é um array associativo. As posições desse array, são definidas pelo valor do atributo name de cada elemento enviado.



SUPERGLOBAIS - $_POST

Quando os dados são transmitidos via POST (definido no atributo method do formulário), a transmissão é implícita e os parâmetros com seu valores não aparecem para o usuário.

Pelo lado do servidor, quem recebe os dados é a superglobal $_POST, que na verdade é um array associativo. As posições desse array, são definidas pelo valor do atributo name de cada elemento enviado.

FORM SUBMISSION POST METHOD

```
<form action="registration_form.php" method="POST">
    First name: <input type="text" name="firstname"><br>
    Last name: <input type="text" name="lastname">
    <br>
    <input type="hidden" name="form_submitted" value="1"/>
    <input type="submit" value="Submit">
</form>
```

Submission URL does not show form values

SUPERGLOBAIS - $_REQUEST

A superglobal $_REQUEST pode receber dados tanto via GET quanto via POST, além de outras entradas de dados. Existe uma diretiva chamada variables_order que define qual a ordem de leitura na $_REQUEST, atente-se para que uma variável não sobrescreva a outra.

```
; This directive determines which super global arrays are registered when PHP
; starts up. G,P,C,E & S are abbreviations for the following respective super
; globals: GET, POST, COOKIE, ENV and SERVER. There is a performance penalty
; paid for the registration of these arrays and because ENV is not as commonly
; used as the others, ENV is not recommended on productions servers. You
; can still get access to the environment variables through getenv() should you
; need to.
; Default Value: "EGPCS"
; Development Value: "GPCS"
; Production Value: "GPCS";
; http://php.net/variables-order
variables_order = "GPCS"
```

SUPERGLOBAIS - $_FILES

A superglobal $_FILES armazena arquivos binários vindos de um formulário (campo do tipo “file”). Não é exatamente um array associativo, mas sim uma matriz que permite acessar outros dados do arquivo recebido.

-name: nome do arquivo original

-size: tamanho do arquivo em bytes

-type: tipo MIME do arquivo

-tmp_name: nome local temporário

-error: código do erro, caso haja, ao carregar o arquivo para o servidor

SUPERGLOBAIS - $_SESSION

Uma sessão guarda valores que são acessíveis por todas as páginas da aplicação, enquanto ela estiver aberta. Todos os valores ficam armazenados na superglobal $_SESSION. Para acessar ou manipular os valores, deve-se antes abrir a sessão com **session_start( ).** Para fechá-la, usa-se **session_destroy( ).**

PÁGINA 1

```php
session_start(); //$_SESSION global variable b
$_SESSION[ 'username' ] = 'clevertechie';
$_SESSION[ 'role' ] = 'admin';
```

PÁGINA 2

```php
session_start(); //mus
print_r( $_SESSION );
session_destroy(); //
```

SUPERGLOBAIS - $_COOKIE

Cookies são pequenos arquivos de texto gravados na máquina do usuário, como se fosse variáveis, com informações úteis para a aplicação, utilizadas para diversas finalidades, como carregamento mais rápido, memorizar preferências, distinção de usuários, etc. Veja a sintaxe abaixo.

setcookie (nome, valor, expiração, path, domínio, secure).

Apenas o nome e o valor são obrigatórios. Se a expiração não for especificada, o cookie deixa de existir ao fechar o navegador. Podemos recuperar o valor do cookie com a superglobal $_COOKIE, informando o nome do mesmo.

SUPERGLOBAIS - $_ENV / $_SERVER

São variáveis de que guardam informações de servidor e do ambiente de execução, como o endereço IP do usuário ou navegador utilizado. Podem ser acessadas de qualquer contexto do código de 3 formas, com a função **getenv( )**, com a superglobal **$_ENV** ou com a superglobal **$_SERVER**, sem diferenças entre elas.

- **REMOTE_ADDR**: captura o IP do visitante;
- **REMOTE_HOST**: hostname do visitante ou seu IP;
- **SERVER_NAME**: hostname do servidor ou seu endereço IP;
- **REQUEST_METHOD**: método de envio de dados utilizado;
- **CONTENT_TYPE**: Tipo dos dados enviados (text/html, etc);
- **CONTENT_LENGHT**: tamanho dos recebidos pelo servidor;
- **HTTP_USER_AGENT**: nome e versão do browser do visitante;

PERSISTÊNCIA DE DADOS EM PHP

A linguagem PHP possui conexão nativa a diversos bancos de dados, sem necessidade de um driver, como em outras linguagens como Java. Vamos ver exemplos de persistência de dados utilizando SGBD MySQL.

- **mysqli_connect**: função para realizar uma conexão com banco de dados MySQL. Os parâmetros são o endereço do servidor, o usuário, a senha e o banco de dados alvo.

```
mysqli_connect (“endereço_do_servidor”, “usuário”, “senha”, “banco_de_dados”);
```

- **mysqli_query**: função que executa comandos no banco de dados, através de uma conexão já estabelecida previamente. Os parâmetros são a conexão e a string de consulta.

```
mysqli_query (conexão, “consulta”);
```

PERSISTÊNCIA DE DADOS EM PHP

A linguagem PHP possui conexão nativa a diversos bancos de dados, sem necessidade de um driver, como em outras linguagens como Java. Vamos ver exemplos de persistência de dados utilizando SGBD MySQL.

- **mysqli_fetch_array**: função para indexar dados da próxima linha de um array, seja associativo ou numérico. Utilizado comumente para recuperar dados de uma consulta SELECT, podendo ser combinada com estruturas de repetição.

```
mysqli_fetch_array (conexão, consulta_query_realizada);
```

mysqli_num_rows: função para exibir a quantidade de linhas retornadas para um resultado de consulta SELECT, bastando informar como parâmetro o objeto que armazenada a consulta realizada.

```
mysqli_num_rows (consulta);
```

PERSISTÊNCIA DE DADOS EM PHP

Outra possibilidade é fazer persistência de dados utilizando um framework, existem vários no mercado voltados para PHP. Um deles é o projeto RedBean, que se caracteriza por ser um framework **ORM**.

Um framework ORM funciona sob o paradigma de orientação a objetos e seu trabalho é mapear os elementos OO (classes, objetos, herança, etc) para armazenamento em um banco de dados relacional. Por exemplo, uma classe pode ser mapeada para virar uma tabela no banco de dados, assim como atributos podem ser mapeados para campos de uma tabela.

Vamos ver alguns dos principais comandos RedBean e detalhes de seu funcionamento.

PERSISTÊNCIA DE DADOS EM PHP

- **Classe R:** todo o código RedBean está contido em uma única classe, chamada R, onde todos os métodos são estáticos.
- **Bean:** os objetos, no RedBean, são chamados de beans. Para criar um bean, chamados o método dispense, passando o tipo do objeto (que será mapeado para um registro de classe).
  - **R::dispense ('pessoa')** // cria um objeto (bean) do tipo Pessoa, mapeada para uma tabela 'pessoa'
- **CRUD:** observe abaixo os métodos para operações básicas:
  - **R::load ('pessoa', 1)** // recupera do banco de dados o objeto do tipo Pessoa, com o ID 1
  - **R::trash (bean)** // exclui um objeto recuperado previamente com o método load'
  - **R::store (bean)** // salva um objeto criado (dispense) ou recuperado (load) previamente.

PERSISTÊNCIA DE DADOS EM PHP

Para buscar registros no banco, o RedBean oferece 3 métodos, para diferentes resultados.

- **find( ):** o método find produz um array com vários resultados, recebendo como parâmetros o tipo de objeto buscado e os filtros. Veja um exemplo
  - **R::find ('pessoa', 'idade > 30')** // irá encontrar todos os registros da tabela pessoa, cuja idade é maior que 30.
- **findAll( ):** o método findAll produz um array com todos os registros da tabela, bastando informar o tipo. Veja um exemplo
  - **R::findAll ('pessoa')** // irá encontrar todos os objetos da tabela, sem filtro.
- **findOne( ):** o método findOne produz um objeto com resultado único, geralmente identificado com ID.
  - **R::findOne ('pessoa', 'id=2')** // irá encontrar o registro do tipo pessoa, com o ID 2.

SQL INJECTION

A mais comum ameaça a um sistema que consome uma base de dados é o SQL Injection, que pode ser traduzido como **injetar comandos na base de dados**, sem a devida permissão, através de brechas e vulnerabilidades da aplicação.

Importante ressaltar que a falha está na construção da aplicação e não no servidor ou no banco de dados.

Como funciona?

As aplicações enviam comandos aos bancos de dados a todo momento, seja por cliques em botões, preenchimento de informações ou utilização de links. São essas situações que podem estar vulneráveis a SQL Injection, isso é, o atacante consegue **manipular o comando correto para outro fraudulento**, podendo desta forma, apagar, criar, editar ou ler informações, as quais não tem permissão para tal.

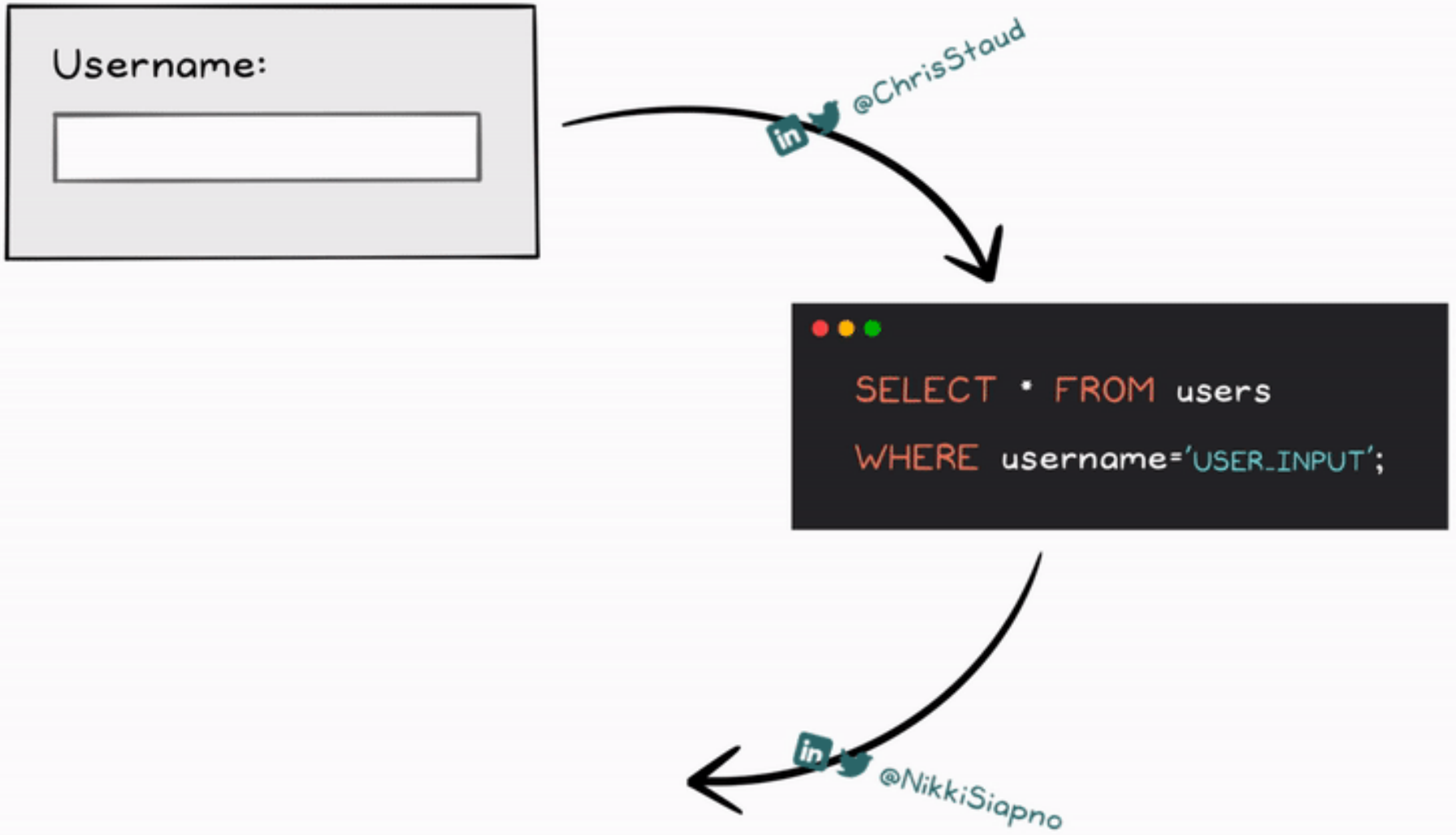
SQL INJECTION - POST

A superglobal $_POST recebe comumente dados vindos de formulário, preenchidos pelo usuário, que muitas vezes são utilizados como parâmetros em um comando SQL. O não tratamento desses dados antes de enviar ao SQL pode representar grande vulnerabilidade.

```
SELECT * FROM usuarios WHERE username='$_POST["username"]' and senha='$_POST["senha"]'
```

Observe o comando SQL acima, comumente executado em um sistema de login. Os termos em amarelos representam valores recebidos de um formulário, via POST. Porém, considere que, ao invés de digitar os valores propostos, o usuário digite **' OR 1=1"--** no primeiro parâmetro. Essa entrada será interpretada como um comando SQL se não for tratada.

```
SELECT * FROM usuarios WHERE username='' OR 1=1"-- ' and senha='$_POST["senha"]'
```


Username:
@ChrisStaud
SELECT * FROM users
WHERE username='USER_INPUT';
@NikkiSiapno

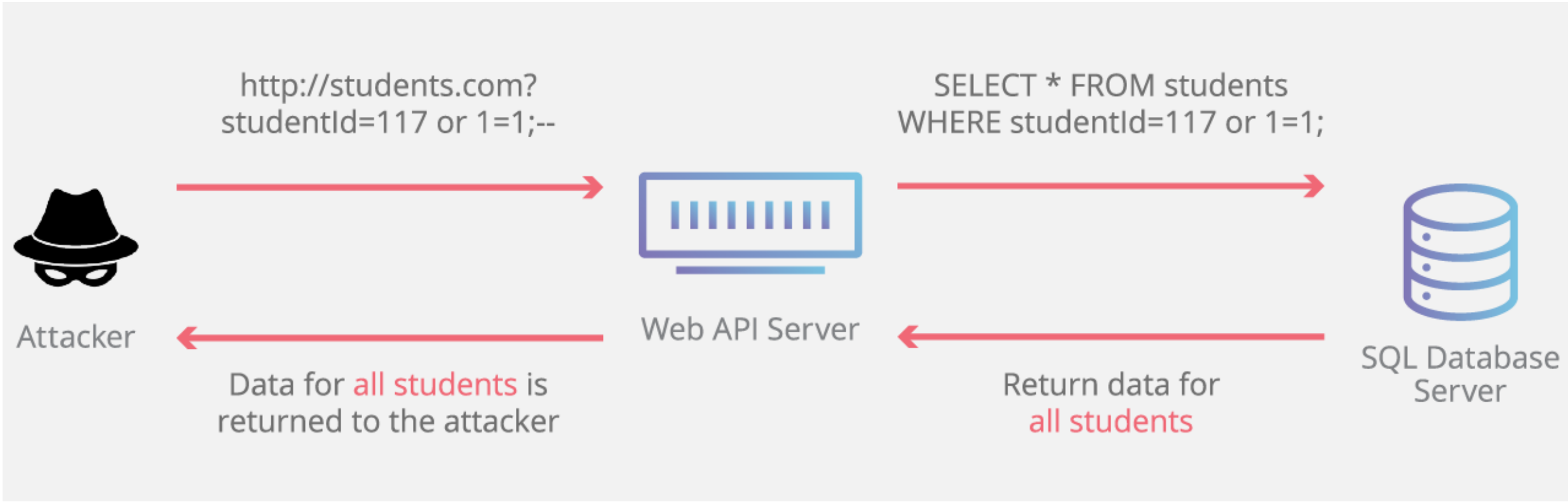
SQL INJECTION - GET

A superglobal $_GET armazena e acessa dados transmitidos pela URL de forma explícita. Considere o endereço de uma página como **www.xyz.com/produtos.php?id=2**, comumente encontrado na Internet. Espera-se que essa página exiba informações sobre o produto cujo ID no banco de dados é 2, com a página executando o comando SQL abaixo:

```
SELECT * FROM produtos WHERE id='$_GET["id"]'
```

Novamente, isso representaria uma vulnerabilidade, pelo não tratamento da informação recebida via GET, com a URL ser editada para **www.xyz.com/produtos.php?id=2' OR 1=1;--**
O comando SQL seria reescrito para:

```
SELECT * FROM produtos WHERE id='2' OR 1=1;-- '
```


http://students.com?
studentId=117 or 1=1;--
SELECT * FROM students
WHERE studentId=117 or 1=1;
Attacker
Web API Server
SQL Database
Server
Data for all students is
returned to the attacker
Return data for
all students

COMO EVITAR SQL INJECTION

Os frameworks PHP mais famosos e robustos do mercado são capazes de evitar ataques SQL Injection através de filtros nativos. Porém quando utilizamos PHP puro precisamos utilizar funções da linguagem para tratar os dados, como **filter_input** e **filter_sanitize**.

A função **filter_input** valida se a informação que está sendo recebida é do mesmo tipo que está sendo esperado. Por exemplo, um ID passado via GET teria de ser obrigatoriamente um número e isso pode ser verificado com o filter_input. Veja a sintaxe.

filter_input (ORIGEM, VARIÁVEL, FILTRO);
ORIGEM = INPUT_GET, INPUT_POST, INPUT_COOKIE, INPUT_SERVER, or INPUT_ENV.
VARIÁVEL = nome da variável em formato string.
FILTRO = filtro utilizado, que pode ser FILTER_VALIDATE_INT, FILTER_VALIDATE_EMAIL, FILTER_VALIDATE_FLOAT, FILTER_VALIDATE_IP, FILTER_VALIDATE_URL

COMO EVITAR SQL INJECTION

Outra função interessante é **filter_sanitize** que literalmente limpa os valores, retirando qualquer caractere inválido, de acordo com o filtro aplicado. Por exemplo, um ID passado via GET teria de ser obrigatoriamente um número e se o valor recebido contiver qualquer outra coisa diferente de número (como comandos de ataque SQL Injection), o filtro sanitize irá desconsiderá-la. A sintaxe é a mesma.

filter_sanitize (ORIGEM, VARIÁVEL, FILTRO);

ORIGEM = INPUT_GET, INPUT_POST, INPUT_COOKIE, INPUT_SERVER, or INPUT_ENV.
VARIÁVEL = nome da variável em formato string.
FILTRO = filtro utilizado, que pode ser FILTER_SANITIZE_INT, FILTER_SANITIZE_EMAIL, FILTER_VALIDATE_SPECIAL_CHARS, FILTER_VALIDATE_URL

FUNÇÕES

As funções em PHP, podem ou não retornar valor. Em PHP não há uma palavra reservada para especificar um procedimento, então usamos a palavra function para criar as funções seja retornando valores ou não. Veja a sintaxe abaixo:

```
function nome ( parâmetros )
{ comandos; }
```

EXEMPLOS

```
function calcula( $num1, $num2)
{
    $resultado = $num1+$num2;
    return $resultado;
}
```

```
function calcula( $num1, $num2)
{ echo $num+$num2; }
```

PARÂMETROS DA FUNÇÃO PHP

As funções podem conter valores pré-definidos para os parâmetros, funcionando como um constante. São usadas para funções em que não são passados parâmetros reais (valores vindos do código principal).

Caso um parâmetro real seja passado, este substituirá o valor padrão da função. Os valores padrão devem ser passados por último.

```
function reajuste ($salario, $margem=1.1)
{
        echo $salario*margem;
 }
```

FUNÇÃO HEADER

Existem muitas funções pré-definidas em PHP. Podemos por exemplo, manipular as informações de cabeçalhos, assim como faríamos na etiqueta <head> em HTML com <meta>. Para manipular o cabeçalho em PHP, utilizamos a função **header.**

```
header ( parâmetros );
header ("Location: http://www.sp.senac.br");
header ("Content-Description: File Transfer");
header ("Content-Type: application/pdf");
```

FUNÇÃO MAIL

O PHP permite enviar e-mails diretamente via código, sem nenhum programa adicional. A função utilizada é mail que recebe como parâmetros, o endereço de destino, o assunto e o corpo da mensagem.

Esta função retorna um valor booleano. Veja a sintaxe e um exemplo abaixo.

```
mail ( “destinatário”, “assunto”, “corpo”);
mail (
        “jonas@sorjonas.com.br”,
        “Dúvida de Programação”,
        “Como faz pra hackear o Orkut?”
);
```

FUNÇÕES NÚMERICAS/MATEMÁTICAS

- **SQRT**: Calcula a raiz quadrada (square root) de um número.
  Exemplo: sqrt(25);
- **RAND**: Gera um valor aleatório dentro de um intervalo.
  Exemplo: rand(1,100);
- **FLOOR**: Arredonda um número decimal para baixo.
  Exemplo: floor(1.25);
- **CEIL**: Arredonda um número decimal para cima.
  Exemplo: ceil(1.25);
- **MIN/MAX**: Encontra o menor/maior valor de um intervalo.
  Exemplo: min(7, 5, 2, 8);

FUNÇÕES PARA ARQUIVOS

- **fopen:** abre o arquivo para ser manipulado. A função deve ser armazenada em uma variável, para utilização em outras operações. Veja a sintaxe:

```
fopen (caminho_do_arquivo, modo);
```

O modo define as permissões. Exceto “r”, os demais tentarão criar o arquivo primeiro.

r	Abre para leitura, com cursor no início do arquivo
r +	Abre para leitura e escrita, com cursor no início do arquivo
w	Abre para escrita, com cursor no início do arquivo e reduz seu tamanho para 0
w +	Abre para escrita e leitura, com cursor no início do arquivo e reduz seu tamanho para 0
a	Abre para escrita, com o cursor no final do arquivo
a +	Abre para escrita e leitura, com o cursor no final do arquivo

FUNÇÕES PARA ARQUIVOS

- **fwrite**: escreve em um arquivo aberto anteriormente com fopen. Recebe como parâmetros, a variável com o arquivo aberto e o texto a ser escrito.

```
fwrite ($arquivo, "TEXTO A SER ESCRITO");
```

- **fgets**: lê uma linha de um arquivo, se houver permissão de leitura. Recebe como parâmetros, a variável com o arquivo aberto. Pode-se especificar uma quantidade de bytes para leitura, até o máximo de 1024 bytes. A sintaxe é **fgets($arquivo);**
- **fread**: Lê uma quantidade exata de bytes de um arquivo, sem limites. Recebe os mesmo parâmetros da função fgets. A sintaxe é **fread($arquivo, 500);**
- **fclose**: fecha um arquivo aberto anteriormente com fopen. Recebe como parâmetro a variável com o arquivo aberto. A sintaxe é **fclose ($arquivo);**

FUNÇÕES PARA ARQUIVOS

- **feof**: quando a função é chamada, verifica se chegou o final do arquivo, retornando um **boolean**. É útil quando manipulamos dados de tamanho indefinido e é comumente usado junto com uma estrutura de repetição. O arquivo deve estar aberto (fopen), mas somente no modo r, de outra forma não é permitida. Vamos ver um exemplo prático de manipulação de arquivos.

```
$arquivo = fopen("lista_alunos.txt", "r");
while (feof($arquivo) == false)
{
        echo fgets ($arquivo);
}
fclose($arquivo);
```

- **filesize**: retorna o tamanho de um arquivo indicado em bytes. A sintaxe é **filesize (caminho);**

FUNÇÕES PARA PASTAS

No PHP também temos funções para manipular diretórios, seja em um Sistema Operacional ou em um servidor.

- **mkdir**: Cria uma criar pasta no local atual, bastando indicar o nome. `mkdir('teste');`
- **rmdir**: remove uma pasta no local atual, bastando indicar o nome. `rmdir('teste');`
- **rename**: renomeia uma pasta no local atual, bastando indicar o nome ou move para outro caminho, indicando a estrutura completa. `('teste', 'teste2');` ou `('/site/teste', 'teste');`
- **dir**: lista o conteúdo do diretório atual ou de algum outro como parâmetro. `dir( );`
- **getcwd**: obtém o diretório atual `getcwd ( );`

PYTHON

LINGUAGEM PODEROSA

O Python é uma das linguagens mais poderosas da atualidade, devido ao seu poder de processamento e formas de utilização em várias áreas do conhecimento:

- Suporte a Orientação a Objetos;
- Alta performance para lidar com grande volume de dados, sendo uma das linguagens preferidas para trabalhar com Big Data
- Bibliotecas e códigos ricos para trabalho com inteligência artificial e Machine Learning;
- Sintaxe limpa, curva de aprendizagem pequena, livre e de código aberto, com acesso a banco, interpretada, portável, dinamicamente tipada e documentação abundante;
- Utilizada no desenvolvimento de jogos, aplicações Desktop, Web Scraping, automações de redes, softwares embarcados e IoT.

QUEM USA HOJE?

Como linguagem robusta e escalável, o Python é utilizado em diversos projetos da Web.

- **Google**;
- **Dropbox**;
- **Youtube**;
- **Netflix**;
- **Uber**;
- **Spotify**;
- **NASA**;
- **Instagram**.

COMO É ESCRITO?

A linguagem Python pode ser escrita e executada de duas formas:

- **Shell Interativo**

Podemos escrever os códigos em Python direto em um shell (prompt), de forma interativa (REPL), onde o interpretador vai respondendo aos comandos linha-a-linha. Um dos exemplo é o IDLE, incluso no pacote de instalação do Python.

- **Arquivos de Script**

Podemos também criar arquivos para ser interpretados e executados em formato de script. Os arquivos devem ter a extensão “.py” e podem ser executados em qualquer interpretador Python.

COMO COMEÇAR?

O ambiente Python precisa ser instalado no dispositivo onde será desenvolvido e executado, assim como em outras linguagens, como Java.

- **Download Python**

O Python pode ser baixado livre e gratuitamente no site oficial python.org, disponível em todas as plataformas, com o ambiente completo necessário para desenvolvimento (CLI, Shell, Core e Interpretador).

- **Editor de Código**

É importante uma IDE robusta para incrementar a produtividade do desenvolvimento. As mais populares são VSCode, PyCharm, Jupyter e Anaconda.

PYTHON BASICS

- **Comando de Saída**

Para exibir informações em tela, usamos a função `print( )`. Podemos também utilizar `printf ( )` para imprimir em tela com recursos de formatação. No terminal interativo, basta também digitar o nome da variável e apertar ENTER.

- **Variáveis**

A declaração de variáveis em Python é semelhante a linguagem PHP, bastando declarar o nome, sem simbologias, e atribuir um valor. Ex: `nome = 'Jonas'`. Importante notar que Python é dinamicamente tipada, não necessitando especificar o tipo da variável. O nome da variável deve respeitar as demais convenções de programação. Por fim, Python é case-sensitive no tratamento de variáveis.

PYTHON BASICS

▪ Tipos de variáveis

O Python trabalha com diversos tipos, chamados dentro da linguagem de ‘classes’, voltados para diversas áreas do conhecimento como matemática, engenharia, raciocínio lógico, etc. São algumas:

- **str:** para textos e caracteres;
- **bool:** para valores lógicos;
- **int, float:** para números inteiros e decimais, respectivamente;
- **complex:** para valores complexos, com sufixo j, utilizados na geometria e engenharia;

Para verificar se um valor é do tipo desejado, utilizamos a função `isinstance(valor, tipo)` com o parâmetro TIPO podendo ser múltiplo, separado por vírgula.

PYTHON BASICS

Operações Aritméticas

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
//	Quociente
%	Resto
**	Potenciação

Operações Relacionais

=	Atribuição
==	Comparação
>	Maior
<	Menor
>=	Maior ou Igual
<=	Menor ou Igual
!=	Diferente

Operações Lógicas

AND	Conjunção
OR	Disjunção
NOT	Negação

PYTHON BASICS

- **Entrada de dados**

A entrada de dados via terminal pode ser feita com a função input( ) podendo opcionalmente receber uma mensagem para ser atribuída a uma variável. Veja:

```
x = input( "Digite um número ")
```

Importante notar que a função input sempre captura um valor como **string**, sendo necessária então fazer a conversão (cast), bastando para isso, colocar o tipo desejado na frente da função input ( ). Por exemplo: `x = int(input( "Digite um número "))`

- **Concatenação**

Para unir texto junto a uma variável em impressão de tela, utilizamos a vírgula.

PYTHON BASICS

- **Estruturas**

As estruturas tem comportamento e formato similar à de outras linguagens, porém a maior diferença é na formação de blocos. O Python não utiliza chaves como Java ou palavras reservadas como Pascal, mas sim o símbolo de : junto a uma **indentação**

```
If ( resultado > 10):
     print (‘o resultado é maior’)
     print (‘aperte ENTER para realizar outro cálculo’)
```

No caso da estrutura if, para criar estruturas compostas, utilizamos a palavra else e para estruturas encadeadas utilizamos a palavra elif.

PYTHON BASICS

- **Estruturas**

A estrutura `while` tem comportamento e formato similar à de outras linguagens. Veja:

```
while ( idade < 30):
        idade = idade + 1
        print ('aperte ENTER para realizar outro cálculo')
```

A estrutura `for` tem comportamento e formato semelhante a estrutura `foreach` do PHP.

```
for ( lista in item):
        print (item)
```

Para o comportamento tradicional da estrutura **for**, devemos combiná-la com a função `range`, por exemplo, para 10 execuções fazemos `for ( lista in range(10) ):`

PYTHON BASICS

Podemos fazer uso de listas em Python, que se comporta como um Array em outras linguagens. Para criar uma lista, utilizamos a notação de colchetes:

```
lista = [1,4,5,8,9]
outra_lista = [“Jonas”, “Juca”, “Zezinho”]
```

`len (lista)`: retorna o tamanho da lista, isto é, a quantidade de elementos.

`sorted (lista, reverse=True)`: exibe a lista ordenada crescente ou decrescente.

`sum (lista)`: soma todos os elementos da lista.

`max/min (lista)`: mostra o maior ou menor valor da lista todos os elementos da lista.

`lista.append (valor)`: adiciona um valor no final da lista.

`lista.pop (posicao)`: remove o último valor da lista, ou na posição informada.

`lista.insert (posição, valor)`: insere um valor na posição informada.

PYTHON BASICS

Outra estrutura de dados bastante utilizada é a tupla, que é muito semelhante a uma lista em diversos aspectos. A diferença fundamental é que uma tupla é **imutável**, ou seja, não é possível alterar seu conteúdo depois de cria-la. Sua notações é em parênteses.

```
tupla = (1,4,5,8,9)
outra_tupla = (“Jonas”, “Juca”, “Zezinho”)
```

Como uma tupla é imutável, a operação `tupla[2] = 3` não seria possível. Porém demais operações como contar elementos, concatenar, exibir, entre outros, são permitidas. Também podemos converter uma lista em tupla, e vice-versa.

```
tupla_convertida = tuple(lista)
lista_convertida = list(tupla)
```

PYTHON BASICS

Mais uma estrutura bastante utilizada é o dicionário, que é semelhante a um array associativo, onde os valores são armazenados, identificados por chaves, e permitem qualquer tipo de dado. Os dicionários utilizam notação de chaves. Vamos ver um exemplo:

```
dados = { “nome” = “Jonas”, “idade” = 37, “fumante” = false }
```

`dados[idade]` Acessa a informação na chave indicada.

`dados[genero] = “M”` Atribui um novo valor a uma chave, se a chave já existir, o valor será atualizado, senão, o valor será criado.

`del dados[fumante]` Exclui chave/valor do dicionário. Se a chave não for informada, o dicionário todo será excluído. Para apenas limpar o dicionário, utilizamos a função `clear`

`dados.items/keys/values ()` Seleciona somente as chaves / valores ou tudo.

PYTHON BASICS

Por fim, temos a estrutura de conjunto, chamada em Python de “set”. São semelhantes ao dicionários, porém sem utilizar chaves, apenas valores. Conjuntos também não permite valores duplicados. Vamos ver um exemplo, considerando o conjunto abaixo:

```
cores = { “azul”, “amarelo”, “verde”, “vermelho” }
```

cores.len: retorna o tamanho do conjunto.
cores.union(outro_conjuto): faz a união entre dois conjuntos.
cores.intersection(outro_conjuto): faz a intersecção entre dois conjuntos.
cores.symmetric_difference(outro_conjuto): oposto da intersecção.
cores.add/remove(“x”): adiciona ou remove o elemento informado.
cores.pop(): remove um elemento aleatório.
cores.clear(): limpa todos os elementos do conjunto.

PYTHON BASICS

O Python permite também manipulação de Strings de forma eficiente, com diversas funções para tal. Vamos conhecer algumas, considerando a String abaixo.

```
string = senac@senacsp.com.br
```

`string[2]`: retorna a letra **r** (o tipo string é na verdade, um array de caracteres).

`len (string)`: retorna o tamanho da String em número de caracteres.

`string.split ()`: quebra a String em itens de lista, a cada espaço.

`string.capitalize/title()`: primeira letra maiúscula das palavras ou da sentença.

`string[x:y]`: seleciona o intervalo de caracteres entre as posições x e y.

`string.find (“x”)`: procura o caractere informado e retorna a posição na string.

`string.upper/lower ()`: converte para maiúsculo/minúsculo.

`string.replace(“x”, “y”)`: substitui x por y na String.

MÓDULOS E PACOTES

A linguagem Python possui uma extensa bibliotecas de pacotes e módulos para utilização. Alguns desses conteúdos precisam ser adicionados à instalação do Python, semelhante a importação de pacotes da linguagem Java.

Os conteúdos adicionais ficam todos no site pypi.org. É possível também publicar pacotes que você venha a criar. A instalação de pacotes pode ser feita pelo utilitário de linha de comando pip. Veja alguns comandos:

py -m pip list: exibe os pacotes instalados atualmente
py -m pip install/uninstall pacote: instala/desinstala o pacote pelo nome. Você pode pesquisar o nome do pacote no pypi.org.
py -m pip show pacote: mostra informações de um pacote específico.

FUNÇÕES / MODULARIZAÇÃO

Assim como outras linguagens, o Python possui a capacidade de modularizar o código, através de funções, promovendo o reuso e melhorando a legibilidade do código.

Para criar uma função em Python, utilizamos a seguinte estrutura:

```
def nome_função (parâmetros):
        corpo da função.
```

Para utilizar a função, basta chama-la no código pelo nome, passando os parâmetros, se for o caso. Os parâmetros da função, na sua construção, podem ser inicializados também. A função pode ou não retornar algum valor, assim como em outras linguagens. Para retornar valores, utilizamos a palavra-chave `return`

FUNÇÕES ESPECIAIS

Vamos ver algumas funções especiais da linguagem Python, utilizadas *inline* atribuídas a uma variável.

- **Lambda**

Espécie de atalho para criação de funções, de modo simplificado. Veja a sintaxe:

```
lambda argumentos: corpo
```

- **Map**

Semelhante ao array_map do PHP, pode aplicar funções sobre elementos de um objeto iterável. Retorna um objeto do tipo map, que pode ser convertido para uma lista.

```
map (função_a_ser_aplicada, objeto_iterável)
```

ORIENTAÇÃO A OBJETOS

O Python também oferece suporte a Orientação a Objeto. Vamos ver um exemplo de códigos e alguns conceitos

```
class Animal:  // criando uma classe
      def fazer_som (self):  // criando um método. A palavra self orienta o ponteiro.
            print ('grrrhh')


      def __init__ (self):  // criando o método construtor
            self.__nome = None  // duplo underscore pra fazer encapsulamento

meu_animal = Animal ()  // instanciando a classe
```

ORIENTAÇÃO A OBJETOS

Para implementar a herança, basta referenciar dentro de parênteses durante a construção da classe. Vamos ver também exemplos de assessores e mutantes.

```
class Cachorro(Animal): // classe Cachorro, herdando de Animal
      def get_nome (self): // criando um método do tipo get.
            return self.__nome


      def set_nome (self, novo_nome): // criando um método do tipo set.
            self.__nome = novo_nome // usamos o ponteiro self para referência

meu_cachorrinho = Cachorro () // instanciando a classe
meu_cachorrinho.set_nome('Paçoca')
```

PACOTE MATPLOTLIB

O pacote MATPLOTLIB é um pacote que adiciona funções matemáticas para a linguagem, além da capacidade de plotar gráficos. Para usar um pacote baixado, basta adicionar com o comando import.

`import math` ou `from math import *`

No código, após a importação, será possível chamar o módulo e/ou função que foram importados:

`math.sqrt()`: calcula a raiz quadrada

`math.ceil()`: arredonda o número para cima

`math.floor()`: arredonda o número para baixo

PACOTE RANDOM

O pacote RANDOM contém funções para lidar com números aleatórios, sendo capaz de gerar de várias formas, escolher, sortear, dentre outras funções. Vamos ver seu uso:

```
import random
```

Agora podemos chamar algumas funções do módulo importado.

`random.randint(inferior,superior)`: gera um número inteiro dentro do intervalo.

`random.random()`: gera um número decimal, entre 0 e 1.

`random.uniform(inferior,superior)`: gera um número decimal dentro do intervalo.

`random.choice(lista)`: escolhe um número dentro de uma lista

`random.sample(lista,qtd)`: escolhe uma quantidade de números numa lista.

`random.shuffle(lista)`: embaralha uma lista.

PACOTE OS

O pacote OS contém funções para lidar com várias tarefas relacionadas ao sistema operacional, com comandos semelhantes a um terminal ou prompt.

```
import os
```

Agora podemos chamar algumas funções do módulo importado.

os.name: identifica o sistema operacional (‘nt’ pra Windows, ‘posix’ para Linux/Mac).

os.getcwd(): mostra o diretório atual.

os.listdir(): mostra o conteúdo do diretório atual. Para um diretório específico, basta informar entre os parênteses, escapando as barras do endereço.

os.chdir(diretorio): abre/acessa um diretório

os.mkdir(nome): cria uma pasta por caminho absoluto ou relativo.

PACOTE PYAUTOGUI / PYPERCLIP

O pacote PYAUTOGUI e PYPERCLIP contém funções para automatizar certas tarefas utilizando ações de mouse e teclado. Vamos ver seu uso:

```
import pyautogui, import pypercli
```

Agora podemos chamar algumas funções do módulo importado.

`pyautogui.hotkey ('ctrl', 't')`: simula o pressionamento de um atalho.

`pyautogui.click ()`: executa um clique do mouse, na posição atual.

`pyautogui.doubleClick ()`: executa um duplo-clique do mouse, na posição atual.

`pyautogui.position ()`: pega a posição atual do mouse.

`pyautogui.write (mensagem)`: escreve uma mensagem na posição atual

`pyautogui.alert (mensagem)`: exibe um pop-up na tela.

`pyautogui.press (tecla)`: pressiona uma tecla do teclado.

PYTHON CRUD

O Python, assim como outras linguagens de backend, pode suportar banco de dados. Vamos fazer um CRUD utilizando o MySQL. Para isso, o primeiro passo é baixar e importar o módulo do MySQL (mysql-connector-python).

```
import mysql.connector
```

Agora vamos estabelecer a conexão com o banco de dados, informando os dados da conexão, semelhante a outras linguagens.

```
mysql.connector (host, usuário, senha, banco)
```

Essa conexão precisa ser armazenada em um variável para manipulação (fechamento, execução de comandos, etc). Vamos armazenar numa variável chamada **conexão**.

PYTHON CRUD

Ao mesmo tempo precisamos criar um elemento para executar as ações no banco de dados a partir da variável que armazena a conexão. Este elemento é o cursor. Criaremos executando a função e guardando numa variável.

```
cursor = conexao.cursor()
```

CREATE / UPDATE / DELETE

Para comandos que não possuem retorno, chamaremos a função **execute**, a partir do cursor. Além disso, para esses comados que alteram o banco de dados, precisamos executar a função commit, a partir da conexão.

```
cursor.execute (comando)
conexao.commit ()
```

PYTHON CRUD

READ/RETRIEVE

Para o comando SELECT, que não faz alteração de banco de dados, não precisamos confirmar a execução com **commit**. Para recuperar um registro do banco de dados, basta executar o comando com a função execute. Como a leitura possuirá um retorno (diferentemente dos outros comandos), devemos capturar esse retorno com a função fetchall.

```
cursor.execute (comando)
cursor.fetchall ()
```

Para visualizar o resultado, armazene em um variável o comando fetchall e execute o comando print nessa variável.

FUNÇÕES DE E-MAIL

Para enviar um e-mail usando Python, precisaremos do módulo de e-mail e servidores SMTP. Vamos iniciar, importando esses pacotes.

```
import smtplib
import email
```

Primeiro faremos a conexão com o servidor SMTP, armazenando a conexão numa variável. A função SMTP recebe 2 parâmetros, o endereço do servidor SMTP e a porta de comunicação. Após configurar a conexão, adicionamos a camada TLS e enviamos os dados para autenticar.

```
servidor = smtplib.SMTP(host='', port=)
servidor.starttls()
servidor.login(ENDERECO_EMAIL, SENHA)
```

FUNÇÕES DE E-MAIL

Devidamente autenticados no servidor, agora montaremos o e-mail com a mensagem, o assunto e endereço de destino.

```
email = MIMEMultiplart()
email['From'] = SEU_EMAIL
email['To'] = EMAIL_DESTINO
email['Subject'] = ASSUNTO

email.attach(MIMEText(MENSAGEM, 'plain'))
```

Com tudo configurado, basta chamar a função para enviar o e-mail, a partir da autenticação feita anteriormente.

```
servidor.send_message(email)
```

FUNÇÕES DE E-MAIL

É possível, também enviar e-mail de forma mais simples, sem linha de assunto e sem formatação na mensagem (texto puro). Neste caso, podemos utilizar a função **sendmail**, recebendo 3 parâmetros

```
from = jonas@sorjonas.com.br
destinatario = alunos@escola.com.br
conteudo = “estuuuuuuuuuudem”
```

Com tudo configurado, basta chamar a função para enviar o e-mail, a partir da autenticação feita anteriormente.

```
servidor.sendmail(from, destinatario, conteudo)
```

HTML

TUDO É HTML!

O HTML (Hypertext Markup Language) é uma linguagem de marcação de hipertexto, permitindo “seccionar” o código de uma página web, estruturando-a. É a mais básica das linguagens web, sendo a base para qualquer página na Internet.

Importante frisar que o HTML não é considerado uma linguagem de programação e sim uma linguagem de marcação, já que serve apenas para marcar pontos do documento onde será inserido algum elemento estrutural como um parágrafo, um fonte maior ou uma imagem.

TENHA ETIQUETA

As marcações ocorrem por meio de tags (etiquetas), que são palavras reservadas representadas entre sinais de maior e menor (< >). Cada etiqueta define uma seção ou estrutura diferente para a página. As etiquetas ainda podem conter atributos que complementam seu funcionamento.

EXEMPLO DE ETIQUETAS

As etiquetas contém algumas características chamadas **atributos**. A etiqueta **<p>** por exemplo, marca um parágrafo na página. Um dos atributos possíveis é **align** que modifica seu alinhamento.

Outro exemplo é a etiqueta **<font>** que faz marcações sobre a fonte/letra da página. Um de seus atributos é **color** que define a cor da letra.

```
<p align="center">
<font color="orange">
```

FECHAMENTO DE ETIQUETAS

Toda etiqueta deve ser fechada para indicar ao navegador que aquela marcação terminou. O fechamento é feito do mesmo modo que a etiqueta é aberta, porém antecedido por uma barra. Algumas etiquetas não possuem fechamento explícito; nesse caso acrescenta-se a barra na mesma linha de sua abertura.

 Olá, moçada bonita

 Este, moçada bonita

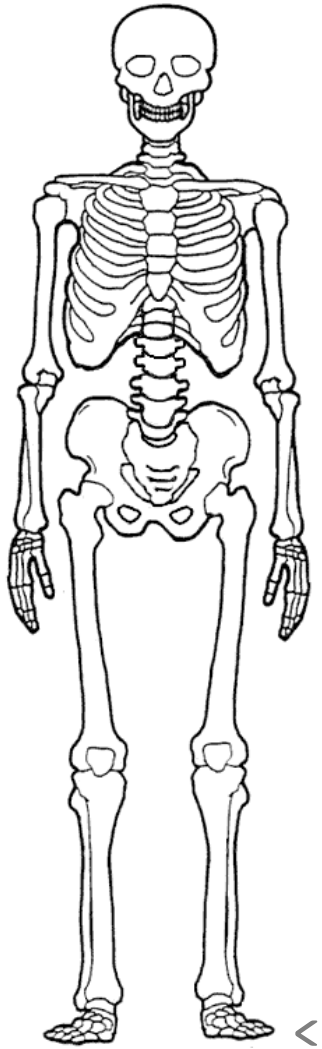
 Esta etiqueta quebra uma linha no texto

<img /> Esta etiqueta insere uma imagem

ESTRUTURA HTML

Todo código HTML deve ter um “esqueleto” base, um conjunto de etiquetas que constituem sua estrutura básica, para um bom funcionamento e interpretação na maioria dos navegadores.

Dentro das seções, inserimos o conteúdo adequado.



```
<html>
    <head>
        <title>...</title>
    </head>
    <body>

    </body>
    <footer>
    </footer>
</html>
```

FORMULÁRIOS

Uma das etiquetas mais utilizadas é a <form>, que marca a abertura de um formulário. Possui o fechamento </form>. Formulários são utilizados para capturar dados inseridos por usuários. O objetivo é tratar os dados em um código de servidor, para, com eles, armazenar em bancos de dados, enviar e-mails, etc.

```
<form name="..." method="..." action="...">

</form>
```

ATRIBUTOS DO FORMULÁRIO

name: o nome do formulário do ponto de vista do código.

method: método de tratamento dos dados (GET ou POST).

action: ação executada quando o formulário for enviado, podendo ir para uma página específica, executar novamente a mesma página, chamar um código javascript, etc.

enctype: quando o formulário enviar arquivos binários, deve-se incluir o atributo enctype com o valor multipart/form-data

ELEMENTOS DE FORMULÁRIO

O conteúdo do formulário é formado por botões, campos de texto, listas e opções de seleção, todos destinados a capturar os dados inseridos pelos usuários. Cada elemento possui a etiqueta correspondente e respectivos atributos. Com tantos elementos, o uso do atributo **name** torna-se essencial para manipulação do código. Alguns atributos são:

- **Input**
- **Select**
- **Text Area**

TAG INPUT

É a etiqueta mais utilizada por permitir representar várias formas de captura de dados. Não possui fechamento explícito, sendo isto então, feito com uma barra ao final da linha de abertura da etiqueta. Cada modo de captura da etiqueta input é representado pelo atributo **type.**

- <input type="text"> cria caixas de texto de uma linha. Alguns atributos são "value" (valor inicial), "size" (tamanho em caracteres) e "maxlenght" (máximo de caracteres permitidos).
- <input type="password"> cria caixas de texto que esconde o conteúdo digitado. Tem as mesmas propriedades do tipo "text".

TAG INPUT

- <input type="checkbox"> cria uma caixa de marcação, para selecionar um ou mais valores em uma lista de itens. Alguns atributos são "value" (valor do item quando selecionado) e "checked" (diz que o item está selecionado).
- <input type="radio"> cria botões de seleção, semelhante ao checkbox, porém permite apenas uma opção selecionada dentro de uma lista. Possui as mesmas propriedades do checked box.
- <input type="hidden"> elemento que não aparece no formulário, geralmente utilizado para enviar valores "escondidos", com a propriedade "value".

TAG INPUT

- <input type="file"> cria uma caixa para envios de arquivos binários (imagens, planilhas, música, vídeos, etc). Alguns atributos são "multiple" (permite enviar vários arquivos) e "size" (tamanho em caracteres).
- <input type="button"> botão genérico que contém uma ação através do atributo "onclick" na linguagem javascript. Outro atributo aceito é o "value" (rótulo do botão que aparece para o usuário).
- <input type="submit"> botão que quando clicado, envia o formulário. Possui o mesmo atributo "value" do tipo button.
- <input type="reset"> limpa os campos e retorna os valores iniciais.

TAG SELECT

Etiqueta que cria uma lista de opções, podendo selecionar um (como um “radio”) ou vários itens (como um “checkbox”). Cada item fica em uma etiqueta **<option>**. Outros atributos aceitos são “name” (nome da lista), “size” (tamanho da lista em número de linhas), “multiple” (permite múltipla seleção), “value” (valor do option, quando selecionado) e “selected” (atributo do option que está selecionado).

```
<select name="..." size="..." multiple>
        <option value="Arroz">Arroz</option>
        <option value="Feijão">Feijão</option>
</select>
```

TAG TEXTAREA

Cria um campo de texto de várias linhas. Assim como a etiqueta “select” (e diferente de input), possui fechamento explícito. Note que o valor da caixa fica fora das marcações da etiqueta. Veja alguns atributos:

- **name**: o nome da caixa, do ponto de vista do código.
- **rows**: altura da caixa, em número de linhas.
- **cols**: largura da caixa, em número de colunas.

```
<textarea name="..." cols="..." rows="...">
        conteúdo da caixa
</textarea>
```