



# TÉCNICAS DE PROGRAMAÇÃO 1

Fernando Albuquerque

# Tópicos



- ✓ Decomposição de problemas usando classes e objetos.
- ✓ Técnicas de desenho e programação orientada a objetos.
- ✓ Reúso com herança e composição de classes.
- ✓ Polimorfismo por subtipos/parametrizado.
- ✓ Tratamento de exceções.
- ✓ Tópicos especiais (distribuição, concorrência etc.).
- ✓ Técnicas avançadas de modularização.



# CONCEITOS DE BASE

# Conceitos de base



**Software.** (1) computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system (IEEE 828-2012) (2) all or part of the programs, procedures, rules, and associated documentation of an information processing system (IEEE 828-2012) (ISO/IEC 19770-1:2012) (ISO/IEC 2382-1:1993).

**Software product.** (1) set of computer programs, procedures, and possibly associated documentation and data (ISO/IEC 25000:2014) (ISO/IEC 12207:2008) (2) any of the individual items in (1) (ISO/IEC/IEEE 24765:2010) (3) complete set of software designed for delivery to a software consumer or end-user that may contain computer programs, procedures and associated documentation and data (ISO/IEC 19770-5:2013).

**System.** (1) combination of interacting elements organized to achieve one or more stated purposes (ISO/IEC 25000:2014) (ISO/IEC TR 90005:2008) (ISO/IEC 15939:2007) (ISO/IEC/IEEE 15288:2015) (2) a set or arrangement of elements that are related, and whose behavior satisfies operational needs and provides for the life cycle sustainment of the products (IEEE 1220-2005).

**Software system.** (1) system for which software is of primary importance to the stakeholders (ISO/IEC/IEEE 24765e:2015).

# Conceitos de base



## ✓ Classes de produtos de software:

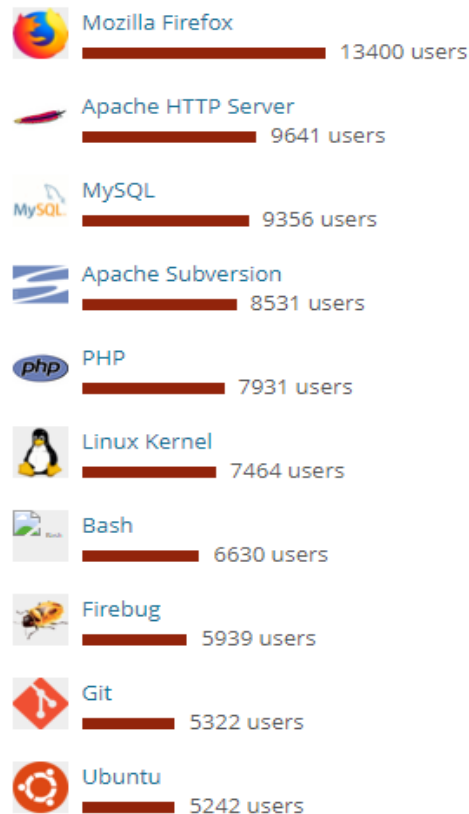
- Produto genérico.
- Produto sob encomenda.

## ✓ Exemplos de produtos genéricos:

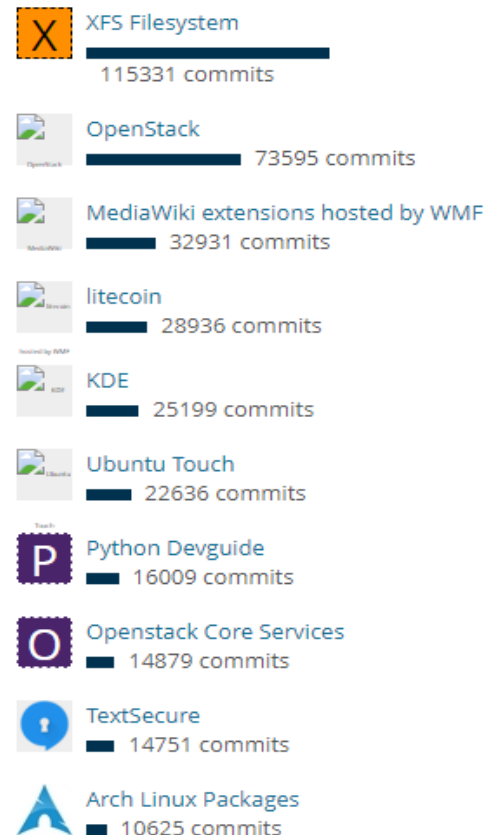
- Editor de texto.
- Apresentação.
- Planilha.
- Sistema de contabilidade.
- Sistema de controle de estoque.
- Navegador (*browser*).

# Conceitos de base

## Most Popular Projects



## Most Active Projects



# Conceitos de base

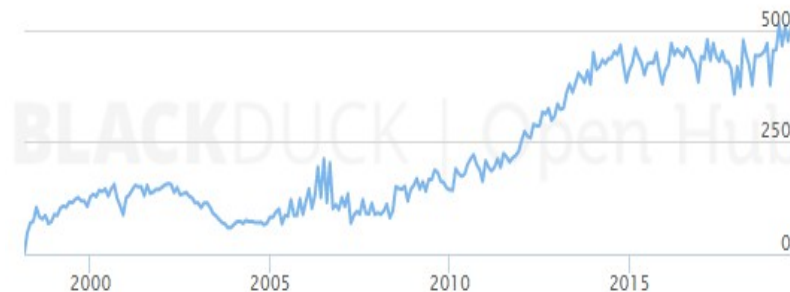


Mozilla Firefox

Settings | Report Duplicate

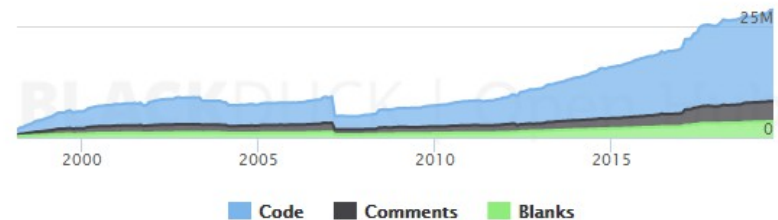
Community

Contributors per Month



Code

Lines of Code



Languages



FONTE: <https://www.openhub.net/>

# Conceitos de base



## ✓ Sistema:

- Composto por conjunto de elementos.
- Elementos conectados com propósito(s) definido(s).
- Elementos conectados compõem um todo.
- Pode ser descrito por modelos.
- Subsistema é sistema parte de outro sistema.

## ✓ Sistema de software:

- Sistema onde software é de importância primária para os interessados (*stakeholders*) (ISO/IEC/IEEE 12207:2017 Systems and software engineering--Software life cycle processes, 3.1.55).





# Conceitos de base



## v Elementos em sistema de software:

1. Sistema
2. Subsistema
3. Programa
4. Módulo
5. Classe
6. Função
7. Bloco de código
8. Linha de código



Complexidade decrescente.

# Conceitos de base



## ✓ Programa de computador:

- Combinação de instruções de computador e definições de dados que capacitam o hardware do computador a realizar funções computacionais e de controle (ISO/IEC/IEEE 24765:2017 Systems and software engineering-Vocabulary).

## ✓ Programar:

- Escrever programa de computador (ISO/IEC/IEEE 24765:2017 Systems and software engineering-Vocabulary).

# Conceitos de base

```
void ComandoSQL::conectar() throw (EErroPersistencia) {
    rc = sqlite3_open(nomeBancoDados, &bd);
    if( rc != SQLITE_OK )
        throw EErroPersistencia("Erro na conexao ao banco de dados");
}

void ComandoSQL::desconectar() throw (EErroPersistencia) {
    rc = sqlite3_close(bd);
    if( rc != SQLITE_OK )
        throw EErroPersistencia("Erro na desconexao ao banco de dados");
}

void ComandoSQL::executar() throw (EErroPersistencia) {
    conectar();
    rc = sqlite3_exec(bd, comandoSQL.c_str(), callback, 0, &mensagem);
    if(rc != SQLITE_OK){
        if (mensagem)
            free(mensagem);
        throw EErroPersistencia("Erro na execucao do comando SQL");
    }
    desconectar();
}

int ComandoSQL::callback(void *NotUsed, int argc, char **valorColuna, char **nomeColuna){
    NotUsed=0;
    ElementoResultado elemento;
    int i;
    for(i=0; i<argc; i++){
        elemento.setNomeColuna(nomeColuna[i]);
        elemento.setValorColuna(valorColuna[i] ? valorColuna[i] : "NULL");
        listaResultado.push_front(elemento);
    }
    return 0;
}
```

# Conceitos de base



## ✓ Programação:

- (1) atividade de desenvolvimento de software (ISO/IEC/IEEE 24765:2017) (2) projeto, escrita, modificação e teste de programas (ISO/IEC 2382:2015).

## ✓ Exemplos de valores na programação:

- Comunicação, simplicidade e flexibilidade (BECK, T. *Padrões de implementação: Um catálogo de padrões indispensável para o dia a dia do programador*. Bookman, 2013).

# Conceitos de base



## ✓ Linguagem de programação:

- Linguagem usada para expressar programas de computador (ISO/IEC/IEEE 24765:2017).










## ✓ Exemplos de linguagens de programação:

- |             |               |        |
|-------------|---------------|--------|
| • C++       | C#            | Eiffel |
| • Java      | Object Pascal | Perl   |
| • PHP       | Python        | Ruby   |
| • Smalltalk |               |        |

# Conceitos de base



## ✓ Populares linguagens de programação:

Jul 2021	Jul 2020	Change	Programming Language		Ratings	Change
1	1			C	11.62%	-4.83%
2	2			Java	11.17%	-3.93%
3	3			Python	10.95%	+1.86%
4	4			C++	8.01%	+1.80%
5	5			C#	4.83%	-0.42%
6	6			Visual Basic	4.50%	-0.73%
7	7			JavaScript	2.71%	+0.23%
8	9	^		PHP	2.58%	+0.68%
9	13	^^		Assembly language	2.40%	+1.46%
10	11	^		SQL	1.53%	+0.13%

FONTE: <https://www.tiobe.com/tiobe-index/>

# Conceitos de base



## ✓ Paradigma de programação:

- Estilo de programação.
- Modo de programar.
- Não diz respeito a linguagem específica.
- Existem diversos paradigmas de programação.

## ✓ Exemplos de paradigmas de programação:

- Programação procedural (*procedural programming*).
- Programação funcional (*functional programming*).
- Programação orientada a objetos (*object oriented programming*).





# ORIENTAÇÃO A OBJETOS

# Orientação a objetos



## ▼ Introdução:

- Paradigma de desenvolvimento de software.
- Paradigma existente há décadas.
- Paradigma que enfatiza encapsulamento.
- Paradigma onde classes e objetos são conceitos relevantes.
- Análise orientada a objetos.
- Desenho (*design*) orientado a objetos.
- Programação orientada a objetos.

# Orientação a objetos



*“It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs.” Kay, A. The early story of Smalltalk. ACM SIGPLAN Notices, 1993.*

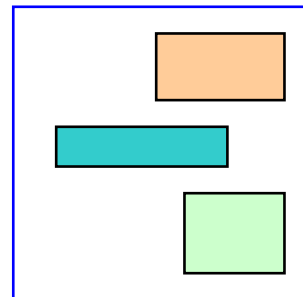
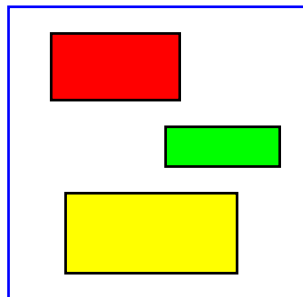
*“Object oriented programming ... Mark Sherman of Dartmouth notes that we must be carefull to distinguish two separate ideas that go under that name: abstract data types and hierarchical types, also called classes. ... Each removes one more accidental difficulty from the process, allowing the designer to express the essence of his design without having to express large amounts of syntatic material that add no new information content. ... Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential; and such attacks make no change whatever in that.” Brooks, F. No Silver Bullet – Essence and Accident in Software Engineering. IEEE Computer, 1987.*

# Orientação a objetos



## ✓ Encapsulamento:

- Aspectos internos escondidos.
- Dados escondidos.
- Acesso aos dados ocorre por meio de códigos apropriados.
- Maior controle sobre efeitos de manutenções.
- Elemento pode ser alterado sem afetar seus usuários.



# Orientação a objetos


## ▼ Classe:

- Padrão a partir do qual objetos são criados.
- Define comportamento de objetos.
- Define estrutura de objetos.
- Declarações e definições de atributos e métodos.

[BuilderSubsistCadastro](#)  
[BuilderSubsistMetricas](#)  
[BuilderSubsistProjeto](#)  
[CntrCadastro](#)  
[CntrDefeito](#)  
[CntrInicializacao](#)  
[CntrNavegacao](#)  
[CntrProjeto](#)  
[CntrTamanho](#)  
[CntrTempo](#)  
[ContainerCadastro](#)  
[ContainerDefeito](#)  
[ContainerProjeto](#)  
[ContainerTamanho](#)  
[ContainerTempo](#)  
[Defeito](#)  
[Fase](#)  
[Projeto](#)

[TelaCadastrros](#)  
[TelaDefeito](#)  
[TelaFase](#)  
[TelaMetricas](#)  
[TelaNavegacao](#)  
[TelaProjeto](#)  
[TelaTamanho](#)  
[TelaTempo](#)  
[TelaTipoDefeito](#)  
[TelaTipoDocumento](#)  
[Tempo](#)  
[TipoDefeito](#)  
[TipoDocumento](#)  
[Utilitarios](#)

# Orientação a objetos



**class.** (1) *abstraction of the knowledge and behavior of a set of similar things (IEEE 1320.2-1998 (R2004))* (2) *static programming entity in an object-oriented program that contains a combination of functionality and data (ISO/IEC/IEEE 24765:2017).*

**declaration.** (1) *non-executable program statement that affects the assembler or compiler's interpretation of other statements in the program (ISO/IEC/IEEE 24765:2017)* (2) *set of statements which define the sets, constants, parameter values, typed variables and functions required for defining the annotations on a high-level Petri Net graph (ISO/IEC 15909-1:2004).*

# Orientação a objetos



## ▼ Biblioteca de classes:

- Repositório de classes.
- Promove reuso.
- Classes tipicamente organizadas em hierarquias.

## ▼ Exemplos de domínios:

- Interface gráfica com o usuário.
- Estatística.
- Comunicação de dados.

# Orientação a objetos

## Free C / C++ Libraries, Source Code and Frameworks

Reusable source code, libraries, components to link with your programs

You are here: [thefreecountry.com \(main page\)](#) > [Free Source Code, Programming Libraries & Components](#) > [Free C/C++ Libraries, Source Code and Frameworks](#)

Ads by Google

Free C C++ Compilers

C++ Software

Learn to Code C++ Free

## Free C / C++ Libraries, Source Code and Frameworks

Here are some C and C++ libraries, DLLs, VCLs, source code, components, modules, application frameworks, class libraries, source code snippets, and the like, that you can use with your programs without payment of fees and royalties. Note that some libraries (etc) listed here may have certain restrictions about its use and/or distribution. Be sure you read the licence conditions in each package before using them.

A few types of libraries have been moved to their own pages, due to the large number of items in those categories. Here are some of the other pages containing free libraries on this site:

- [Free Audio, Sound, Music, Digitized Voice Libraries](#)
- [Free Compression and Archiving Libraries](#)
- [Free Database Libraries, SQL Servers, ODBC Drivers, and Tools](#)
- [Free Debugging Libraries, Memory Leak and Resource Leak Detection, Unit Testing](#)
- [Free Edit Controls, Libraries and Source Code](#)
- [Free Encryption Libraries](#)
- [Free C/C++ Font Libraries](#)
- [Free Curses Implementations](#)
- [Free Games Programming Libraries and Source Code](#)
- [Free Graphics, Image Drawing Libraries, 3D Engines, 2D Engines](#)
- [Free GUI Libraries](#)
- [Free Numerical, Graphs, Statistics and Mathematical Libraries](#)
- [Free Operating System Development Libraries](#)
- [Free PDF Programming Libraries and Source Code](#)



# Orientação a objetos



## ✓ Desenvolvimento de biblioteca de classes:

- Estudar domínio (*domain*) de aplicação.
- Definir interfaces das classes.
- Tornar intuitivas as interfaces.
- Refinar interfaces para facilitar reuso.
- Avaliar possibilidade de herança.
- Avaliar impacto sobre classes na hierarquia de classes.
- Realizar manutenção.
- Testar.
- Documentar.

# Orientação a objetos



## ✓ Atributo:

- Elemento estático.
- Propriedade ou característica.
- Pode ter valor fixo (constante) ou variável.
- Pode ter valor diferente para cada objeto (instância).
- Pode ter valor igual para cada objeto (instância).
- Declaração pode informar nome.
- Declaração pode informar tipo.
- Declaração pode informar visibilidade (público etc.).
- Declaração pode informar valor *default* inicial.

# Orientação a objetos



## ✓ Método:

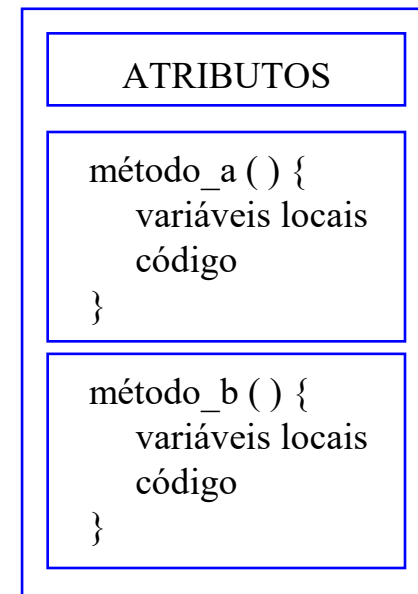
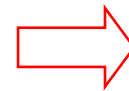
- Elemento dinâmico.
- Pode controlar acesso a atributos.
- Declaração informa assinatura de método.
- Pode ser público, privado etc.
- Pode ser método de instância ou método de classe.
- Pode ser método abstrato ou método não abstrato.

## ✓ Método abstrato:

- Método declarado mas não definido.
- Classe com método abstrato é classe abstrata.
- Não pode existir objeto (instância) de classe abstrata.

# Orientação a objetos

***method.** (1) implementation of an operation (ISO/IEC 19500-2:2012) (2) statement of how property values are combined to yield a result (IEEE 1320.2-1998 (R2004)) (3) code that is executed to perform a service (ISO/IEC 19500-1:2012).*



# Orientação a objetos



## ▼ Recomendações de projeto (*design*):

- Escolher e usar nomes que sejam significativos.
- Enfocar propósito claramente definido.
- Métodos devem ser pequenos.
- Centralizar código comum.
- Herdar métodos comuns a diferentes classes.
- Esconder estrutura interna da classe.
- Esconder estruturas de dados.
- Definir visibilidades e informá-las (público, privado etc.).
- Implementar código que facilite entendimento.

# Orientação a objetos



- Evitar acessar atributo de outra classe.
- Evitar muitas responsabilidades.
- Evitar falta de responsabilidades.
- Evitar responsabilidades não usadas.
- Evitar nomes que levem a erros de interpretação.
- Evitar abreviações confusas.
- Evitar responsabilidades não relacionadas.
- Evitar responsabilidade replicada.
- Evitar uso inadequado de herança.

# Orientação a objetos



## v Objeto:

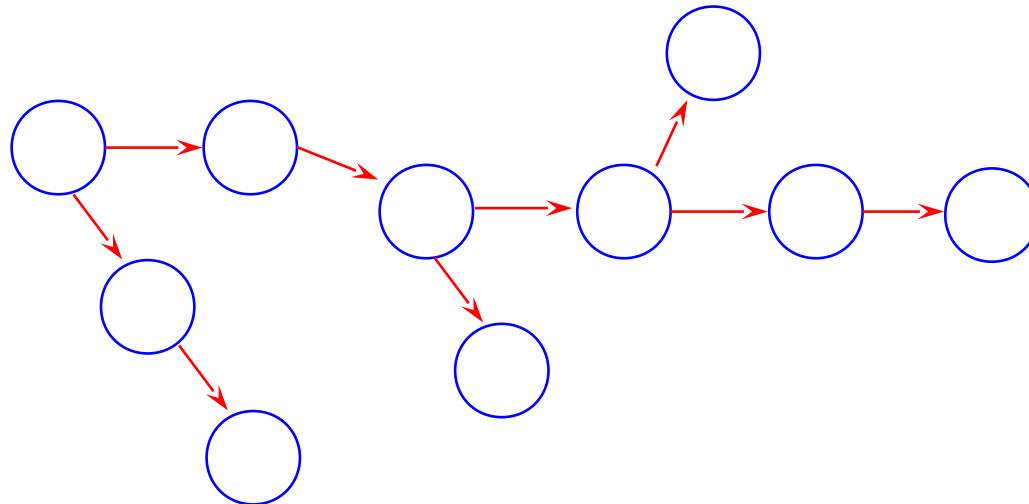
- Instância de classe.
- Podem existir vários objetos (instâncias) de uma classe.
- Ao objeto tem-se associado um identificador (*uoid*).
- Estado do objeto é tipicamente escondido.
- Estado definido por operações anteriormente executadas.
- Serviços providos por interações entre objetos.

***object.** (1) encapsulation of data and services that manipulate that data (ISO/IEC 19500-1:2012) (2) specific entity that exists in a program at runtime, in object-oriented programming (ISO/IEC/IEEE 24765:2017) (3) member of an object set and an instance of an object type (IEEE 1320.2-1998 (R2004)).*

# Orientação a objetos

## ✓ Mensagem:

- Possibilita interação entre objetos.
- Possibilita comunicação entre objetos.
- Pode ocorrer por meio de invocação a método.

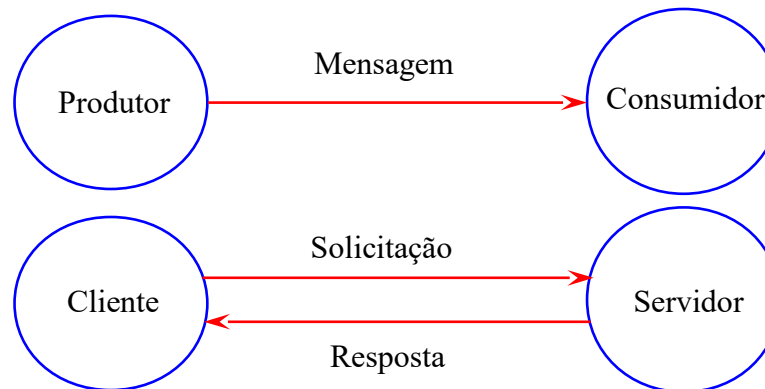




# Orientação a objetos

## ✓ Padrões de interação entre objetos:

- **Padrão produtor-consumidor** ocorre comunicação em uma direção, por exemplo, quando usuário seleciona opção em menu.
- **Padrão cliente-servidor** ocorre comunicação em duas direções, cliente solicita serviço e espera por resposta.



# Orientação a objetos



## ✓ Estado de objeto:

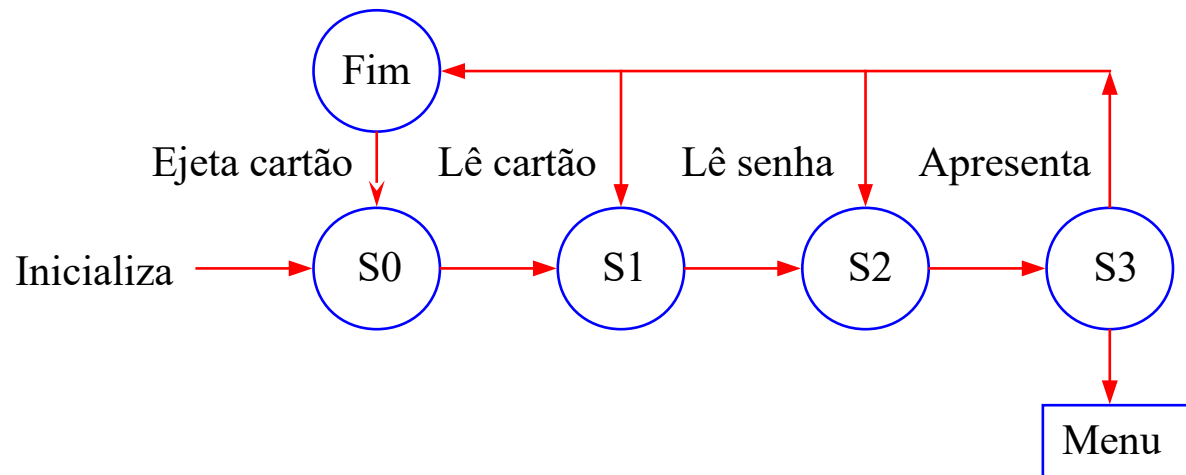
- Objeto pode passar por vários estados durante a sua vida.
- Estado é definido por valores de atributos.
- Mudança de estado decorre de mensagem.

## ✓ Evento:

- Evento pode ser descrito por classe.
- Classe que descreve evento instanciada quando de evento.
- Evento é comunicado ao interessado (*event listener*).
- Evento é processado.

# Orientação a objetos

## ▼ Máquina de estados:



# Orientação a objetos



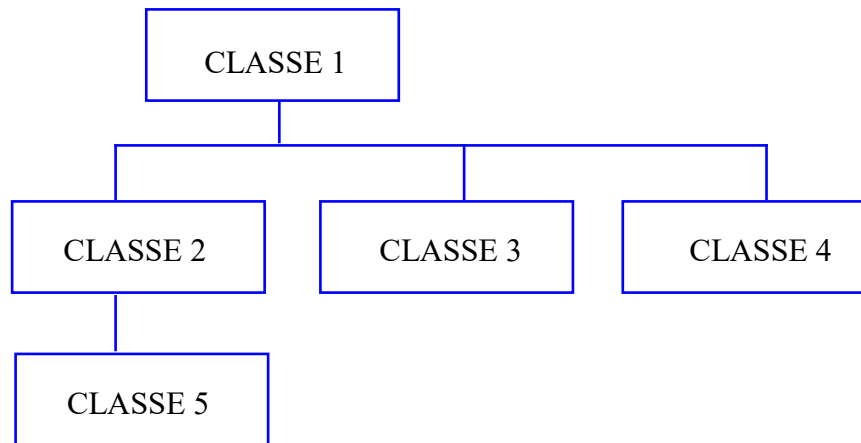
## ✓ Herança entre classes:

- Possibilita generalização e especialização.
- Promove reuso.
- Herança de atributo.
- Herança de método.
- Novo atributo.
- Novo método.
- Nova implementação de método.
- Restrição de valor de atributo.
- Modificação de visibilidade de atributo.
- Modificação de visibilidade de método.


# Orientação a objetos

## ✓ Hierarquia de classes:

- Herança possibilita construção de hierarquia de classes.
- Existem diversas hierarquias de classes.
- Promove reuso.



# Orientação a objetos



GNOME DEVELOPER

About Users Administrators **Developers** Search

## gtkmm: Class Hierarchy

Related Pages Modules Namespaces **Classes**

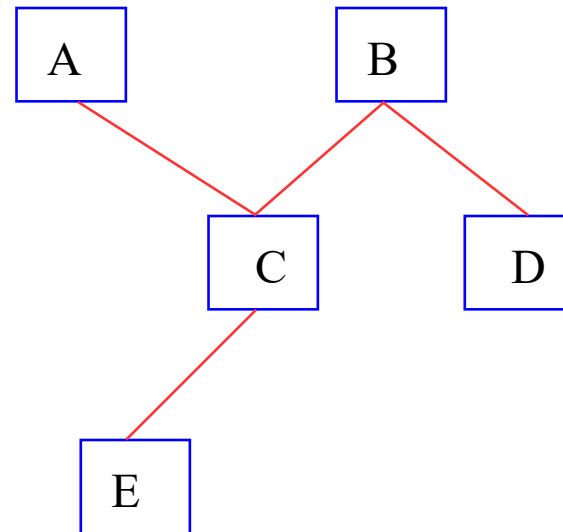
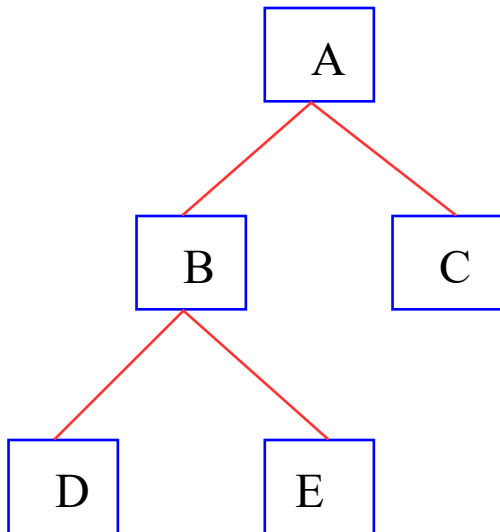
Go to the graphical class hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<b>CGdk::Color</b>	<b>Gdk::Color</b> is used to describe an allocated or unallocated color
<b>CGdk::Event</b>	
<b>CGdk::FrameTimings</b>	Object holding timing information for a single frame
<b>CGdk::PixbufFormat</b>	
<b>CGdk::Point</b>	This is a simple structure containing an x and y coordinate of a point
<b>CGdk::Rectangle</b>	<b>Gdk::Rectangle</b> is a structure holding the position and size of a rectangle
<b>CGdk::RGBA</b>	An <b>RGBA Color</b>
<b>CGdk::TimeCoord</b>	A <b>Gdk::TimeCoord</b> instance contains a single event of motion history
▼ <b>CGlib::Exception</b> [external]	
▼ <b>CGlib::Error</b> [external]	
<b>CGdk::GLError</b>	
<b>CGdk::PixbufError</b>	Exception class for <b>Gdk::Pixbuf</b> errors
<b>CGtk::BuilderError</b>	Exception class for <b>Gdk::Builder</b> errors
<b>CGtk::CssProviderError</b>	Exception class for <b>Gtk::CssProvider</b> errors
<b>CGtk::FileChooserError</b>	Exception class for <b>Gdk::FileChooser</b> errors
<b>CGtk::IconThemeError</b>	Exception class for <b>Gtk::IconTheme</b> errors

# Orientação a objetos

- ✓ Herança simples e herança múltipla:

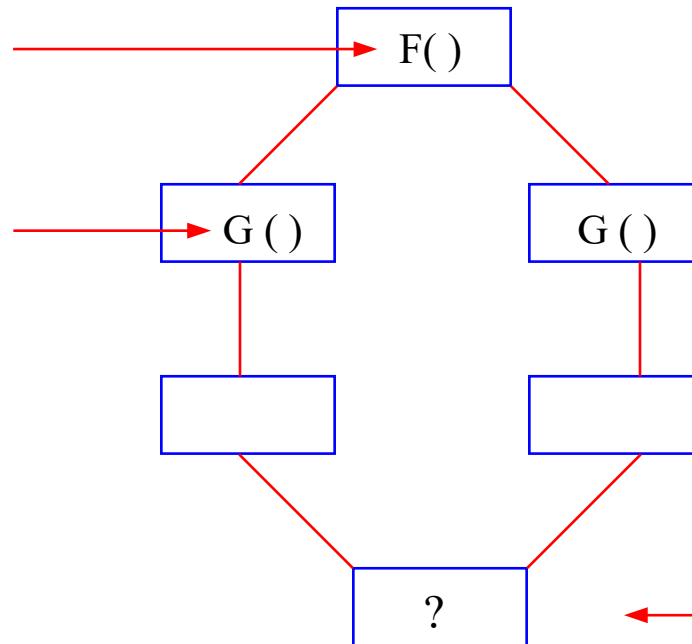


# Orientação a objetos

## ✓ Exemplo de ambiguidade:

Método herdado  
via dois ramos da  
hierarquia.

Métodos com  
mesma  
assinatura.

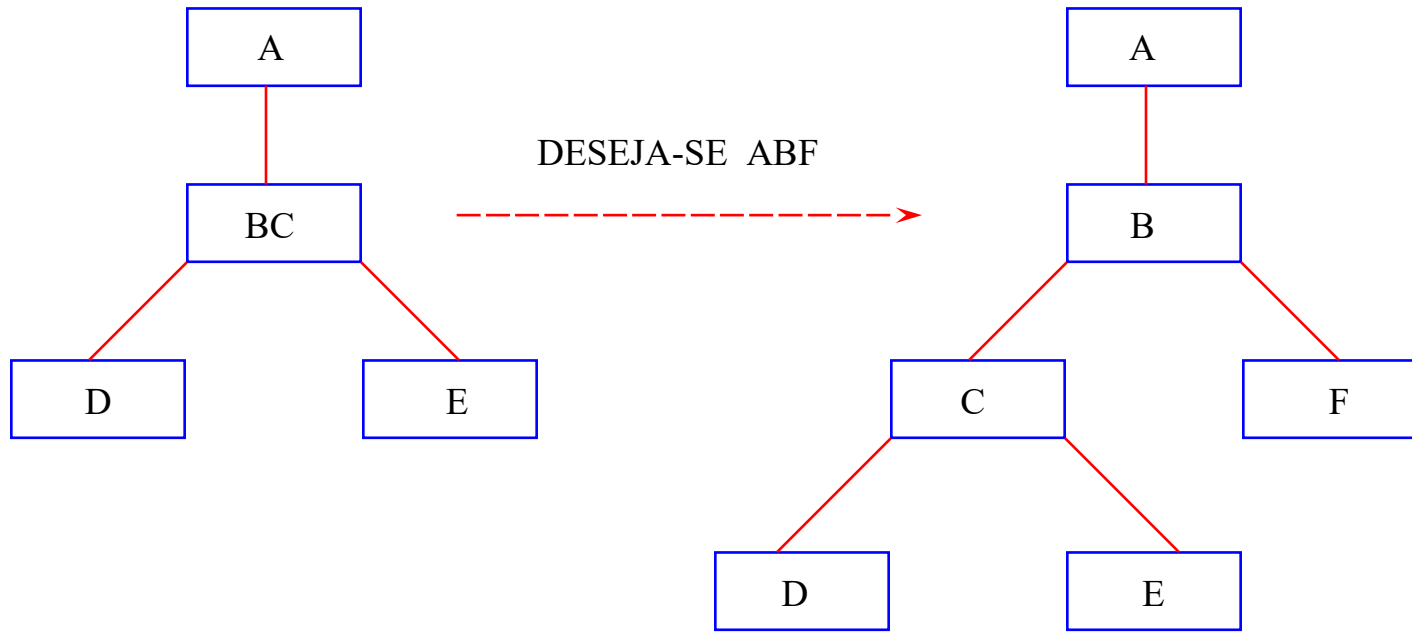


Necessário remover  
ambiguidade.



# Orientação a objetos

## ✓ Manutenção de hierarquia:



# Orientação a objetos



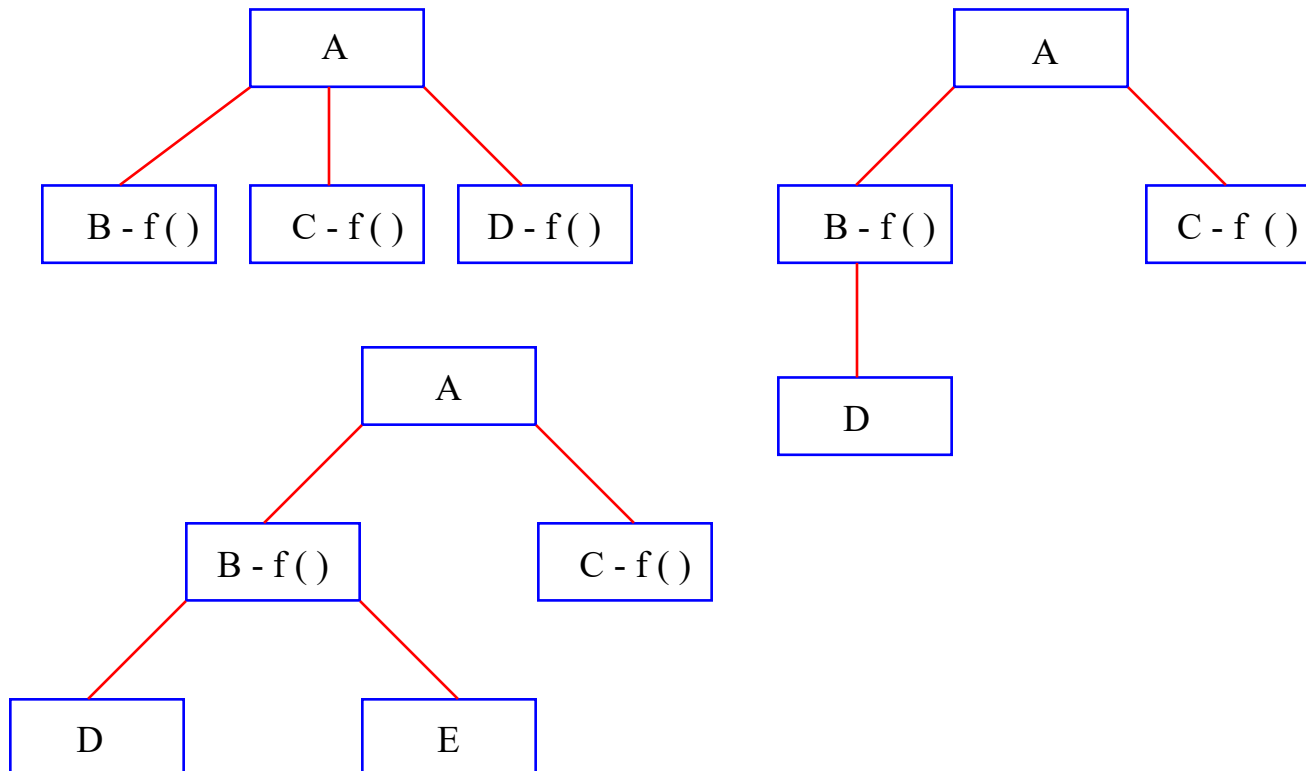
## ✓ Polimorfismo:

- Diferentes reações a um mesmo nome de mensagem.
- Mesmo nome para mesma ação lógica.

## ✓ Aspectos desejáveis em métodos similares:

- Mesmos nomes.
- Mesma ordem de argumentos.
- Mesmos tipos de argumentos.
- Mesmos tipos de valores retornados.
- Mesmas notificações de erros.

# Orientação a objetos



# Orientação a objetos



## ✓ *Static binding:*

- Tipicamente mais seguro.
- Erros identificados em tempo de compilação.
- Tipicamente mais eficiente.
- Resolução é executada uma vez.

## ✓ *Dynamic binding:*

- Tipicamente mais flexível.
- Modificações podem ser facilmente realizadas.

# Orientação a objetos



## ✓ Sobrecarga de método (*overloading*):

- Métodos com mesmo nome.
- Assinatura diferente.
- Resolução por comparação de assinaturas.

## ✓ Exemplo:

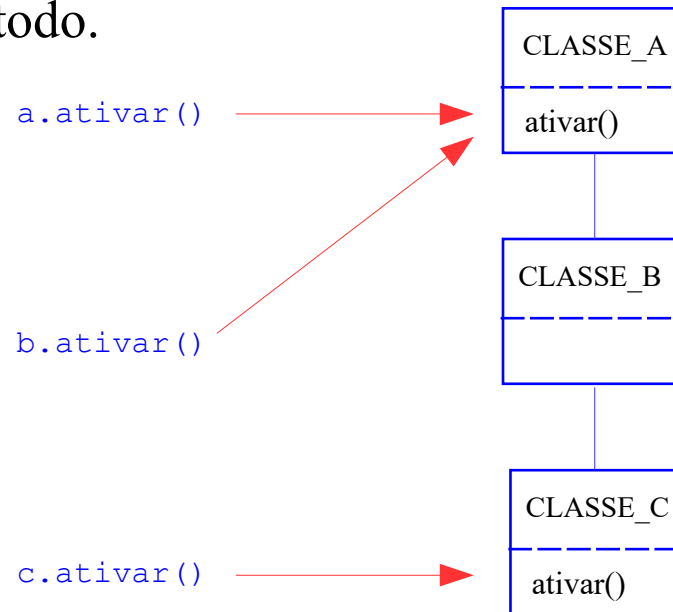
```
public CalcularMedia ( );  
public CalcularMedia ( int e );  
public CalcularMedia ( int e, int f );
```

← Diferentes  
quantidades  
ou tipos de  
parâmetros.

# Orientação a objetos

## ✓ Sobrescrita de método (*method overriding*):

- Métodos com mesma assinatura em hierarquia de classes.
- Diferentes implementações.
- Substituição do método.





# TÓPICOS ESPECIAIS



# QUALIDADE DE SOFTWARE



# Qualidade de software

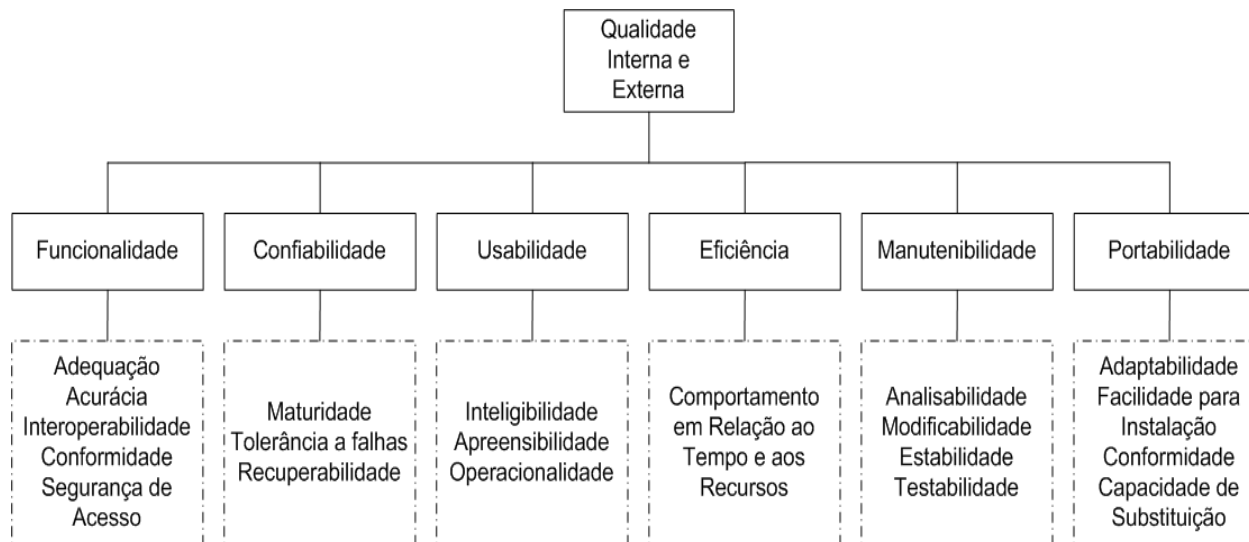


***Quality.*** (1) *degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value (ISO/IEC 25010:2011)* (2) *ability of a product, service, system, component, or process to meet customer or user needs, expectations, or requirements (ISO/IEC/IEEE 24765:2017).*

***Quality attribute.*** (1) *characteristic of software, or a generic term applying to quality factors, quality subfactors, or metric values (IEEE 1061-1998 (R2004))* (2) *requirement that specifies the degree of an attribute that affects the quality that the system or software must possess (ISO/IEC/IEEE 24765:2017).*

***Software quality.*** (1) *capability of a software product to satisfy stated and implied needs when used under specified conditions (ISO/IEC 25000:2014)* (2) *degree to which a software product meets established requirements (IEEE 730-2014 IEEE).*

# Qualidade de software



# Qualidade de software



**Quality management.** (1) *coordinated activities to direct and control an organization with regard to quality (ISO/IEC/IEEE 12207:2017) (ISO/IEC/IEEE 15288:2015) (ISO/IEC TS 24748-1:2016).*

**Quality management (QM):** *Managing activities and resources of an organization to achieve objectives and prevent nonconformances.*

**Quality assurance (QA).** (1) *part of quality management focused on providing confidence that quality requirements will be fulfilled (ISO/IEC/IEEE 12207:2017) (ISO/IEC/IEEE 15288:2015) (ISO/IEC TS 24748-1:2016).*

**Quality control (QC).** (1) *set of activities designed to evaluate the quality of developed or manufactured products (ISO/IEC/IEEE 24765:2017) (2) monitoring service performance or product quality, recording results, and recommending necessary changes (ISO/IEC/IEEE 24765c:2014).*

# Qualidade de software



## ✓ Teste:

- Não pode assegurar ausência de defeitos.
- Realizado com determinado propósito.
- Bem sucedido se encontra defeito.
- Experimento controlado.
- Execução de código de software (teste dinâmico).
- Baseado em casos de teste.
- Pode ser manual ou automatizado.
- Precisa ser projetado, planejado e aprovado.

# Qualidade de software



## ✓ Possíveis fases ou classes de teste:

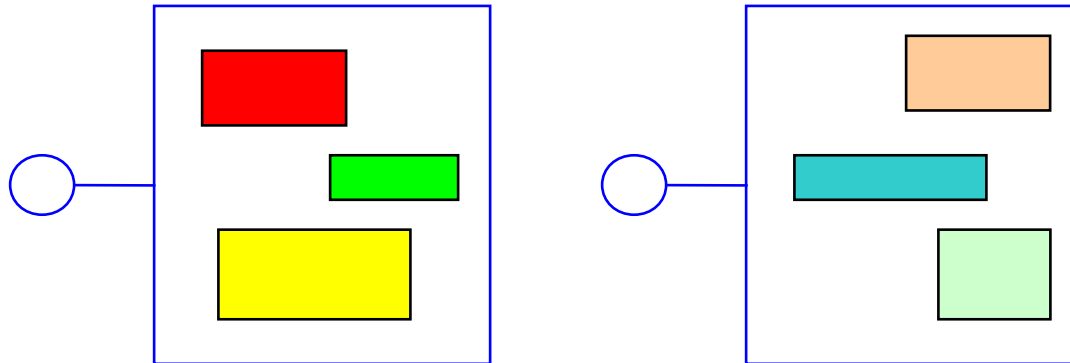
- Teste de unidade (*unit test*).
- Teste de integração.
- Teste fumaça (*smoke test*).
- Teste de sistema.
- Teste de aceitação
- Teste *alpha*.
- Teste *beta*.
- Teste de regressão.

# Qualidade de software



## ✓ Teste de unidade:

- Teste de unidade (*unit*) de modo individual.
- Teste de baixo nível.
- Tipicamente estrutural.



# Qualidade de software



## ✓ Elementos em teste de unidade:

- Nível de cobertura.
- Teste com parâmetro normal, em limite e fora de limite.
- Acesso a estruturas de dados.
- Acesso a arquivos.
- Término normal e anormal de *loops* e recursões.
- Tratamento de erro.
- Temporização.
- Sincronização.
- Dependência de hardware.

# Qualidade de software

## ✓ Arcabouço para teste de unidade:

- Padroniza estrutura de teste de unidade.
- Facilita reúso.
- Existem em diversas linguagens de programação.

C++ [ edit ]

Name	License	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping
Aeryn		No	Yes	Yes	No	No	Yes	Yes	Yes	Yes
API Sanity Checker	GNU LGPL	Yes	Yes (spectypes)	Yes (spectypes)	Yes					
ATF	BSD						Yes	Yes	Yes	Yes
Bandit	MIT	No (describe/it)	Yes (describe)	Yes (Nested describe)	No	No	Yes	Yes	No	Yes (Nested describe)
Boost Test Library	Boost	Yes <sup>[83]</sup>	Yes <sup>[84]</sup>	Yes <sup>[85][86]</sup>	Yes	With additional library "Turtle" <sup>[87]</sup>	Yes	User decision	Yes	Suites and labels

Fonte: Wikipedia contributors. List of unit testing frameworks. Wikipedia, The Free Encyclopedia. 2020.



# Qualidade de software



## ✓ Teste de integração:

- Pode ser considerado extensão de teste de unidade.
- Testa unidades integradas.
- Identifica erros revelados quando da integração.
- Pode ser realizado em diversas fases de desenvolvimento.
- Pode focar cenários de uso.
- Cenário de uso reflete requisito.
- Integração de unidades necessárias ao cenário de uso.

# Qualidade de software



## ✓ Estratégias de teste de integração:

- *Bottom-up*.
- *Top-down*.
- Híbrida.

## ✓ Estratégia *bottom-up*:

- Unidades de mais baixo nível são primeiro integradas.
- Minimiza a necessidade de *stubs*.
- Há necessidade de *drivers*.
- Pode demorar até haver algo a apresentar aos interessados.



# Qualidade de software



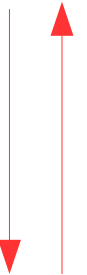
## ✓ Estratégia *top-down*:

- Unidades de mais alto nível são integradas primeiro.
- Minimiza-se a necessidade de *drivers*.
- Aumenta a necessidade de *stubs*.



## ✓ Estratégia híbrida:

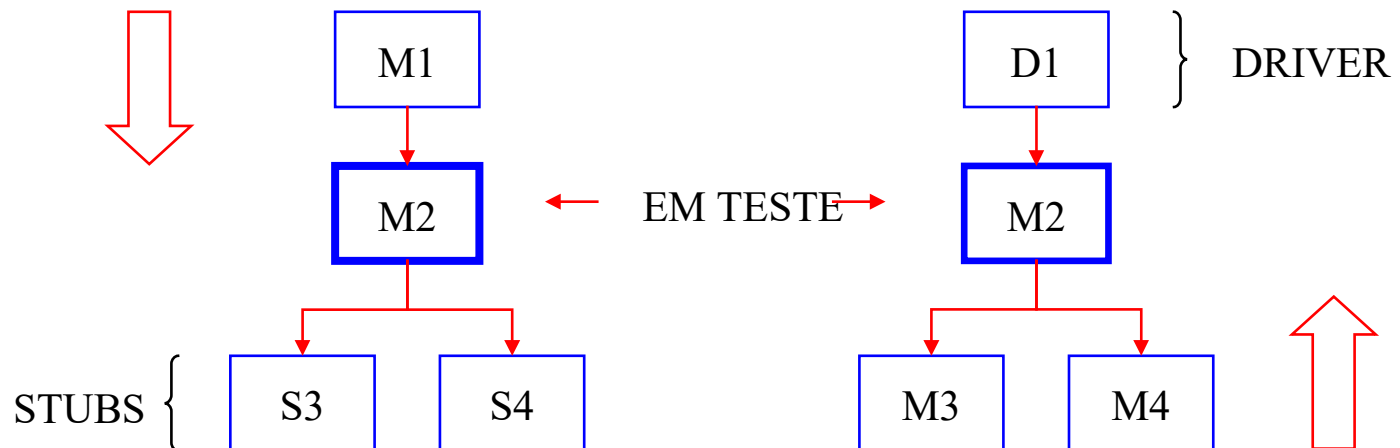
- Pode enfocar funcionalidades.
- Pode prover versões com funcionalidades limitadas.
- Minimiza *drivers* e *stubs*.
- Menos sistemática do que outras estratégias.
- Pode requerer maior quantidade de testes de regressão.



# Qualidade de software

## ✓ *Stub e driver:*

- *Stub* substitui servidor.
- *Stub* tipicamente usado na integração *top-down*.
- *Driver* substitui cliente.
- *Driver* tipicamente usado em integração *bottom-up*.



# Qualidade de software



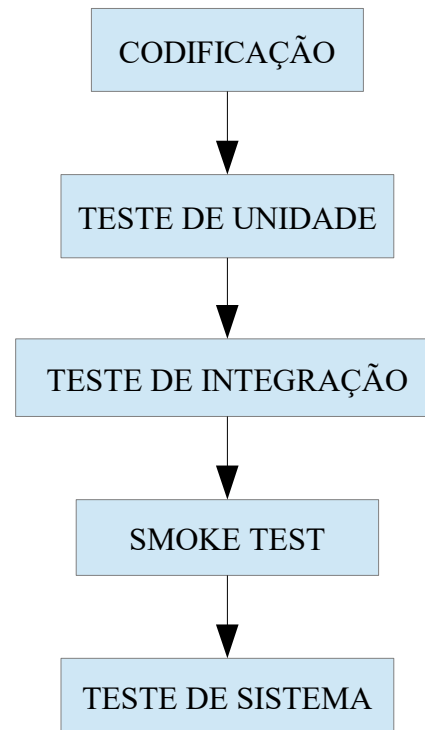
## v Teste fumaça (*smoke test*):

- Teste não exaustivo.
- Teste de principais funcionalidades.
- Expõe principais defeitos.
- Procura avaliar se o sistema é estável.
- Realizado antes de testes formais.
- Testes formais realizados se *smoke test* aprovar o item.
- Pode ser executado manualmente ou automatizado.
- Desenvolvedores são informados sobre os casos de teste.

# Qualidade de software

Um *smoke test* é um conjunto de casos de teste através do qual é possível estabelecer que o sistema está estável e que as principais funcionalidades estão presentes e funcionam em condições normais de uso.

Craig & Jaskiel.



# Qualidade de software



## ✓ Atividades em processo de teste:

- Definir estratégia de teste e planejar teste.
- Identificar e criar casos de teste.
- Identificar e criar procedimento de teste.
- Definir como realizar teste.

## ✓ Documentação de teste:

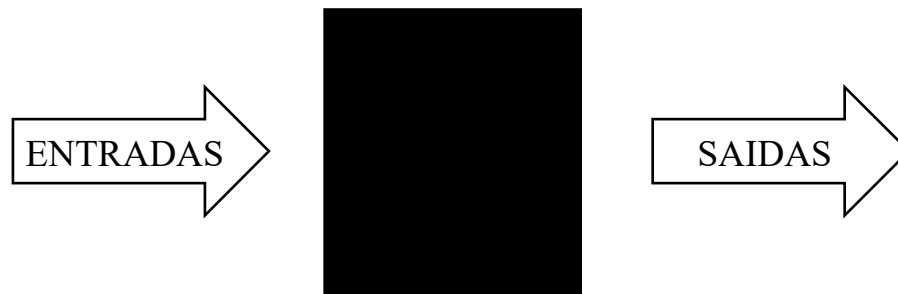
- Escopo.
- Descrição de ambiente de teste.
- Informação sobre atividades, entradas e saídas.
- Informação sobre atividades em falha ou interrupção.

# Qualidade de software



## ✓ Método caixa preta (*black box*):

- Enforca entradas e saídas.
- Ignora mecanismos internos.
- Falhas reveladas por resultados gerados.
- Pode enforçar especificações de funcionalidades.

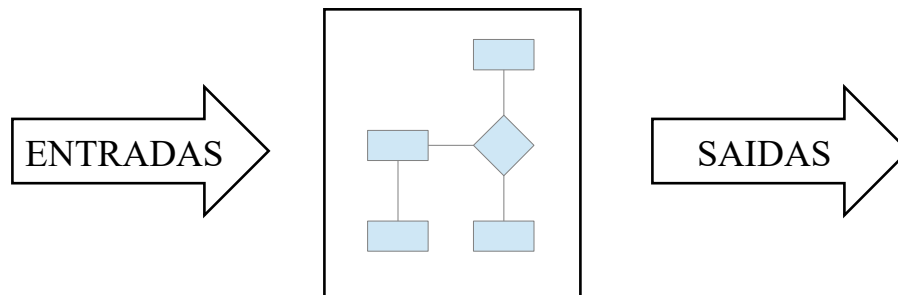




# Qualidade de software

## ✓ Método caixa branca (*white box*):

- Enfoca estrutura (mecanismos internos).
- Teste projetado considerando-se estrutura interna.
- Pode ser suportado por critério de cobertura.
- Procura atingir cobertura definida.



# Qualidade de software



## ✓ Critério de cobertura:

- Tipicamente usado em teste caixa branca (estrutural).
- Auxilia no projeto (*design*) de teste.
- Auxilia na avaliação de teste.
- Satisfazer critério não garante ausência de defeitos.
- Escolha de critério depende de recursos disponíveis.

# Qualidade de software



## ✓ Exemplos de critérios de cobertura:

- Cobertura de linhas de código.
- Cobertura de ativação e retorno.
- Cobertura da interface com os usuários.
- Cobertura de caminhos.
- Cobertura de estruturas de dados.

## ✓ Cobertura de linhas de código:

- Critério de cobertura simples.
- Percentual de linhas de código executadas.
- Procura-se maximizar o percentual de linhas executadas.

# Qualidade de software



## ✓ Cobertura de ativação e retorno:

- Exercitar funções com dados válidos e dados inválidos.
- Exercitar formas de retornar.
- Exercitar exceções.
- Exercitar cancelamento de execução.
- Exercitar mensagens que podem ser geradas.

# Qualidade de software



## ✓ Cenário:

- Descrição de situação do sistema em produção.
- Definição de resposta requerida do sistema.

## ✓ Cenário pode capturar:

- Interações com usuários.
- Processamento que precisa acontecer em dado instante.
- Situação de pico.
- Modificação que pode precisar ser feita.
- Situação que sistema precisa ser projetado para lidar.

# Qualidade de software



## ✓ Classes de cenários:

- Funcional.
- Não funcional.

## ✓ Cenário funcional:

- Definido em termos de sequência de eventos externos.
- Sistema deve responder de determinado modo.
- Pode ser derivado de caso de uso.

## ✓ Cenário não funcional:

- Baseado em atributo de qualidade não funcional.
- Como o sistema deve reagir a certa mudança no ambiente.

# Qualidade de software



## ✓ Possíveis fontes de cenários:

- Requisitos funcionais.
- Experiência prévia no domínio, nas tecnologias etc.
- Interessados (*stakeholders*).

## ✓ Definindo um interessado (*stakeholder*):

- Tem interesse no software sendo desenvolvido.
- Pode ser pessoa.
- Pode ser grupo.
- Pode ser organização.
- Tipicamente *stakeholder* representa classe de pessoas.

# Qualidade de software



## ✓ Documentação de cenário funcional:


- Nome do cenário.
- Breve descrição do que o cenário pretende ilustrar.
- Estado do sistema antes da ocorrência do cenário.
- Descrição de ambiente de execução.
- Descrição do que causa o cenário (estímulos, dados etc.).
- Descrição de como o sistema deve responder ao cenário.





# DOCUMENTAÇÃO DE SOFTWARE

# Documentação de software



**documentation.** (1) collection of documents on a given subject (ISO/IEC/IEEE 24765:2017) (2) written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results (ISO/IEC/IEEE 26531:2015) (3) process of generating or revising a document (ISO/IEC/IEEE 24765:2017) (4) information that explains how to use software, devices, applications, or services (ISO/IEC/IEEE 26513:2017).

**document.** (1) uniquely identified unit of information for human use (ISO/IEC/IEEE 15289:2017) (2) to create a document as in (1) (ISO/IEC/IEEE 24765:2017) (3) to add comments to a computer program (ISO/IEC/IEEE 24765:2017) (4) information and the medium on which it is contained (ISO/IEC 29110-4-3:2018).

# Documentação de software



- ✓ Exemplos de classes de documentação:
  - Documentação de produto (sistema e usuário).
  - Documentação de processo.
  - Documentação interna e documentação externa.
- ✓ Exemplos de documentação de software:
  - Documentação de requisitos.
  - Documentação de desenho (*design*) e arquitetura.
  - Documentação de código fonte (*source code*).
  - Documentação de teste.
  - Documentação de manutenção.

# Documentação de software



## ✓ Exemplos de atividades:

- Prover informação necessária ao entendimento de código.
- Atentar para clareza de código.
- Atentar para facilidade de leitura de código.
- Adotar padrão de codificação (*coding standard*).
- Adotar guia de codificação (*guideline*).
- Comentar código.
- Revisar código.
- Documentar interface.
- Publicar documentação.

# Documentação de software



## C++ Core Guidelines

December 8, 2019

Editors:

- Bjarne Stroustrup
- Herb Sutter

In: Introduction  
P: Philosophy  
I: Interfaces  
F: Functions  
C: Classes and class hierarchies  
Enum: Enumerations  
R: Resource management  
ES: Expressions and statements  
Per: Performance  
CP: Concurrency  
E: Error handling  
Con: Constants and immutability  
T: Templates and generic programming  
CPL: C-style programming  
SF: Source files  
SL: The Standard library

# Documentação de software



## Coding Standards

### Contents of this section:

- What are some good C++ coding standards?
- Are coding standards necessary? Are they sufficient?
- Should our organization determine coding standards from our C experience?
- What's the difference between `<CXX>` and `<CXX.h>` headers?
- Should I use `using namespace std` in my code?
- Is the `?:` operator evil since it can be used to create unreadable code?
- Should I declare locals in the middle of a function or at the top?
- What source-file-name convention is best? `foo.cpp`? `foo.C`? `foo.cc`?
- What header-file-name convention is best? `foo.H`? `foo.hh`? `foo.hpp`?
- Are there any lint-like guidelines for C++?
- Why do people worry so much about pointer casts and/or reference casts?
- Which is better: identifier names `that_look_like_this` or identifier names `thatLookLikeThis`?
- Are there any other sources of coding standards?
- Should I use "unusual" syntax?
- What's a good coding standard for using global variables?

# Documentação de software

## ✓ Ferramentas para documentação:

- Documentação pode ser construída usando-se ferramenta.





# TÉCNICAS DE MODULARIZAÇÃO



# Técnicas de modularização



## v Elementos em sistema de software:

1. Sistema
2. Subsistema
3. Programa
4. Módulo
5. Classe
6. Função
7. Bloco de código
8. Linha de código



Complexidade decrescente.

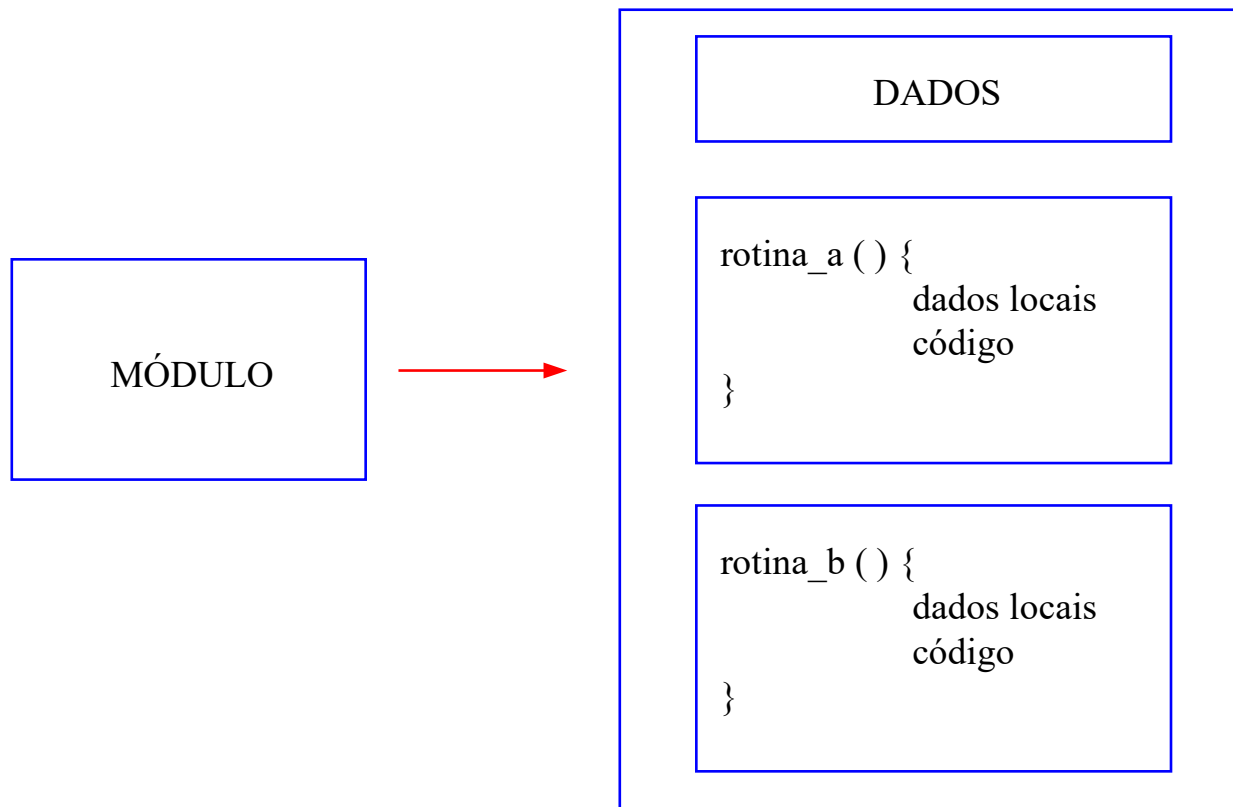
# Técnicas de modularização

## v Módulo:

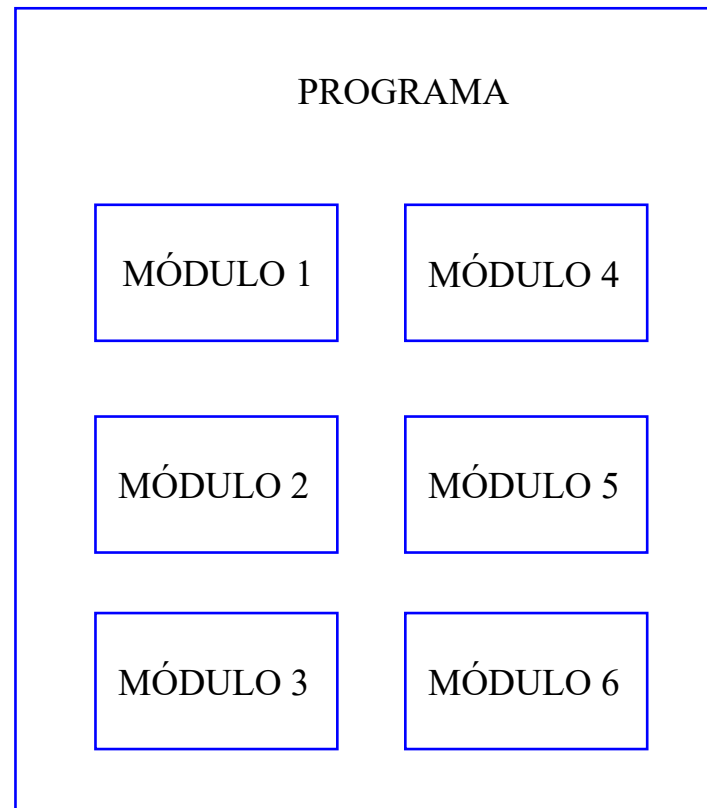
- Elemento auto-contido com implementação delimitada.
- Promove encapsulamento e provê interfaces definidas.
- Pode ser independentemente desenvolvido.
- Tipicamente é item de compilação independente.
- Tipicamente podem ser combinados por ligação (*link*).

***module.** (1) program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading (ISO/IEC 19506:2012) (2) logically separable part of a program (ISO/IEC 19506:2012) (3) set of source code files under version control that can be manipulated together as one (ISO/IEC/IEEE 24765:2017) (4) collection of both data and the routines that act on it (ISO/IEC/IEEE 24765:2017).*


# Técnicas de modularização



# Técnicas de modularização



# Técnicas de modularização



**encapsulation.** (1) software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module (ISO/IEC/IEEE 24765:2017) (2) concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization (IEEE 1320.2-1998 (R2004)) (3) idea that a module has an outside that is distinct from its inside, that it has an external interface and an internal implementation (ISO/IEC/IEEE 24765:2017).

**information hiding.** (1) software development technique in which each module's interfaces reveal as little as possible about the module's inner workings and other modules are prevented from using information about the module that is not in the module's interface specification (ISO/IEC/IEEE 24765:2017) (2) containment of a design or implementation decision in a single module so that the decision is hidden from other modules (ISO/IEC/IEEE 24765:2017).

# Técnicas de modularização



## ✓ Conector:

- Comunicação entre módulos ocorre via conectores.
- Ponto onde pode-se passar controle de execução.
- Espaço para dados a serem trocados.

## ✓ Exemplos de conectores:

- Função.
- Variável global ou estática.
- Variável pública.
- Arquivo.
- Ponto de ativação e retomada em exceções.

# Técnicas de modularização



## ✓ Acoplamento:

- Decorre do relacionamento entre módulos.
- Procura-se minimizar o acoplamento.

## ✓ Acoplamento determinado segundo:

- Quantidade de conectores.
- Tamanhos dos conectores.
- Coesões dos conectores.
- Tipos dos elementos que compõem os conectores.
- Complexidade dos conectores.
- Instante e modo de conexão.

# Técnicas de modularização




## ▼ Interface:

- Informa elementos providos.
- Informa elementos requeridos.
- Informa elementos visíveis externamente.
- Modificações de interface são desencorajadas.
- Implementação realiza interface.
- Possível mudar implementação de interface.
- Sintaxe (regras e restrições aplicáveis).
- Semântica (significado).
- Protocolo (como usar a interface).

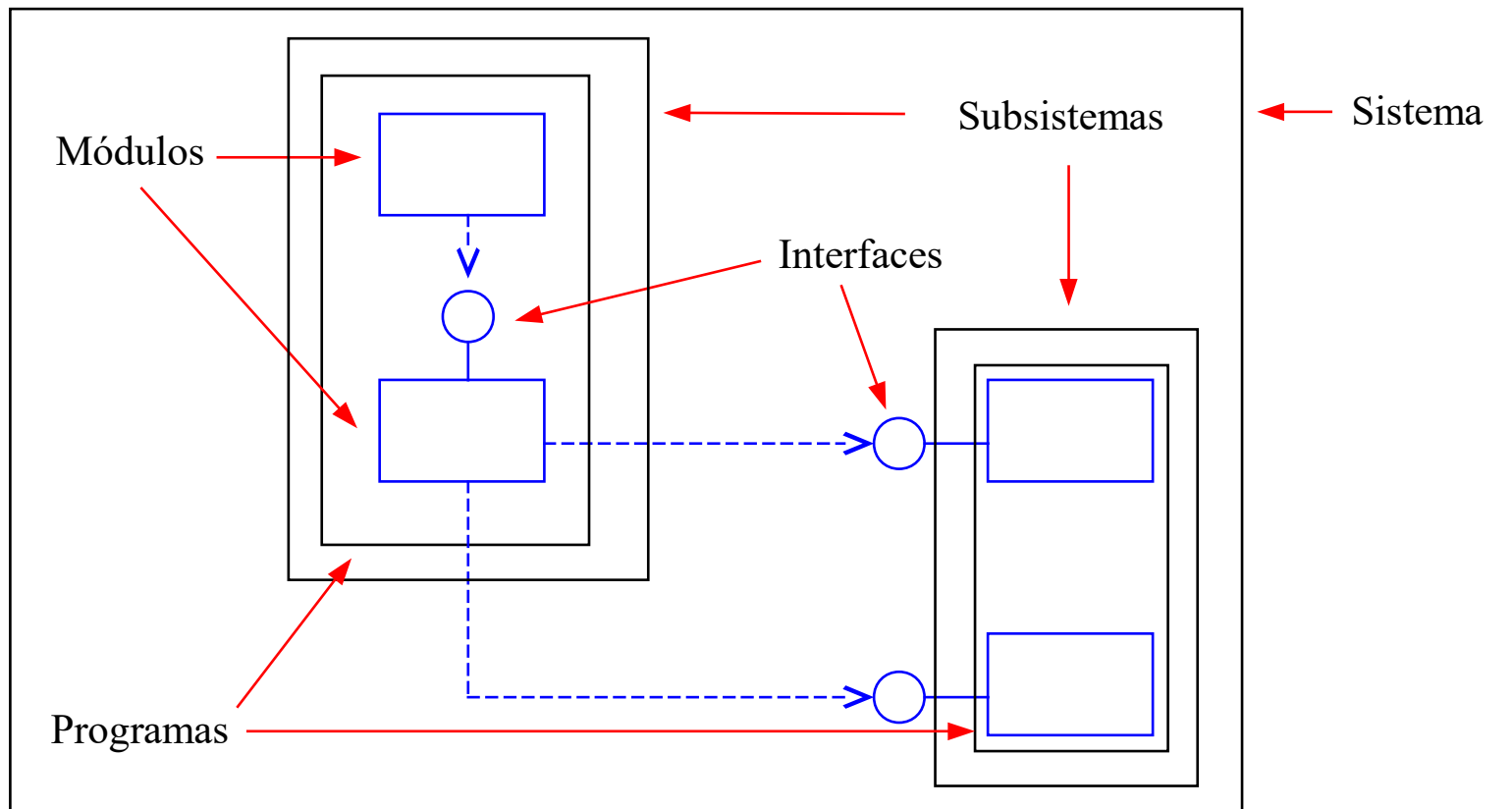


# Técnicas de modularização



*interface. (1) shared boundary between two functional units, defined by various characteristics pertaining to the functions, physical signal exchanges, and other characteristics (ISO/IEC 2382:2015) (2) hardware or software component that connects two or more other components for the purpose of passing information from one to the other (ISO/IEC/IEEE 24765:2017) (3) to connect two or more components for the purpose of passing information from one to the other (ISO/IEC/IEEE 24765:2017) (4) to serve as a connecting or connected component as in (2) (ISO/IEC/IEEE 24765:2017) (5) declaration of the meaning and the signature for a property or constraint (IEEE 1320.2-1998 (R2004)) (6) abstraction of the behavior of an object that consists of a subset of the interactions of that object together with a set of constraints on when they can occur (ISO/IEC 10746-2:2009) (7) named set of operations that characterize the behavior of an entity (ISO/IEC 19506:2012).*

# Técnicas de modularização





# TÉCNICAS DE DESENHO



# PADRÕES DE ARQUITETURA

# Padrões de arquitetura



## ✓ Arquitetura de software:

- Sistema de software tem arquitetura.
- Sistema de software tem componentes e relacionamentos.
- Descreve estrutura de software.
- Relacionamentos entre elementos de software.
- Tipicamente omite detalhes.
- Não necessariamente é documentada.
- Pode existir independente de registro da mesma.
- Pode ser mais ou menos apropriada aos requisitos.
- Deve ser avaliada considerando-se requisitos de software.

# Padrões de arquitetura



## ✓ Padrão de arquitetura:

- Estrutura previamente identificada.
- Soluciona problema frequente em determinado contexto.
- Aplicável em diferentes níveis de abstração.
- Promove qualidade.
- Promove entendimento de software.
- Promove especificação de software.
- Promove construção de software.
- Promove documentação de software.

# Padrões de arquitetura



## ✓ Características de padrão de arquitetura:

- Solução de problema real.
- Solução comprovada.
- Solução não trivial.
- Solução tipicamente obtida indiretamente.
- Diversos níveis de abstração.
- Diversos níveis de detalhamento.
- Não captura apenas princípios abstratos ou estratégias.
- Não descreve teorias ou especulações.

# Padrões de arquitetura




## ✓ Documentação de padrão de arquitetura:

- Nome.
- Descrição resumida.
- Outros nomes pelos quais o padrão é conhecido.
- Problema.
- Objetivo.
- Contexto onde ocorre problema.
- Contexto onde ocorre solução.
- Motivação.
- Descrição de solução.



# Padrões de arquitetura

- 
- Exemplos de uso.
  - Contexto resultante.
  - Consequências de uso.
  - Nomes de padrões relacionados.
  - Usos conhecidos.
  - Situações onde o padrão pode ser usado.
  - Representação gráfica.
  - Responsabilidades de classes no padrão.
  - Responsabilidades de objetos no padrão.
  - Descrição de como os elementos integrantes colaboram.
  - Detalhes de implementação.

# Padrões de arquitetura



## ✓ Classes de padrões de arquitetura:

- Estilo de arquitetura (*architectural style*).
- Padrão de projeto (*design pattern*).
- Idioma (*idiom*).
- Anti-padrão (*anti-pattern*).

## ✓ Estilo de arquitetura:

- Estratégia de alto nível que descreve estrutura de sistema.
- Sugere conjunto de subsistemas.
- Sugere responsabilidades de subsistemas.
- Sugere relacionamentos entre subsistemas.

# Padrões de arquitetura



## ✓ Exemplos de estilos de arquitetura:

- *Blackboard*.
- *Client-server* (cliente-servidor).
- *Data-centric*.
- *Event-driven* (orientado a eventos).
- *Layered* (camadas).
- *Microservices architecture*.
- *Monolithic* (monolítico).
- *Peer-to-peer (P2P)*.
- *Pipes and filters* (dutos e filtros).
- *Service-oriented* (orientado a serviços).

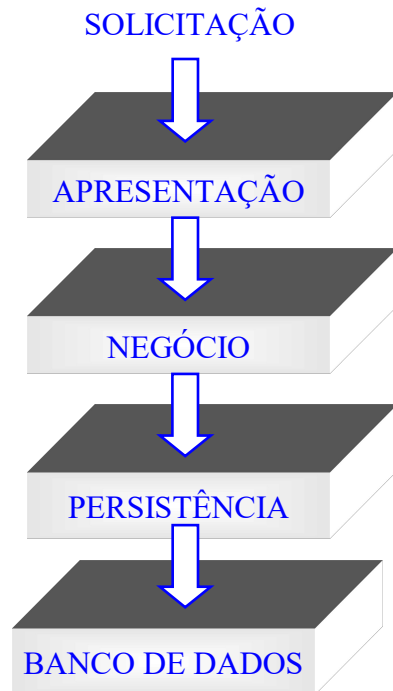
# Padrões de arquitetura



## ✓ Estilo de arquitetura em camadas:

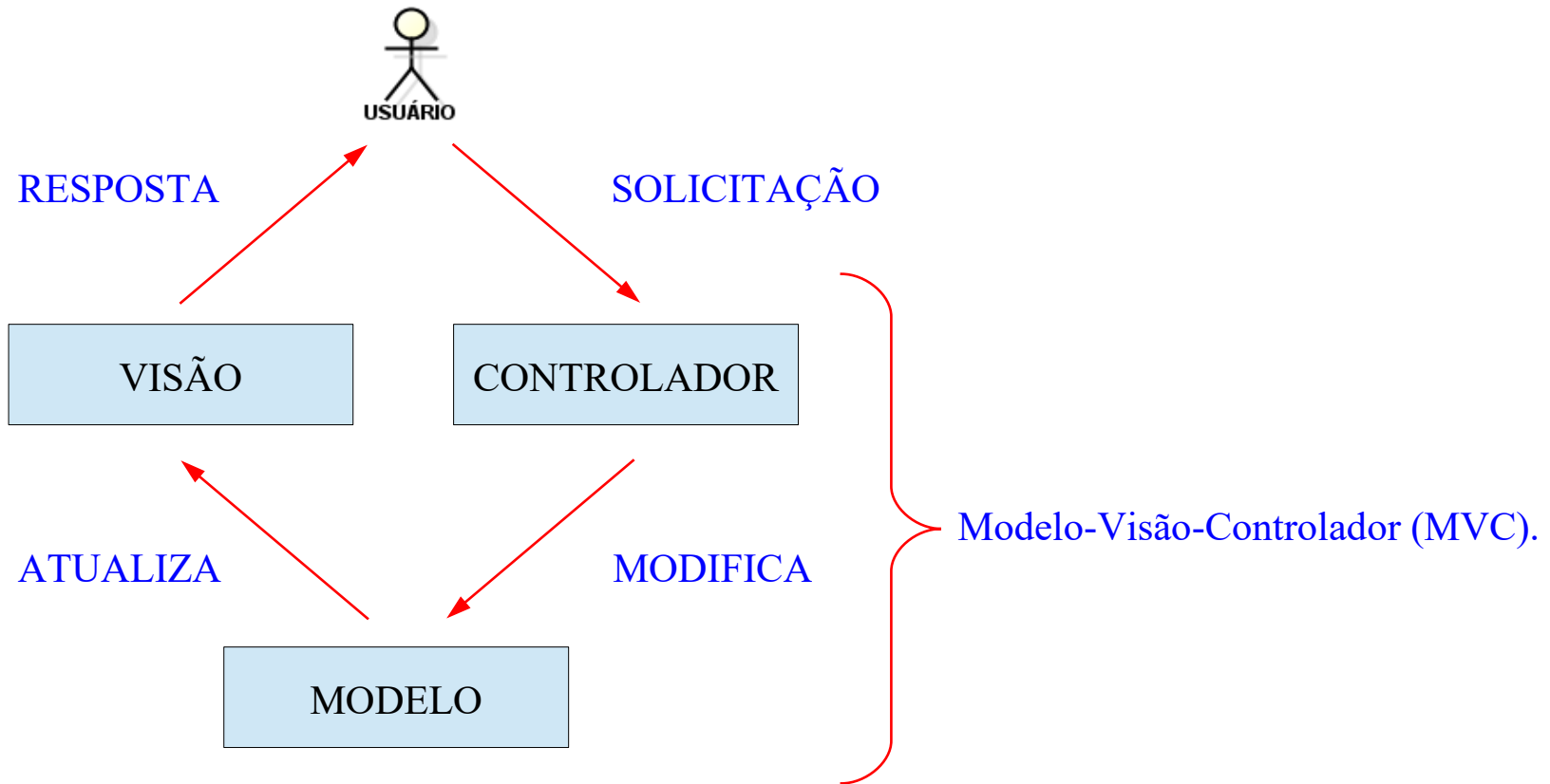
- Estrutura composta por camadas horizontais.
- Cada camada tem determinadas responsabilidades.
- Promove separação de responsabilidades.
- Promove isolamento de efeitos de modificações.
- Solicitações ocorrem entre camadas.
- Solicitação ocorre para camada imediatamente abaixo.
- Salto entre camadas é violação do estilo.
- Solicitação para camada acima é violação do estilo.

# Padrões de arquitetura



Exemplos de camadas.

# Padrões de arquitetura



# Padrões de arquitetura



## ✓ Padrão de projeto (*design pattern*):

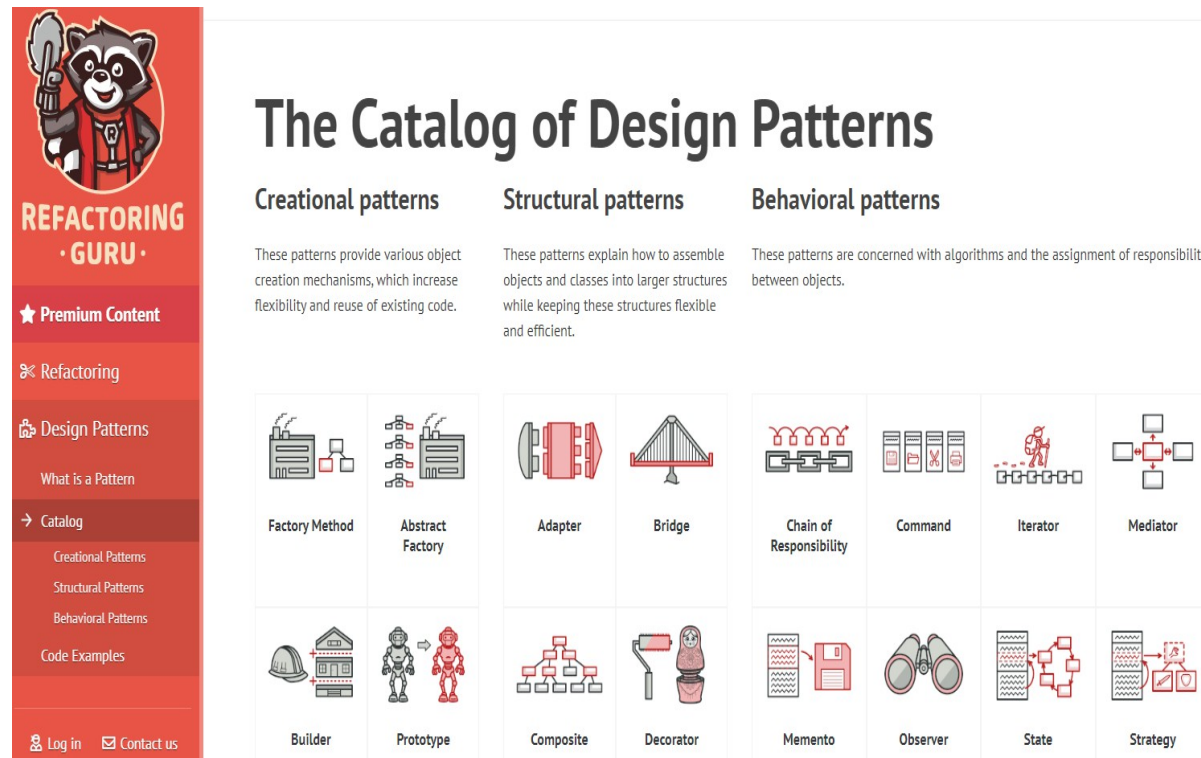
- Descreve estrutura.
- Descreve comportamento.
- Não afeta arquitetura do sistema como um todo.
- Define microarquitetura.
- Tipicamente enfoca módulos ou classes.

## ✓ Catálogo de padrões de projeto:

- Conjunto de padrões relacionados.
- Padrões podem ser organizados segundo classes.

# Padrões de arquitetura

## ✓ Exemplo de catálogo de padrões de projeto:





# Padrões de arquitetura

## ✓ Padrões de projeto GOF:

- Livro clássico.
- Soluções para problemas específicos.
- Total de 23 padrões de projeto.
- Padrões organizados em classes.
- Padrões para criar objetos.
- Padrões estruturais.
- Padrões de comportamento.



# Padrões de arquitetura



## ✓ Padrões para criar objetos:

- Padrões para abstrair processo de instanciação.
- Promovem menor dependência de como criar objetos.
- Aumentam uso de composição de objetos.
- Encapsulam conhecimento de classes concretas.
- Escondem como instâncias de classes são criadas.

***Abstract factory***

*Groups object factories that have a common theme.*

***Builder***

*Constructs complex objects by separating construction and representation.*

***Factory method***

*Creates objects without specifying the exact class to create.*

***Prototype***

*Creates objects by cloning an existing object.*

***Singleton***

*Restricts object creation for a class to only one instance.*

FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

FONTE: Wikipedia contributors. Design Patterns. Wikipedia, The Free Encyclopedia. 2020.

# Padrões de arquitetura



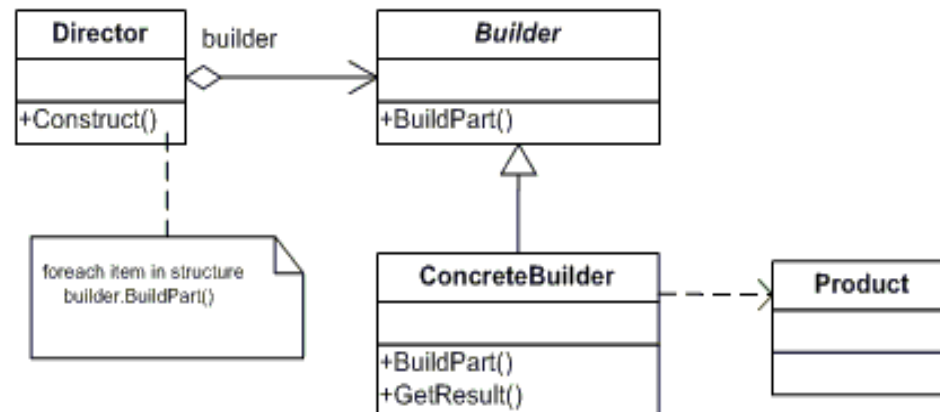
## ✓ Padrão *Builder*:

- Promove separação da construção de objeto complexo da sua representação.

## ✓ Aplicabilidade:

- Algoritmo para construir objeto complexo deve independer das partes que compõem o objeto e de como são montadas.
- Processo de construção deve possibilitar diferentes representações para o objeto.

# Padrões de arquitetura



FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

# Padrões de arquitetura



## ✓ Padrão *Singleton*:

- Garante que classe só tem uma instância.
- Provê ponto de acesso global à instância da classe.

Singleton
-instance : Singleton
-Singleton()
+Instance() : Singleton

## ✓ Aplicabilidade:

- Quando só pode existir uma instância de uma classe e que precisa estar acessível aos clientes de um ponto de acesso conhecido.

# Padrões de arquitetura



## ▼ Padrões estruturais:

- Descrevem como classes e objetos são compostos para formar estruturas maiores.

<b><i>Adapter</i></b>	<i>Allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.</i>
<b><i>Bridge</i></b>	<i>Decouples an abstraction from its implementation so that the two can vary Independently.</i>
<b><i>Composite</i></b>	<i>Composes zero-or-more similar objects so that they can be manipulated as one object.</i>
<b><i>Decorator</i></b>	<i>Dynamically adds/overrides behaviour in an existing method of an object.</i>
<b><i>Facade</i></b>	<i>Provides a simplified interface to a large body of code.</i>
<b><i>Flyweight</i></b>	<i>Reduces the cost of creating and manipulating a large number of similar objects.</i>
<b><i>Proxy</i></b>	<i>Provides a placeholder for another object to control access, reduce cost, and reduce complexity.</i>

FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

FONTE: Wikipedia contributors. Design Patterns. Wikipedia, The Free Encyclopedia. 2020.

# Padrões de arquitetura



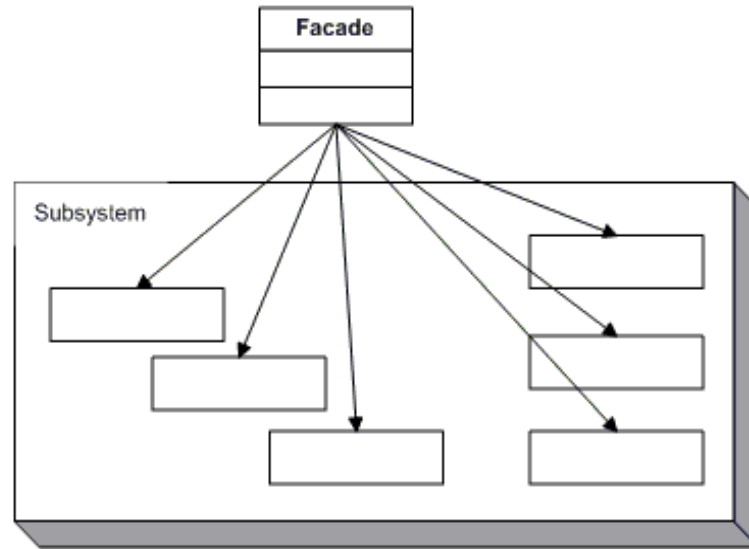
## ✓ Padrão *Facade*:

- Provê interface para conjunto de interfaces em subsistema.
- Define interface que facilita uso de subsistema.

## ✓ Aplicabilidade:

- Quando é necessário prover interface simples para subsistema complexo.
- Quando existe número elevado de dependências entre clientes e as implementações de uma abstração.
- Quando se deseja organizar subsistemas em camadas.

# Padrões de arquitetura



FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.



# Padrões de arquitetura



## ✓ Padrões de comportamento:

- Preocupam-se com algoritmos e atribuições de responsabilidades entre objetos.
- Caracterizam fluxos de controle complexos difíceis de seguir em tempo de execução.
- Padrões de comportamento de classe usam herança para distribuir comportamento entre classes.
- Padrões de comportamento de objetos usam composição para distribuir comportamento.

# Padrões de arquitetura



<b>Chain of responsibility</b>	<i>Delegates commands to a chain of processing objects.</i>
<b>Command</b>	<i>Creates objects which encapsulate actions and parameters.</i>
<b>Interpreter</b>	<i>Implements a specialized language.</i>
<b>Iterator</b>	<i>Accesses the elements of an object sequentially without exposing its underlying representation.</i>
<b>Mediator</b>	<i>Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.</i>
<b>Memento</b>	<i>Provides the ability to restore an object to its previous state (undo).</i>
<b>Observer</b>	<i>Publish/subscribe pattern which allows a number of observer objects to see an event.</i>
<b>State</b>	<i>Allows an object to alter its behavior when its internal state changes.</i>
<b>Strategy</b>	<i>Allows one of a family of algorithms to be selected on-the-fly at runtime.</i>
<b>Template method</b>	<i>Defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.</i>
<b>Visitor</b>	<i>Separates an algorithm from an object structure by moving the hierarchy of methods into one object.</i>

FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

FONTE: Wikipedia contributors. Design Patterns. Wikipedia, The Free Encyclopedia. 2020.

# Padrões de arquitetura



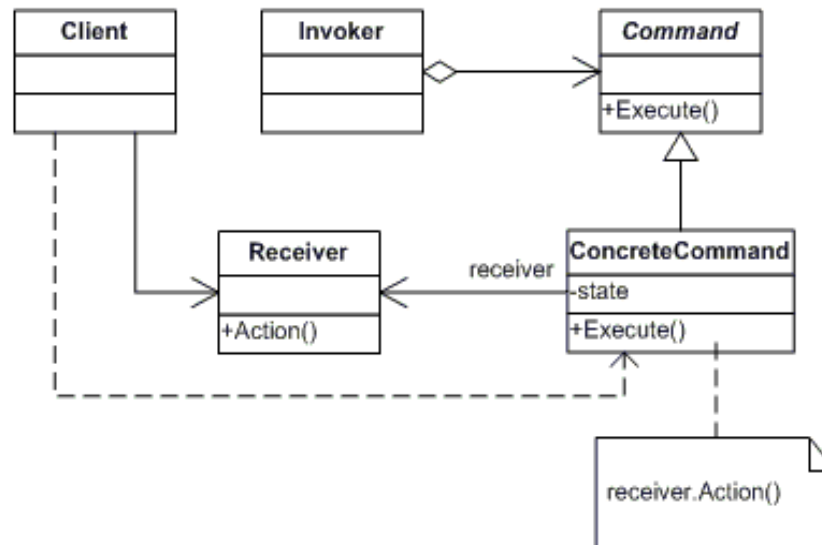
## ✓ Padrão *Command*:

- Encapsula solicitação como objeto.

## ✓ Aplicabilidade:

- Parametrização de objetos por ação a realizar.
- Prover suporte ao registro (*log*) de modificações.
- Estruturação de sistema em operações de alto nível.

# Padrões de arquitetura



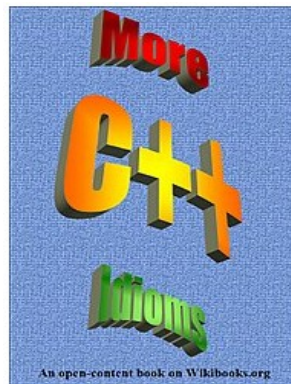
FONTE: Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

# Padrões de arquitetura

## ✓ Idioma:

- Particular a determinada linguagem de programação.
- Como implementar em determinada linguagem.
- Técnica específica a determinada linguagem.

## ✓ Exemplo de catálogo de idiomas:



### Table of Contents [\[ edit \]](#)

*Note: synonyms for each idiom are listed in parentheses.*

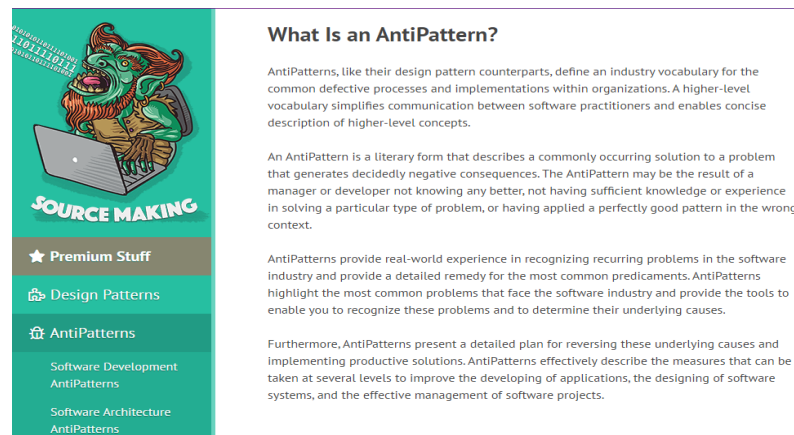
1. Address Of <#>
2. Algebraic Hierarchy <#>
3. Attach by Initialization <#>
4. Attorney-Client <#>
5. Barton-Nackman trick <#>
6. Base-from-Member <#>
7. Boost mutant <#>
8. Calling Virtuals During Initialization <#>
9. Capability Query <#>
10. Checked delete <#>
11. Clear-and-minimize <#>
12. Coercion by Member Template <#>
13. Computational Constructor <#>
14. Concrete Data Type <#>
15. Construct On First Use <#>
16. Construction Tracker <#>
17. Copy-and-swap <#>

# Padrões de arquitetura

## ✓ Anti-padrão:

- Lição aprendida.
- Pode informar solução inadequada.
- Pode informar como migrar para solução adequada.

## ✓ Exemplo de catálogo de anti-padrões:





# PERSISTÊNCIA DE OBJETOS

# Persistência de objetos



## ▼ Persistência:

- Objeto destruído só quando é necessária a sua destruição.
- Objeto em memória não volátil.
- Pode ser usado sistema de arquivos.
- Pode ser usado sistema gerenciador de banco de dados.
- Pode ser usado arcabouço (*framework*) para persistência.

## ▼ Exemplos de arcabouços para persistência:

- Hibernate.
- Torque.
- LiteSQL.



# Persistência de objetos



## ▼ Possíveis responsabilidades do SGBD:

- Controlar concorrência.
- Prover consistência em caso de erros.
- Facilitar acesso a dados.

## ▼ Motivações para uso de SGBD:

- Dados compartilhados por diferentes aplicações.
- Pesquisas complexas.
- Relatórios complexos.
- Aplicação transacional.
- Elevado número de objetos (instâncias).

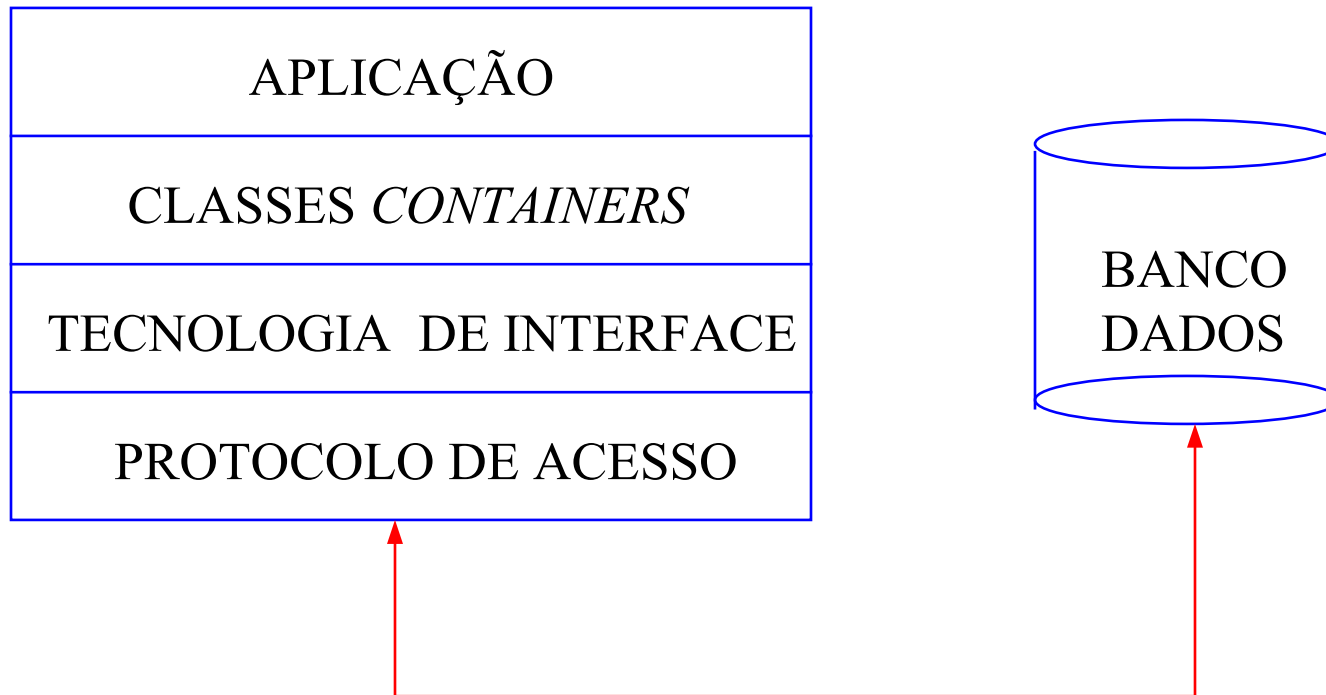
# Persistência de objetos



## ✓ Banco de dados orientado a objetos:

- Armazena objetos.
- Provê suporte a objetos complexos.
- Cada objeto tem identidade.
- Provê suporte ao encapsulamento.
- Provê suporte ao comportamento de objetos.
- Provê suporte à implementação de hierarquias de classes.
- Possibilita definição de novas classes.
- Pode simplificar o desenho (*design*) orientado a objetos.
- Falta de linguagem padronizada para acesso a dados.
- Acesso a dados via linguagem de programação.

# Persistência de objetos





# REFATORAÇÃO

# Refatoração



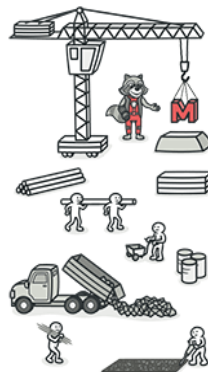
## ✓ Refatoração (*refactoring*):

- Transformações de projeto (*design*) existente.
- Reestrutura sistema sem mudar comportamento.
- Contínua reestruturação torna pouco útil documentar projeto detalhado.
- Remover duplicações e simplificar.
- Melhorar comunicação.
- Acrescentar flexibilidade.
- Eliminar funcionalidades não usadas.
- Manter código conciso e de fácil entendimento.
- Evitar desordem e complexidade desnecessária.
- Procurar facilitar manutenção futura.

# Refatoração



## Refactoring Techniques



### Composing Methods

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

§ [Extract Method](#)

§ [Inline Method](#)

§ [Extract Variable](#)

§ [Inline Temp](#)

§ [Replace Temp with Query](#)

§ [Split Temporary Variable](#)

§ [Remove Assignments to Parameters](#)

§ [Replace Method with Method Object](#)

§ [Substitute Algorithm](#)



# MODELAGEM

# Modelagem



## ✓ Modelo:


- Simplificação da realidade.
- Adere a perspectiva e a nível de abstração.

## ✓ Exemplos de potenciais benefícios:

- Facilitar entendimento de sistema sendo construído.
- Facilitar simplificação e reuso.
- Possibilitar documentação de decisões tomadas.
- Possibilitar foco em aspectos relevantes.
- Adiamento de detalhes.
- Facilitar comunicação e divisão de trabalho.



# Modelagem



**Model.** (1) *representation of a real-world process, device, or concept (ISO/IEC/IEEE 24765:2017 Systems and software engineering-Vocabulary)* (2) *representation of something that suppresses certain aspects of the modeled subject (IEEE 1320.2-1998 (R2004))* (3) *representation of a system of interest, from the perspective of a related set of concerns (ISO/IEC 19506:2012).*

**Modeling.** (1) *the activity of representing some elements of a process, device, or concept (Software Extension to the PMBOK(R) Guide Fifth Edition).*

# Modelagem



## ✓ Modelagem orientada a objetos:

- Comportamento e dados são mantidos integrados.
- Suporta classe, objeto e herança.
- Sugere como identificar classes.
- Sugere como identificar responsabilidades.
- Classes agrupadas em hierarquias.
- Sugere como identificar interações entre objetos.

# Modelagem



## ✓ Unified Modeling Language (UML):

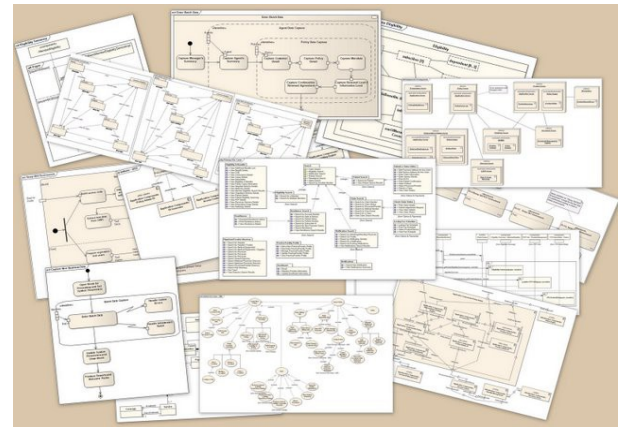
- Popular linguagem de modelagem.
- Desenvolvimento iniciado em 1994 e 1995 quando Booch, Rumbaugh e Jacobson passaram a trabalhar na Rational.
- Pode ser usada para construir modelos de análise, desenho (*design*) e implementação.



# Modelagem

## ✓ Exemplos de diagramas UML:

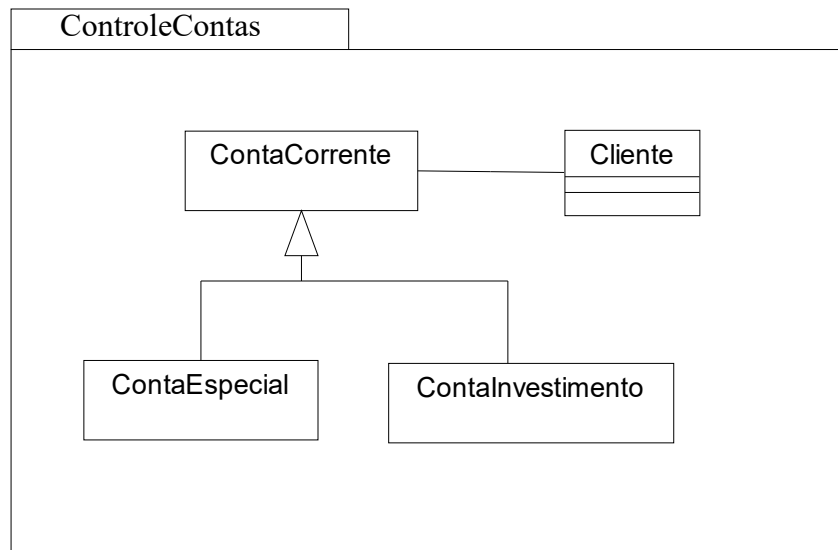
- Diagrama de classes.
- Diagrama de objetos.
- Diagrama de casos de uso.
- Diagrama de sequência.
- Diagrama de colaboração.
- Diagrama de atividades.
- Diagrama de componentes.
- Diagrama de implantação.



# Modelagem

## ✓ Pacote (*package*):

- Mecanismo para organizar elementos relacionados.

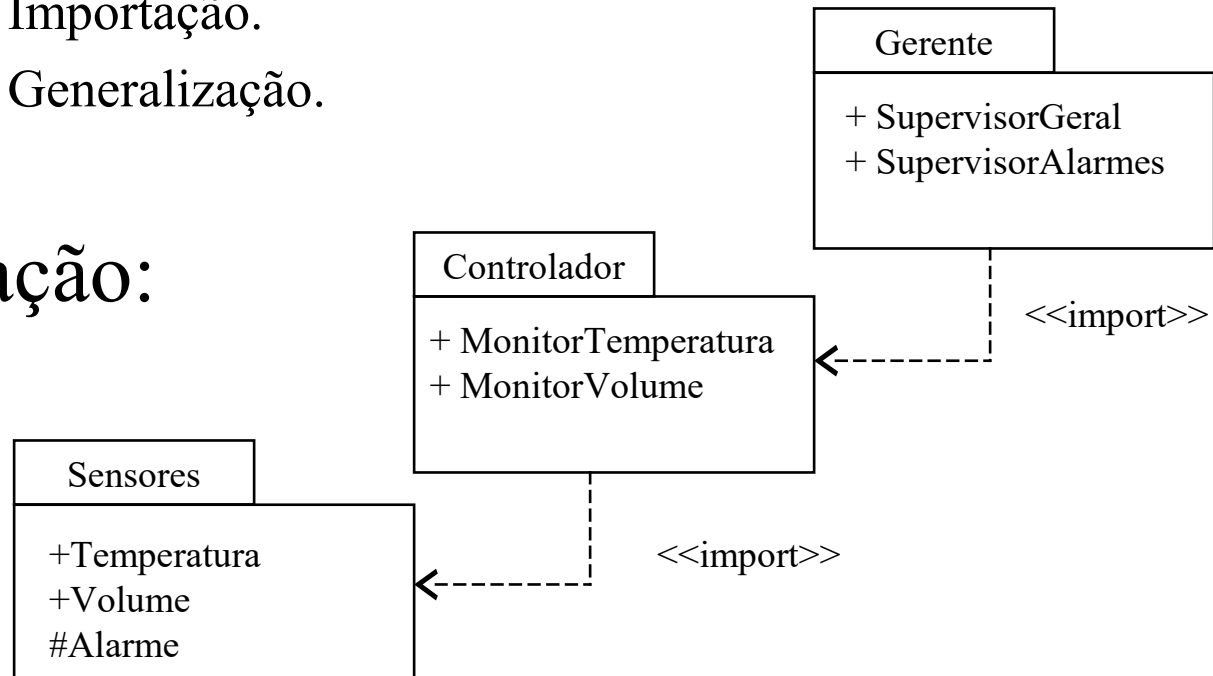


# Modelagem

## ▼ Relacionamento entre pacotes:

- Podem existir relacionamentos entre pacotes.
- Importação.
- Generalização.

## ▼ Importação:



# Modelagem

## ▼ Classe:

Nome
Atributos
Métodos

Pessoa
nome nascimento endereço telefone RG CPF

Pessoa
-nome:String -nascimento:Date -endereço:String -telefone:Integer -RG:Integer -CPF:Integer

# Modelagem

## v Nome de classe:

- Nome simples `Button`
- Nome de trajeto `java::awt::Button`
- Texto composto por letras, números e caracteres.

Motor

ContaCorrente

java:: awt:: Button



# Modelagem

## v Nome de atributo:

- Texto como no caso de nomes de classes.
- Pode ser especificado o tipo do atributo e valor inicial.

[visibilidade] nome[multiplicidade]  
[: tipo][ = valor\_inicial] [ {propriedades} ]

## v Exemplos de propriedades:

- changeable
- addOnly
- frozen



## ✓ Declaração de método:

- Texto como no caso de nomes de classes.
- Podem ser informados nomes, tipos, valores *default* de parâmetros e tipo de retorno.

[visibilidade] nome [(parametros)]  
[: tipo-retorno][ {propriedades}]

- Parâmetros tem a sintaxe seguinte.

[direção] nome: tipo [ = valor-default ]

# Modelagem



## ✓ Direção:

- **in**                      Parâmetro de entrada não modificável.
- **out**                     Parâmetro de saída modificável.
- **inout**                  Parâmetro de entrada modificável.

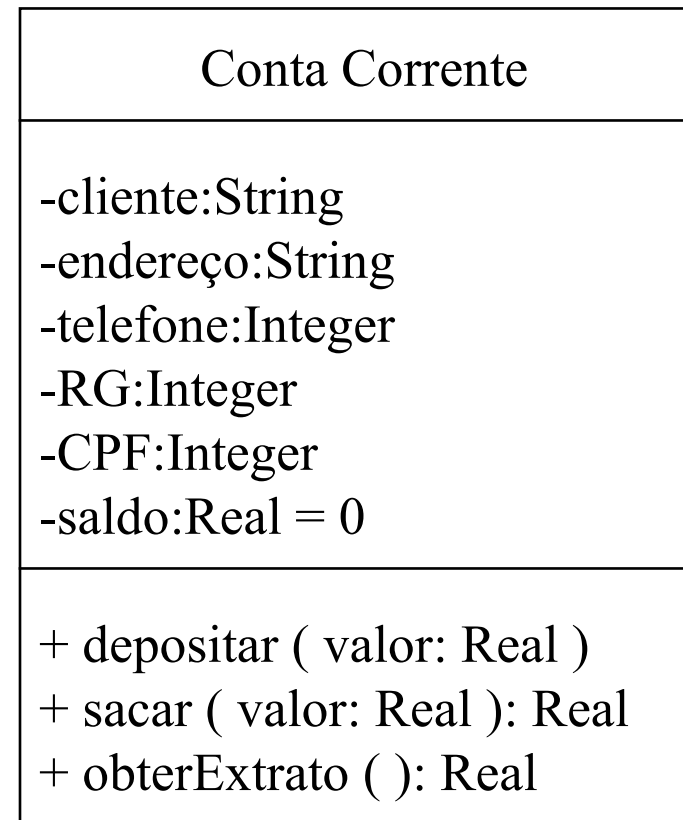
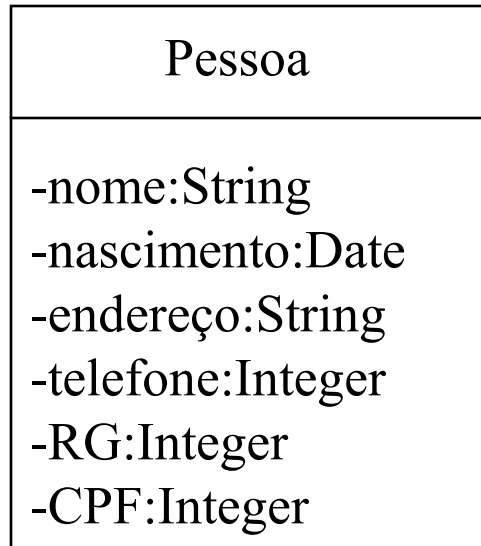
## ✓ Exemplos de propriedades:

- **isQuery**                Estado do objeto não é modificado.
- **sequential**            Execução tem que ser sequencial.
- **concurrent**            Possibilita execução concorrente.

# Modelagem

## ✓ Visibilidade:

- Público +
- Protegido #
- Privado -



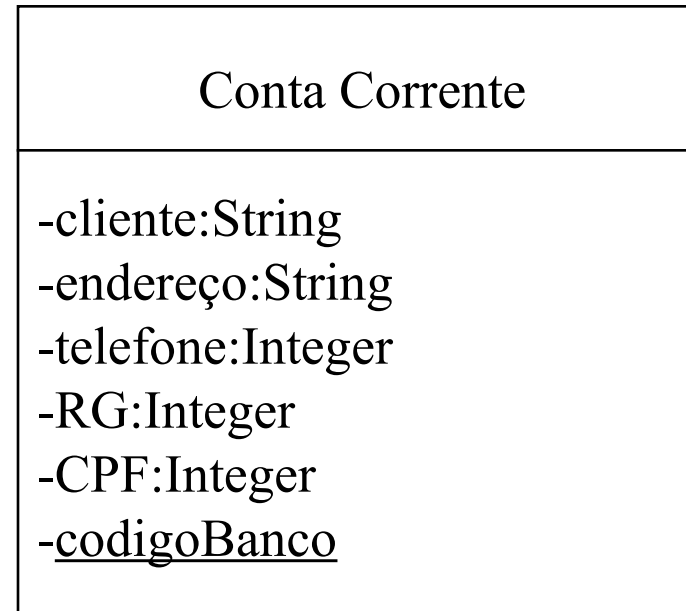
# Modelagem

## ✓ Escopo:

Atributos de instância



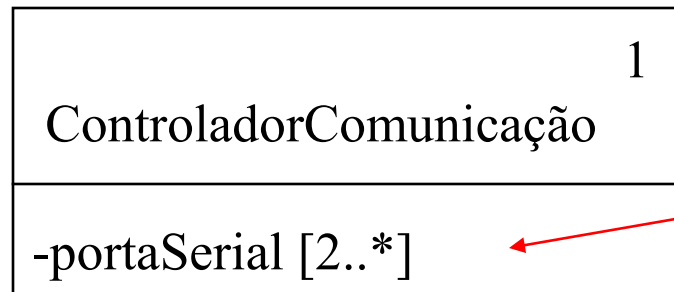
Atributo de classe



# Modelagem



## ✓ Multiplicidade de atributo:



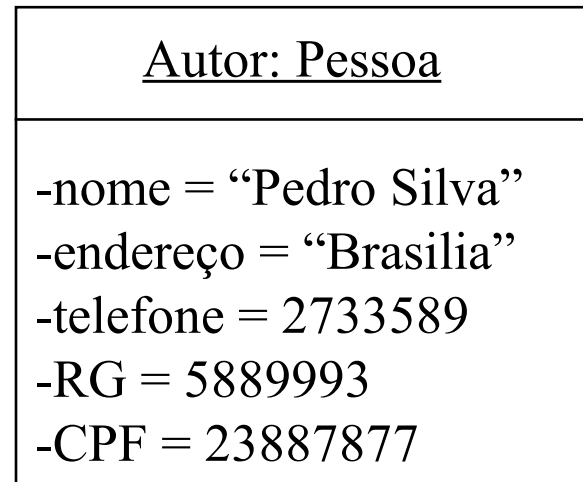
Multiplicidades.

# Modelagem

## v Objeto:

Nome de instância é  
sublinhado.

Nomes de objetos  
e classes são opcionais.

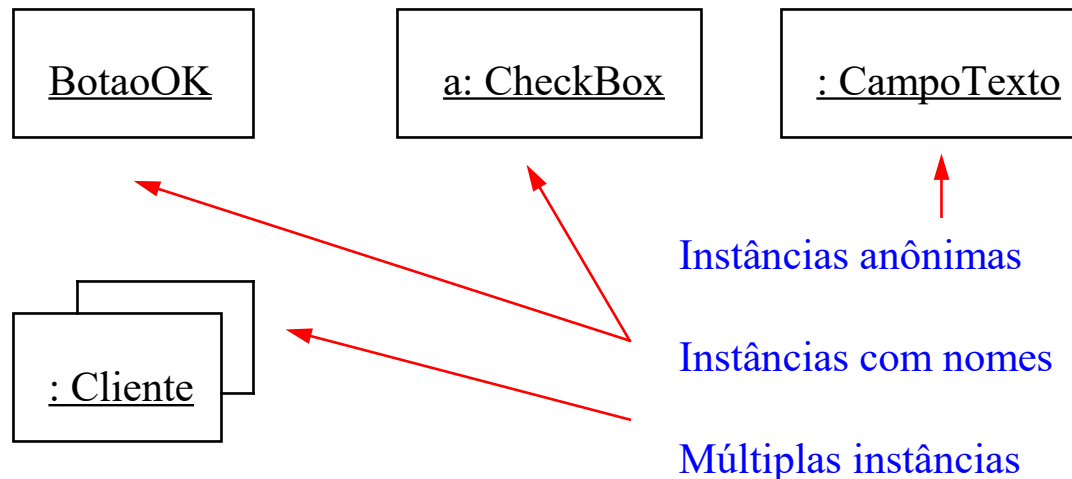


# Modelagem

## v Nome de instância:

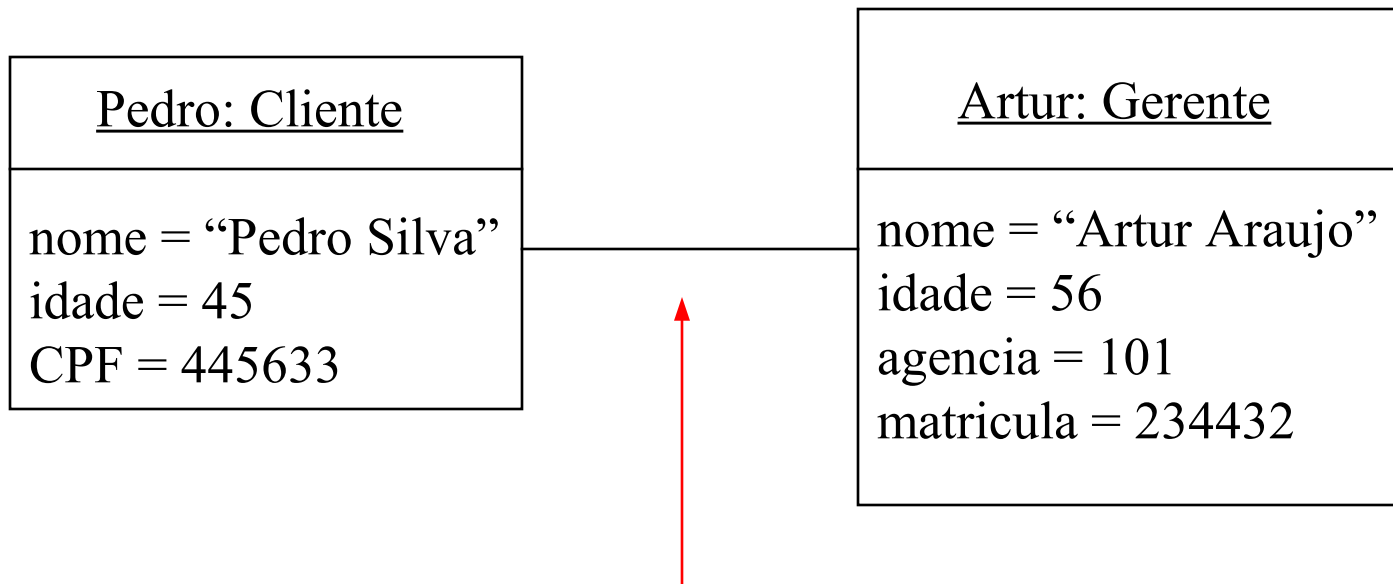
- Nomes simples `a:Botao`
- Nomes de trajeto `b:java::awt::Botao`

## v Nomes em diagramas:





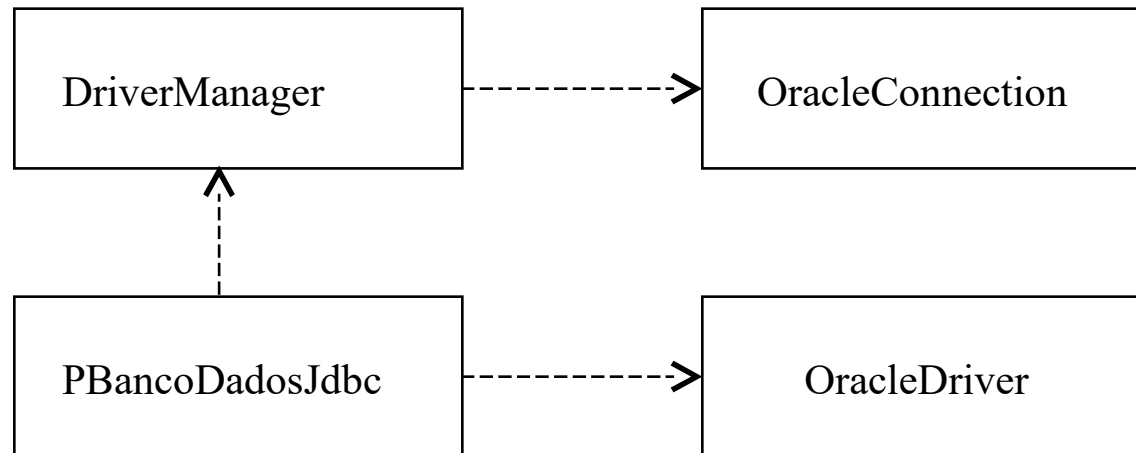
# Modelagem



Ligações (*links*) são instâncias de relacionamentos entre classes.

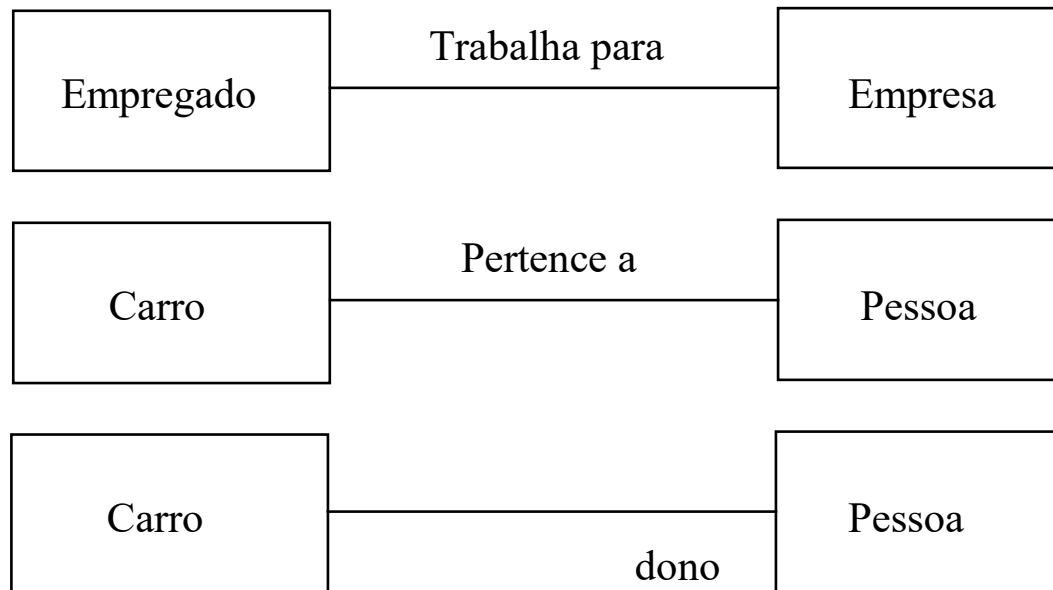
# Modelagem

## ▼ Dependência:



# Modelagem

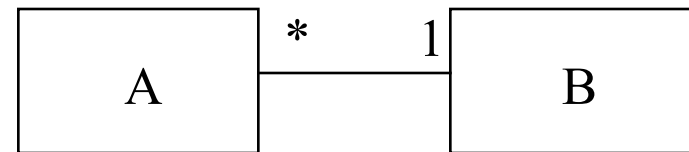
## ✓ Associação:



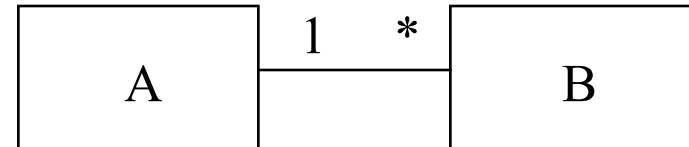
# Modelagem

## ✓ Multiplicidade em associação:

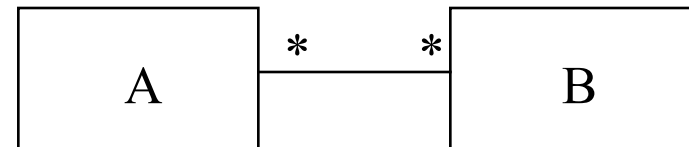
Muitos para um.



Um para muitos.

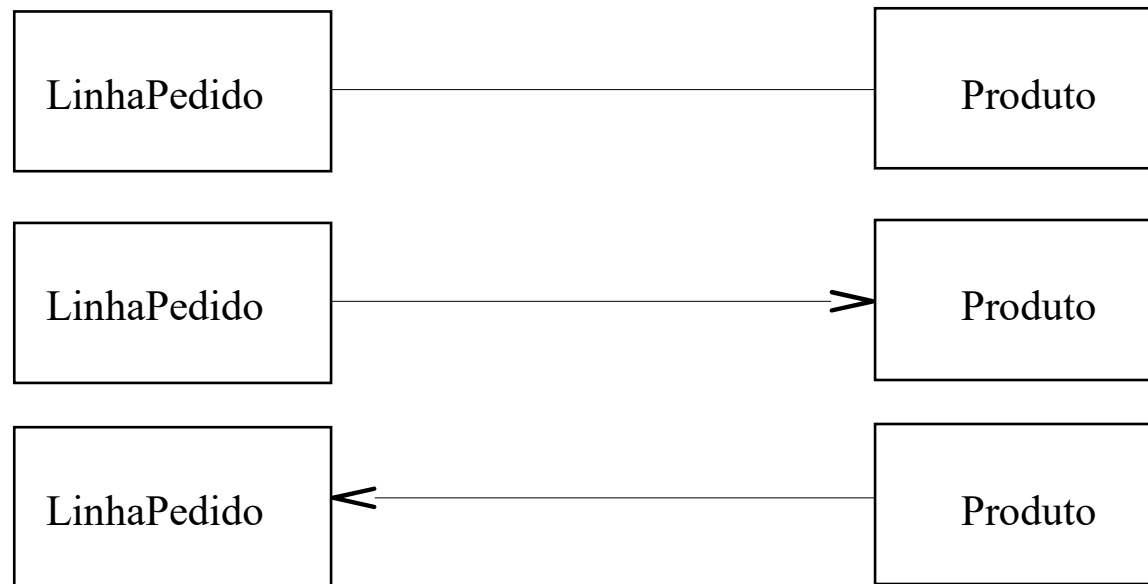


Muitos para muitos.



# Modelagem

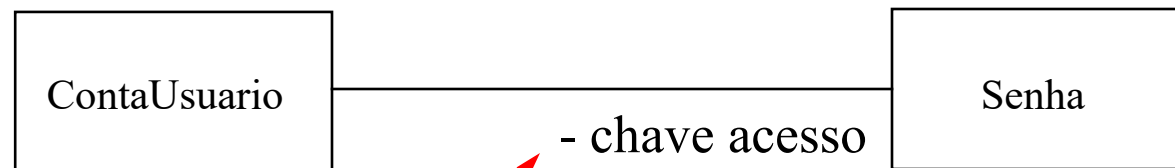
## ✓ Navegabilidade em associação:



# Modelagem

## ✓ Visibilidade em associação:

- Pode-se limitar a visibilidade relativa a objetos que não façam parte da associação.
- Visibilidade relativa pode ser pública, privada ou protegida.

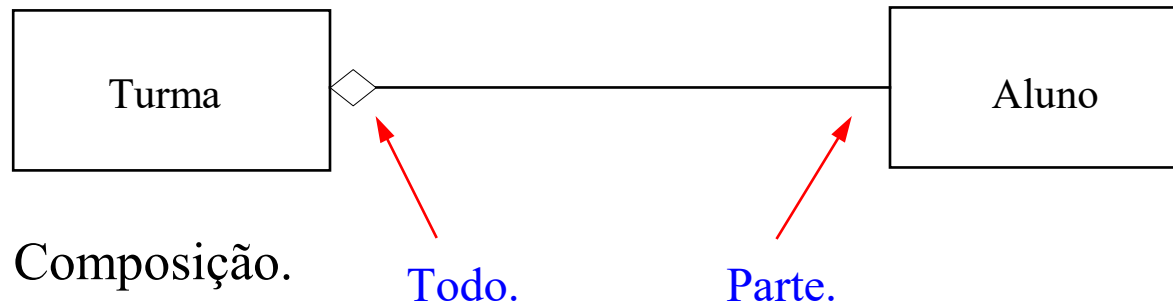


Informa visibilidade.

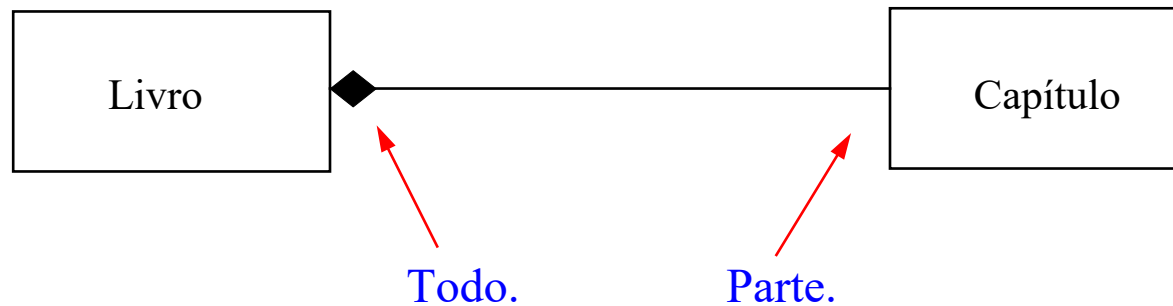
# Modelagem

## ✓ Associação em relação todo/parte:

- Agregação.

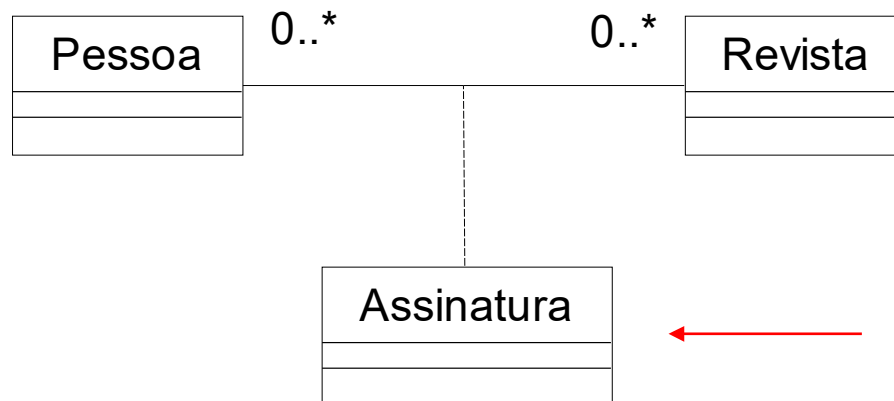


- Composição.



# Modelagem

## ✓ Classe de associação:

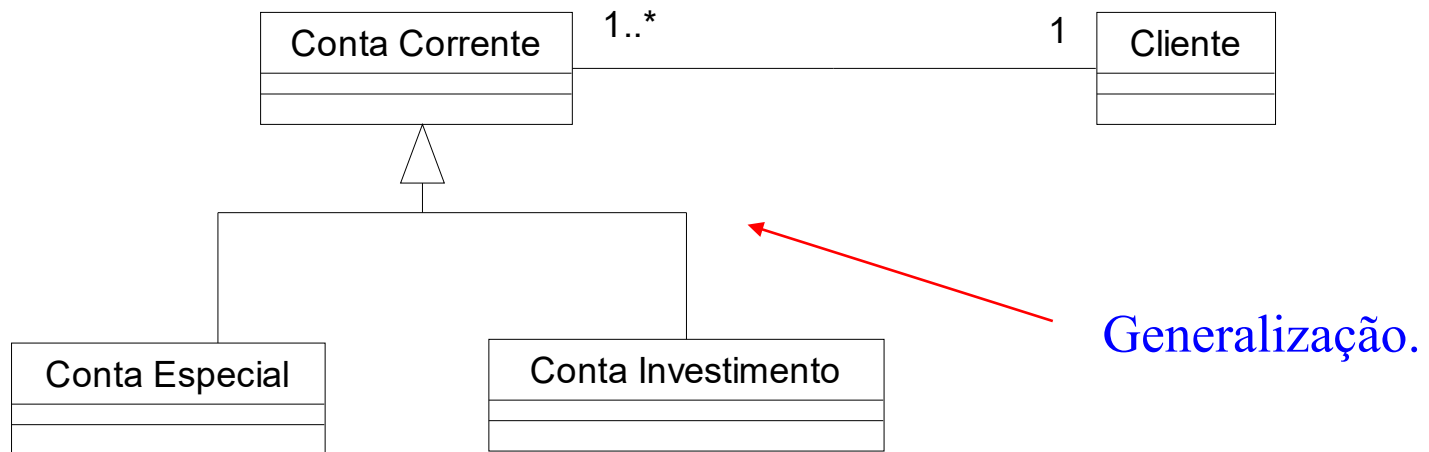


← Classe de associação.



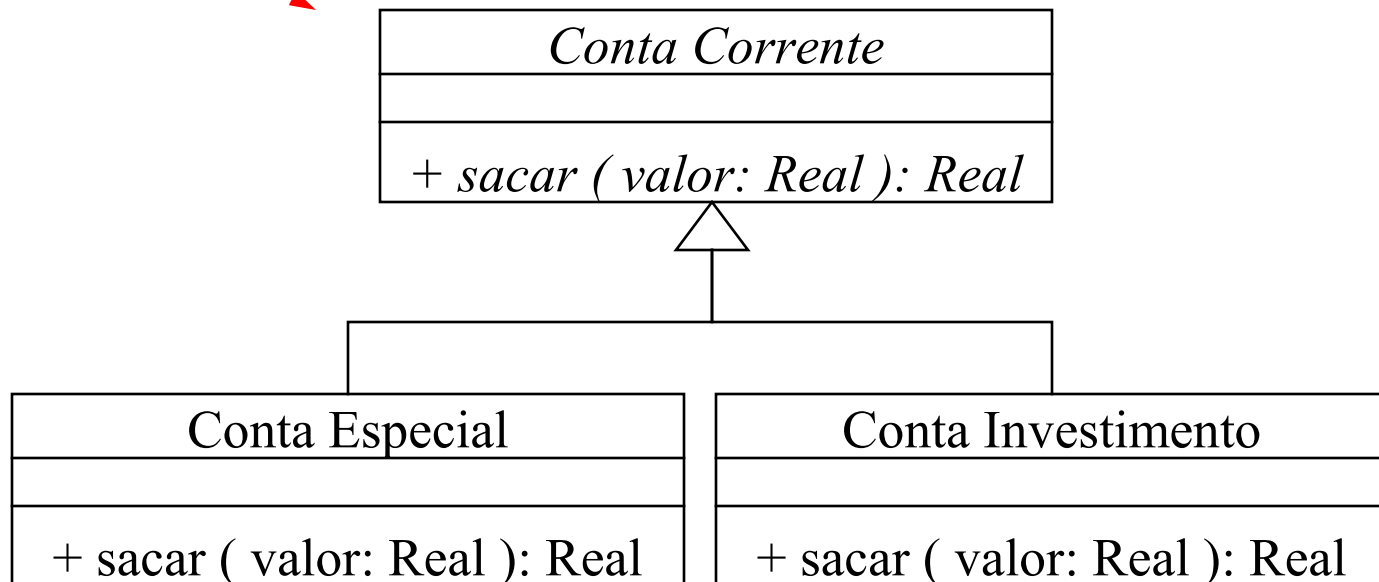
# Modelagem

## ✓ Herança:

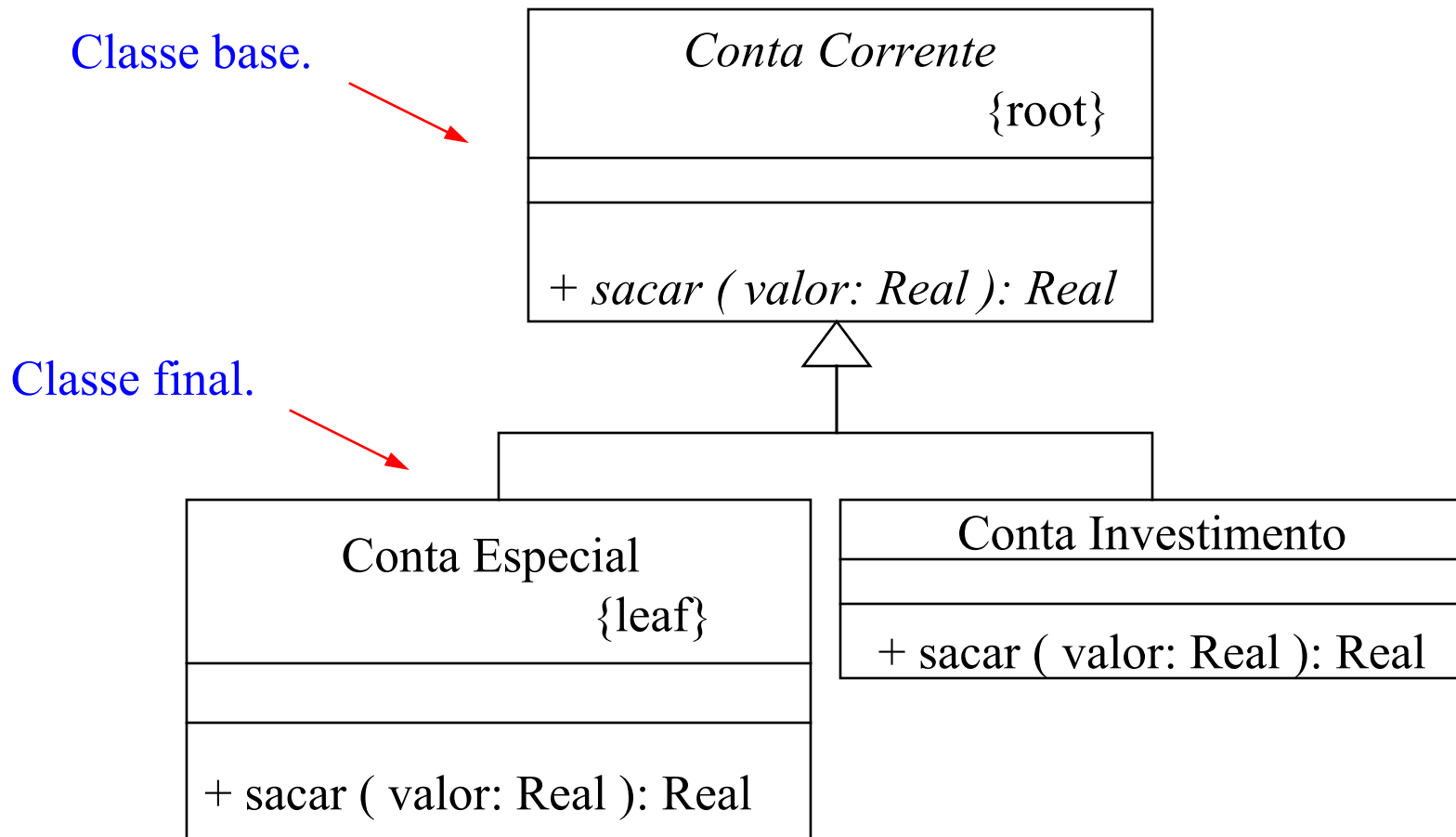


# Modelagem

Classe e método  
abstratos.



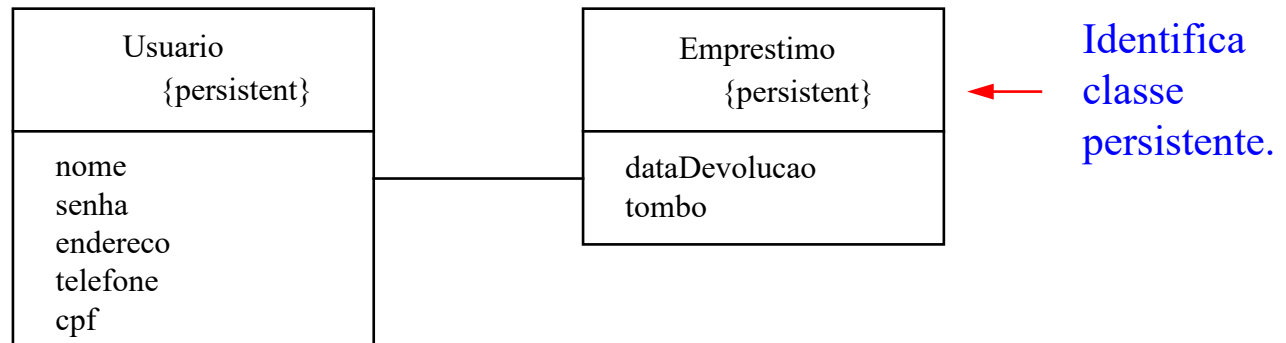
# Modelagem



# Modelagem

## ✓ Diagrama de classes:

- Representação gráfica de classes e seus relacionamentos.
- Pode ser desenhado segundo diversas perspectivas.
- Pode ser desenhado segundo diversos níveis de abstração.

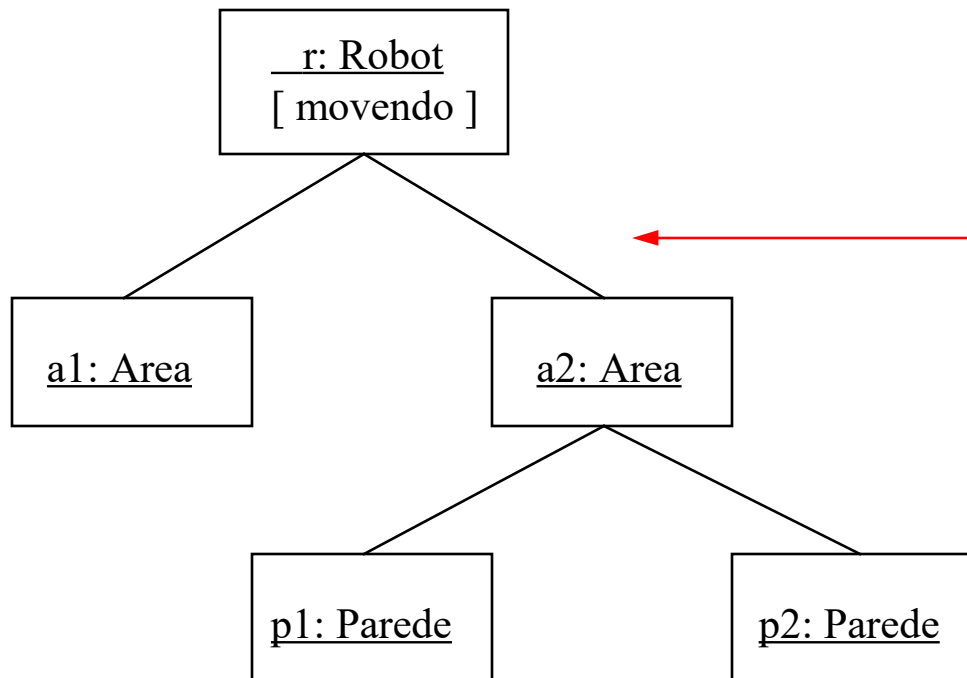




## ✓ Diagrama de objetos:

- Tipicamente representa objetos, ligações (*links*) e valores.
- Tipicamente representa objetos que colaboram.
- Pode ser instância de diagrama de classes.
- Pode ser parte estática de diagrama de interação.
- Pode facilitar entendimento de diagrama de classes.

# Modelagem

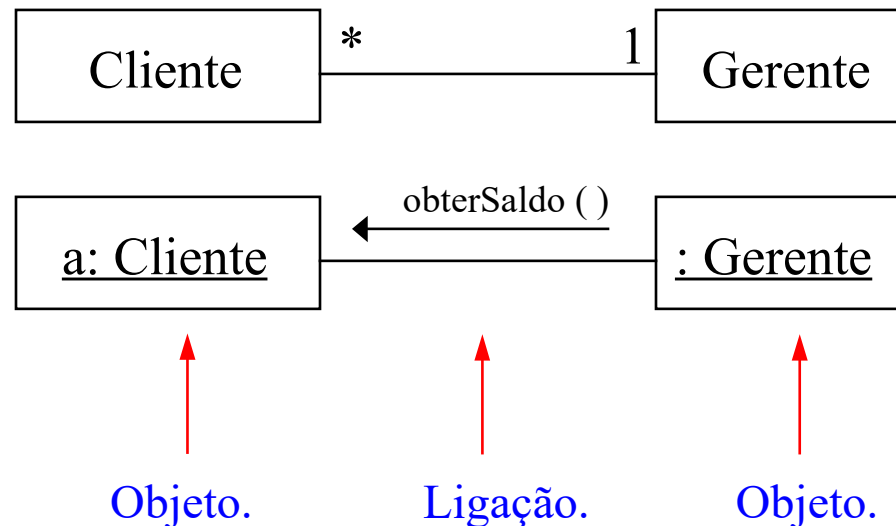


Ligações (*links*)  
entre objetos  
em determinado  
instante.

# Modelagem

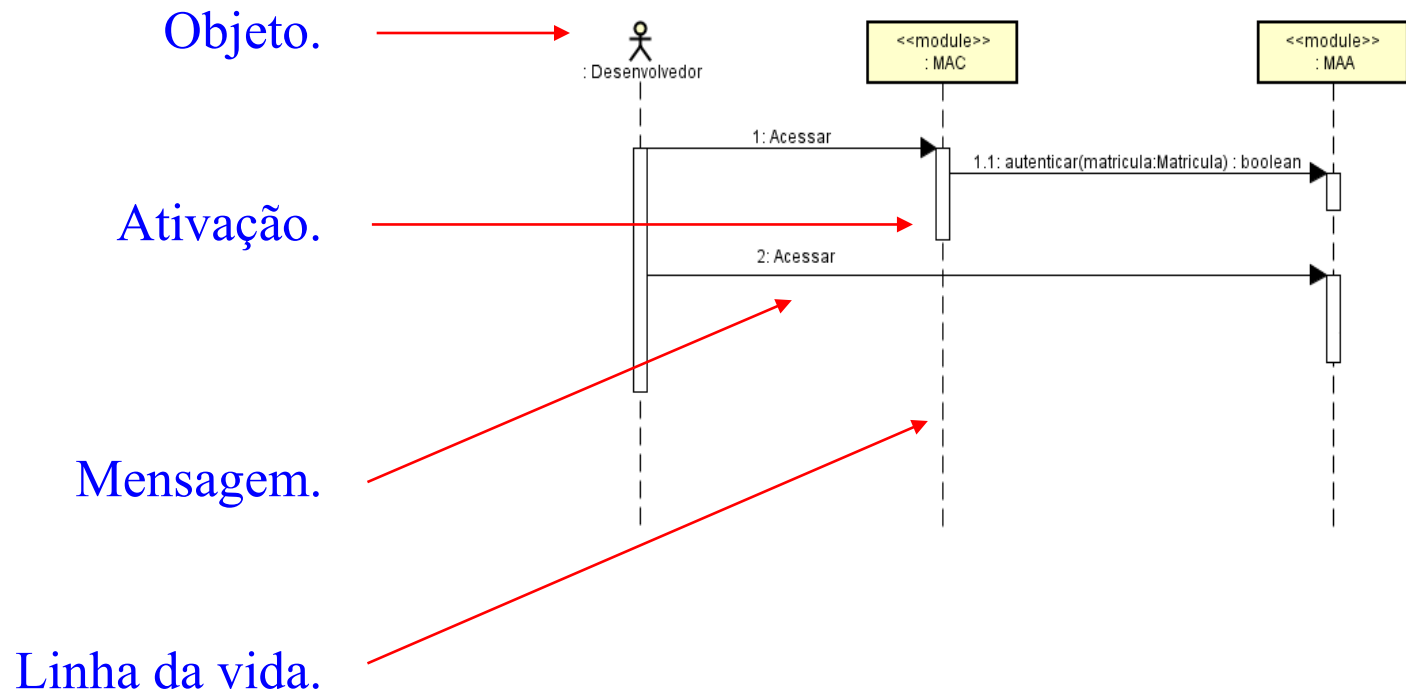
## ✓ Ligação (*link*):

- Instância de associação.
- Caminho através do qual mensagem pode ser enviada.
- Multiplicidade não é aplicável.



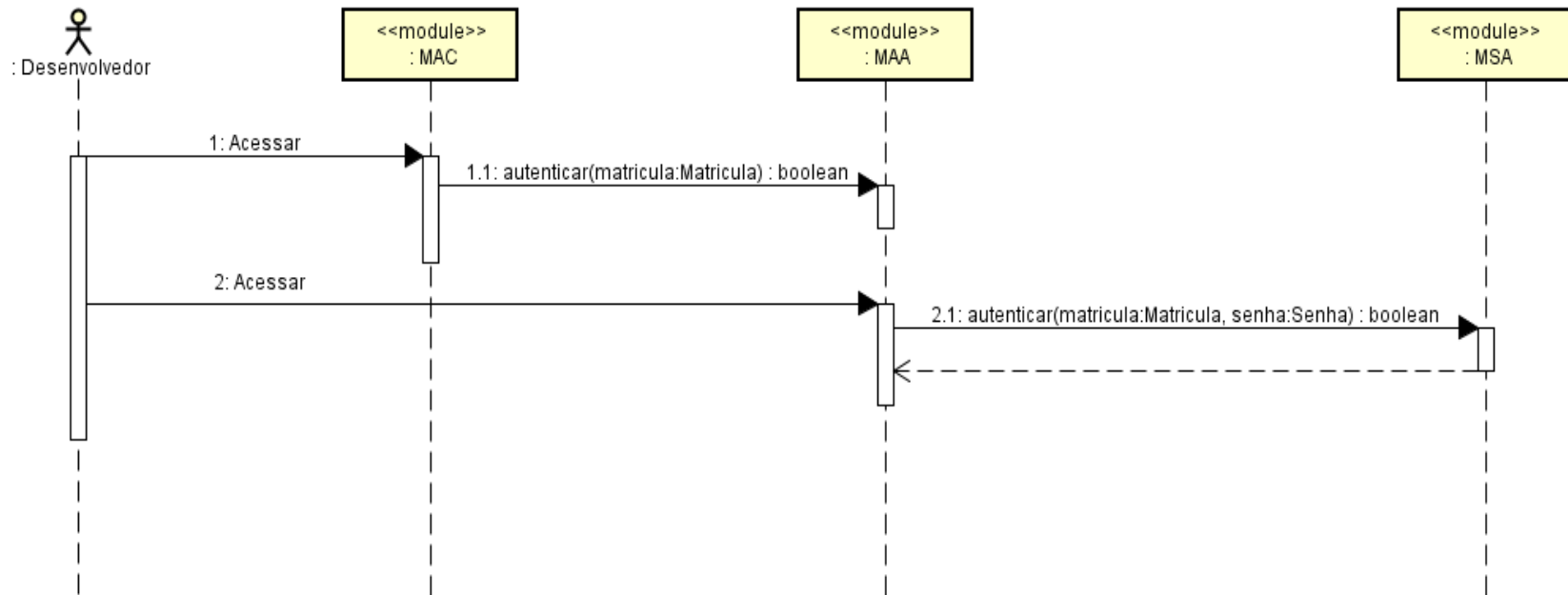
# Modelagem

## ✓ Diagrama de sequência:



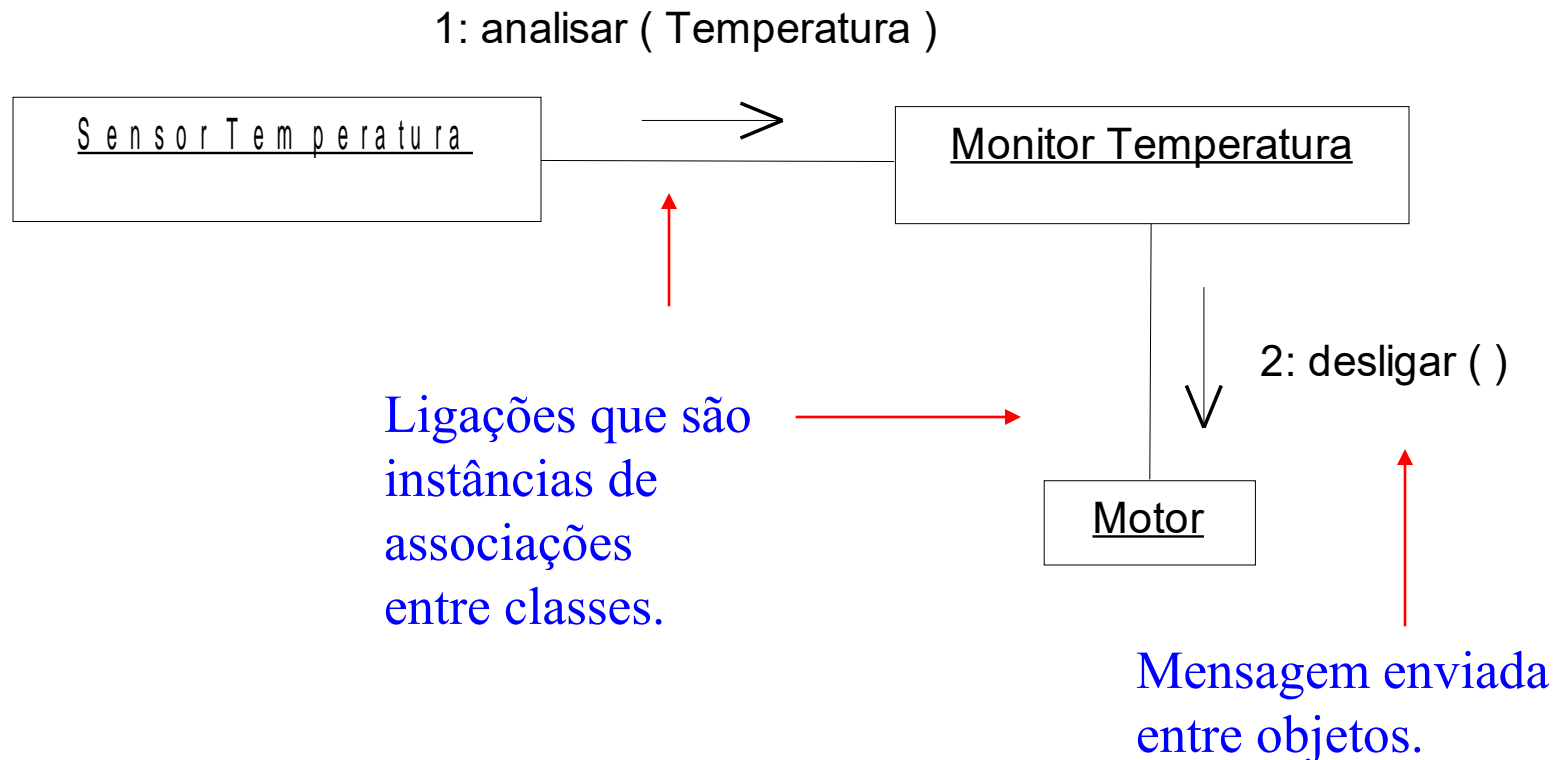


# Modelagem



# Modelagem

## ✓ Diagrama de colaboração:





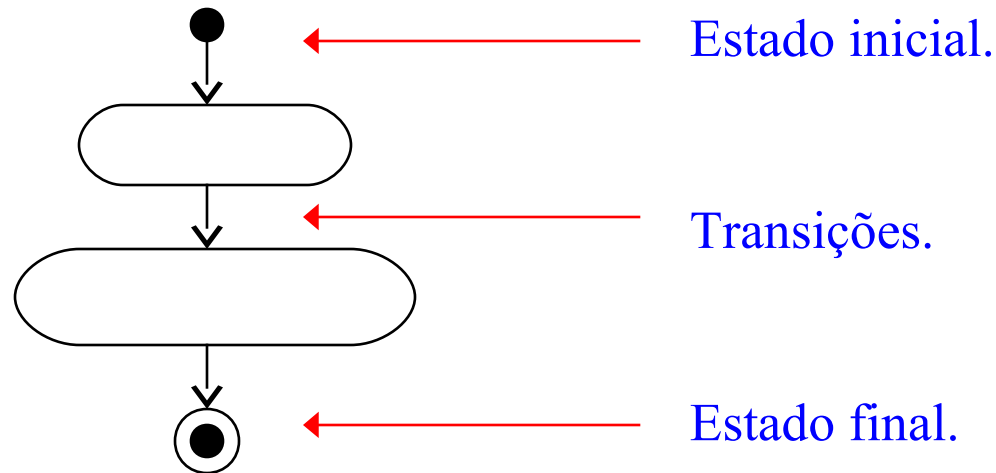
## ✓ Diagrama de atividades:

- Descreve fluxo de controle entre atividades.
- Representa passos em processo computacional.
- Representa fluxo de controle entre atividades.
- Representa atividades realizadas ao longo do tempo.
- Classe de diagrama de estados.
- Estados podem representar atividades ou ações.
- Atividades são não atômicas.
- Ações são atômicas.

# Modelagem

## ▼ Transição:

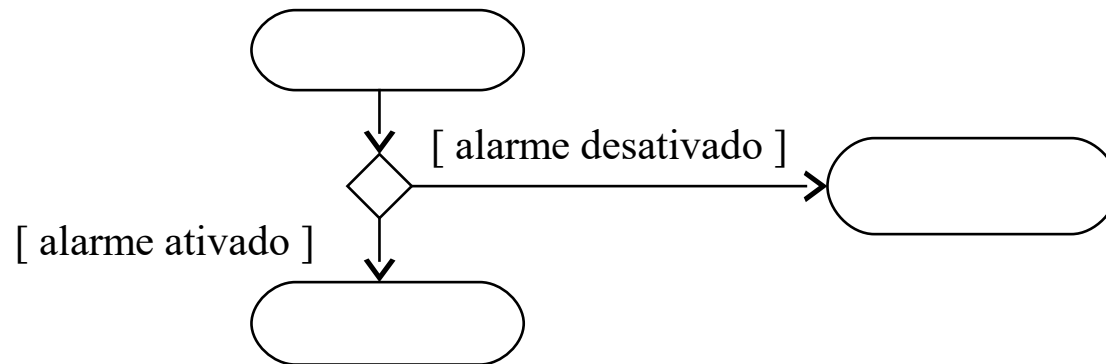
- Ocorre quando ação ou atividade é concluída.
- Fluxo de controle continua até ser encontrado estado final.



# Modelagem

## ✓ Salto:

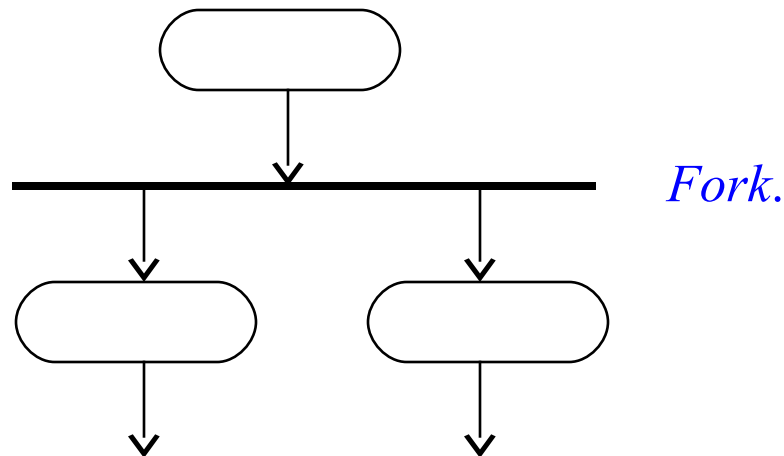
- Trajeto alternativo.
- Controle de fluxo baseado em teste.
- Transição associada a teste.



# Modelagem

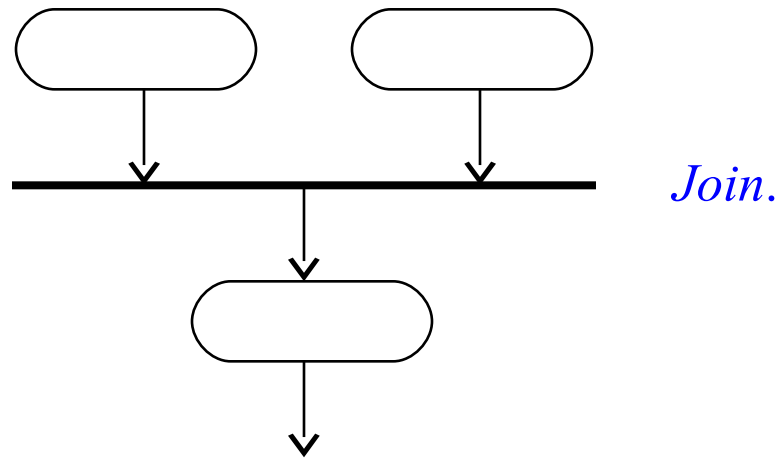
## ✓ *Fork e join:*

- Fluxos de atividades podem ser concorrentes.
- Barras de sincronização são usadas.
- *Fork* usado para dividir em fluxos concorrentes.



# Modelagem

- *Join* usado para sincronizar fluxos concorrentes.



- Atividades em fluxos concorrentes podem se comunicar através de sinais.



## ✓ Evento:

- Especificação de ocorrência significativa.
- Pode ser síncrono ou assíncrono.
- Pode ser interno ou externo.
- Eventos similares devem ser organizados em hierarquias.

## ✓ Exemplos de eventos:

- Sinal.
- Chamada a operação.
- Passagem de tempo.
- Mudança de estado.



# Modelagem

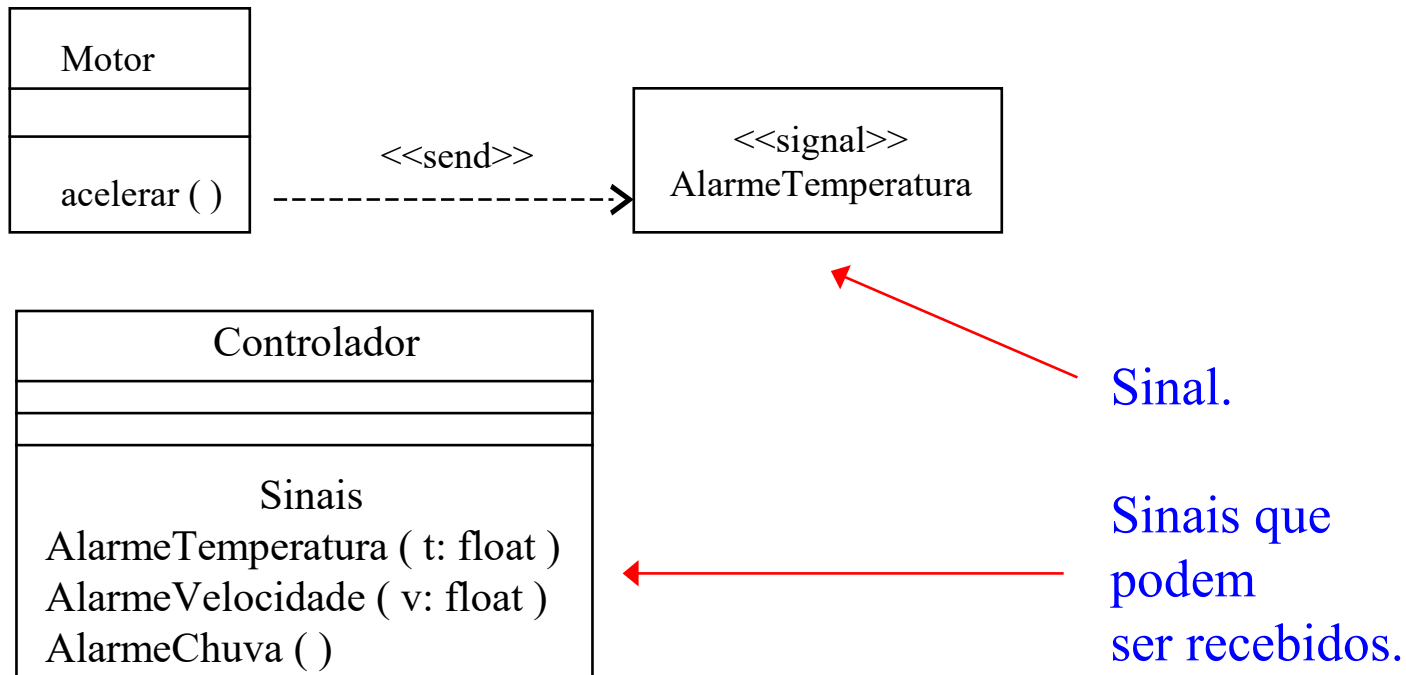


## ✓ Sinal:

- Enviado por um objeto e recebido por outro.
- Pode ser organizado em hierarquias.
- Pode ter atributos e operações.
- Sinais não devem ser usados em substituição ao fluxo normal de controle.
- Sinais não devem ser usados em substituição ao fluxo normal de controle.

# Modelagem

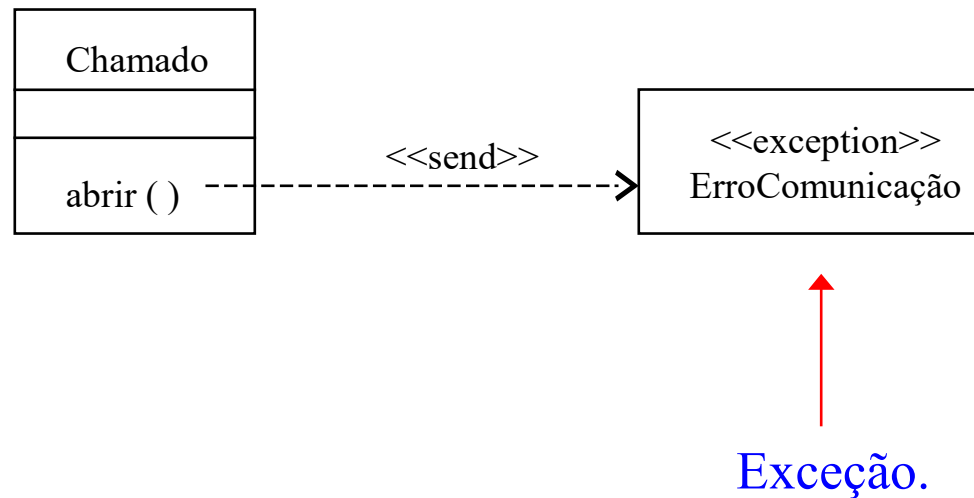
## ✓ Representação:



# Modelagem

## ✓ Exceção:

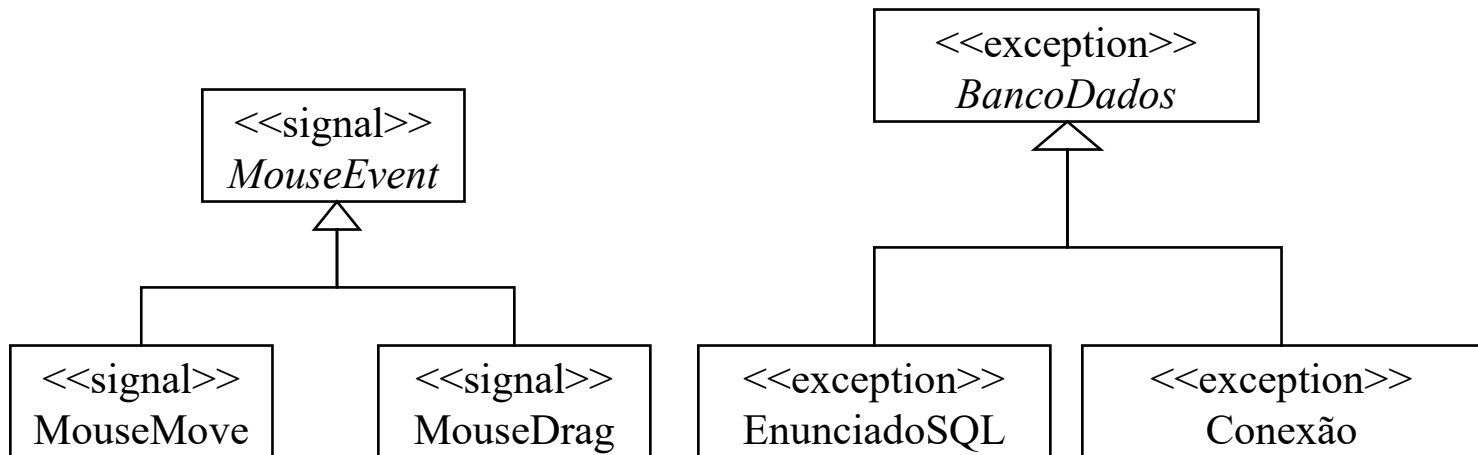
- Classe de sinal.
- Sinal enviado quando de operações.



# Modelagem

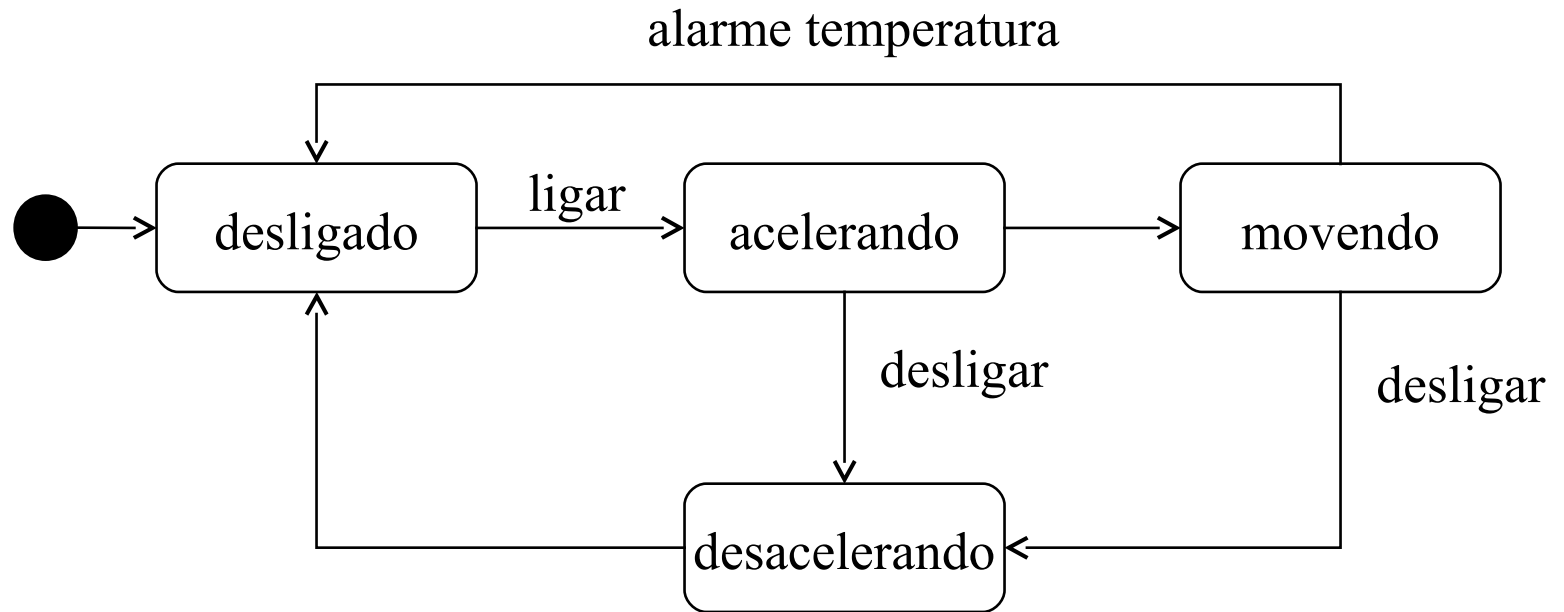
## ✓ Família de sinais:

- Identificadas as propriedades em comum.
- Sinais organizados em hierarquias.



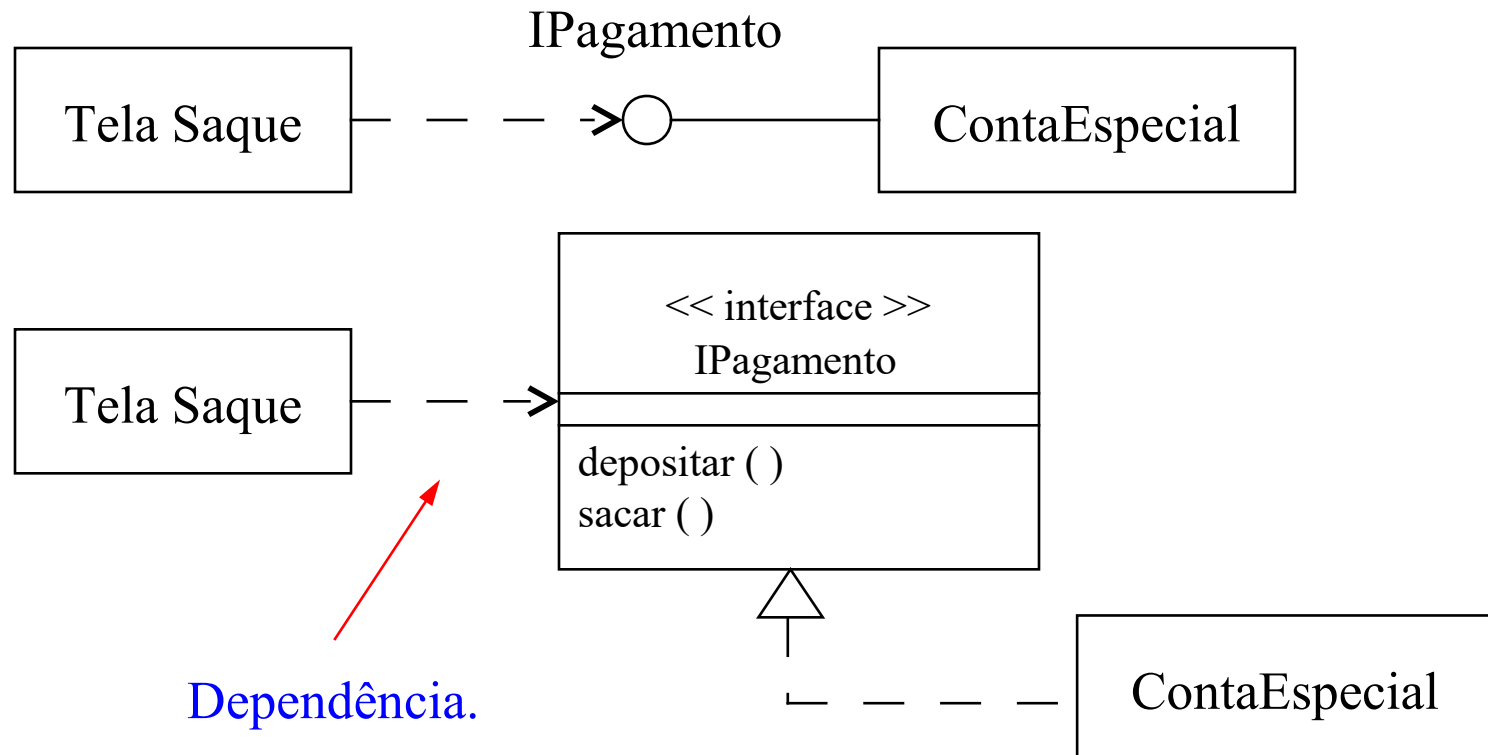
# Modelagem

## ▼ Máquina de estados:



# Modelagem

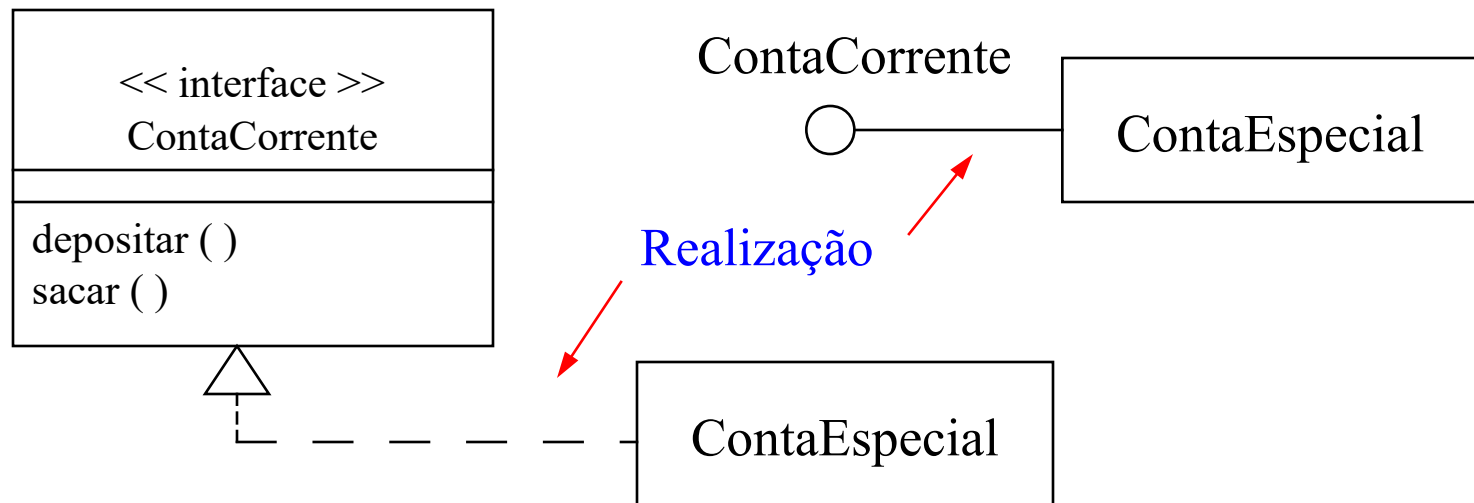
## ▼ Dependência:



# Modelagem

## ✓ Realização:

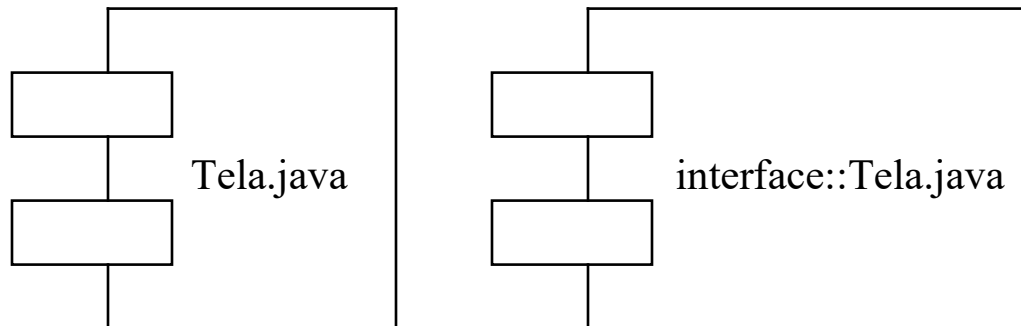
- Relacionamento entre interface e classe que a provê.
- Uma interface pode ser realizada por diferentes classes.
- Uma classe pode realizar diferentes interfaces.



# Modelagem

## ▼ Componente:

- Provê abstração de um elemento “físico” do software.
- Executável, biblioteca, tabela, arquivo, documento etc.

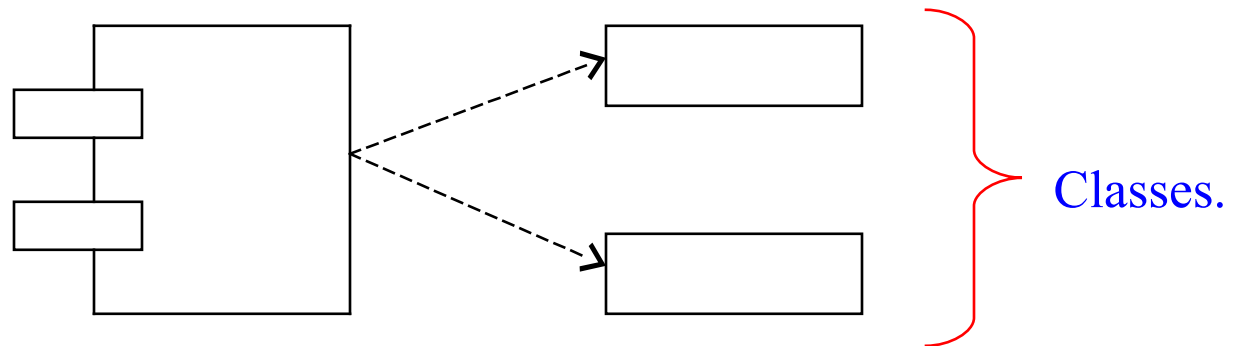




# Modelagem

## ✓ Classe e componente:

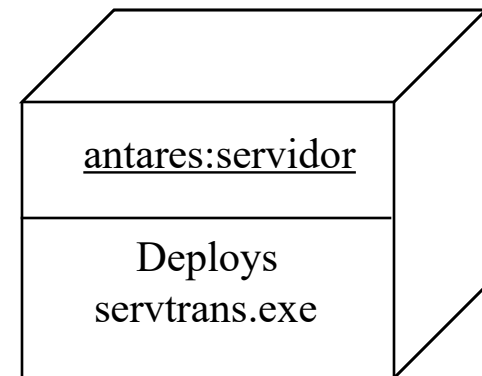
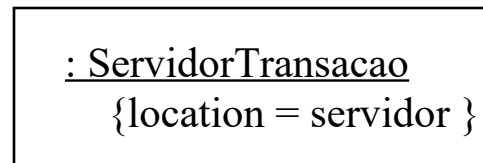
- Classe representa abstração lógica.
- Componente representa elemento “físico”.



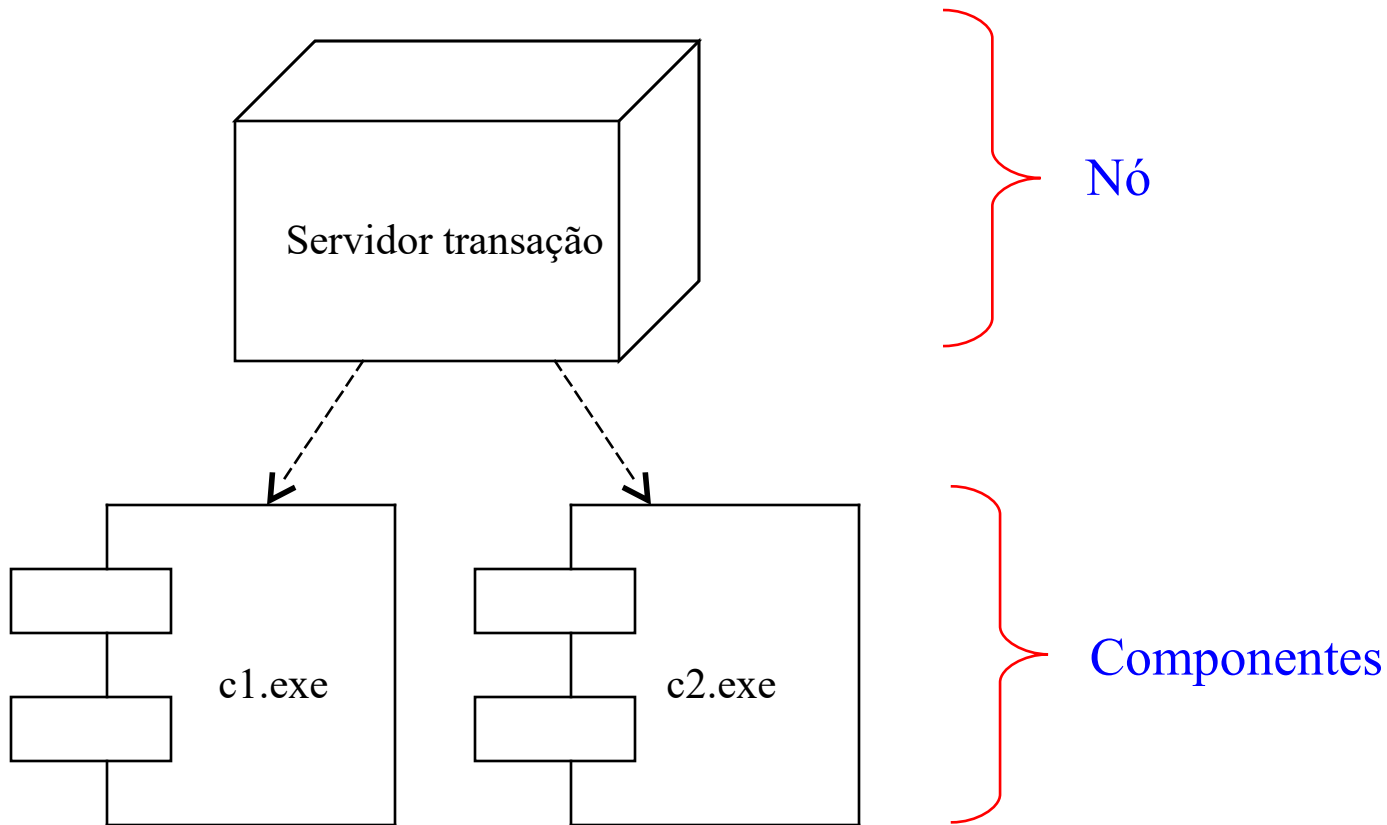
# Modelagem

## ✓ Nó (*node*):

- Recurso computacional.
- Elemento de infraestrutura.
- Componentes podem ser distribuídos entre nós.



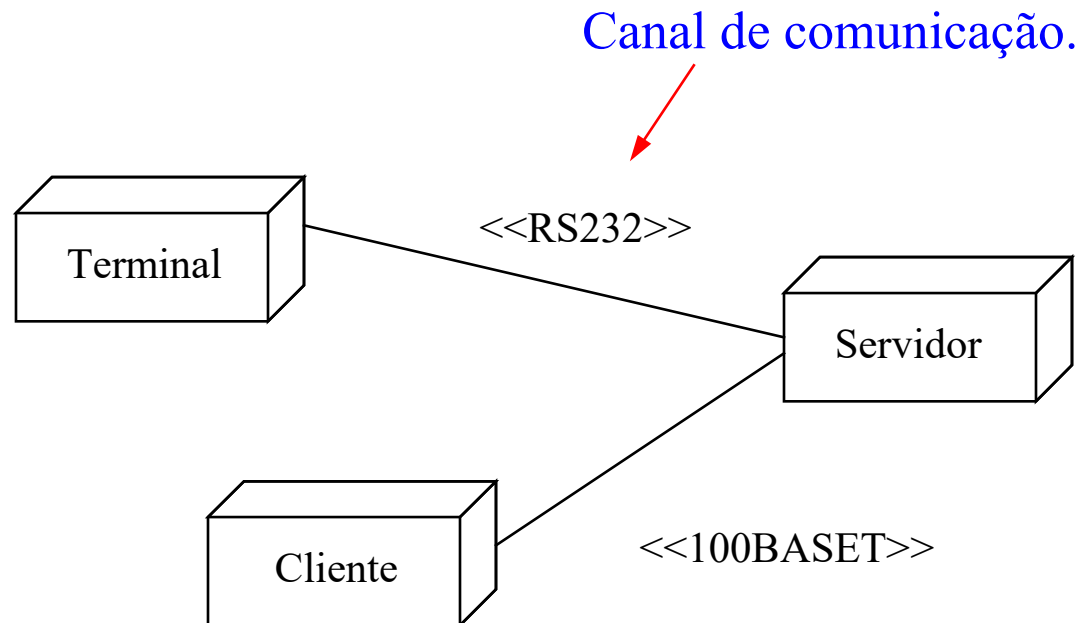
# Modelagem



# Modelagem

## ✓ Canal de comunicação:

- Interliga nós.





# DISTRIBUIÇÃO E CONCORRÊNCIA

# Distribuição e concorrência



## ✓ Sistema distribuído:

- Distribuição do processamento.
- Múltiplos processadores.
- Concorrência.
- Existem diversas classes de sistemas distribuídos.
- Sistema distribuído fortemente acoplado.
- Sistema distribuído fracamente acoplado.

# Distribuição e concorrência



## ✓ Aspectos de projeto (*design*):

- Arquitetura (estrutura) do software.
- Existem diversas arquiteturas (cliente/servidor etc.).
- Tempo de resposta percebido.
- Tempo gasto na criação de processos.
- Tempo de entrada/saída comparado com processamento.
- Capacidade do servidor tratar múltiplas solicitações.
- Nível de concorrência.

# Distribuição e concorrência



## ✓ Características de servidor:

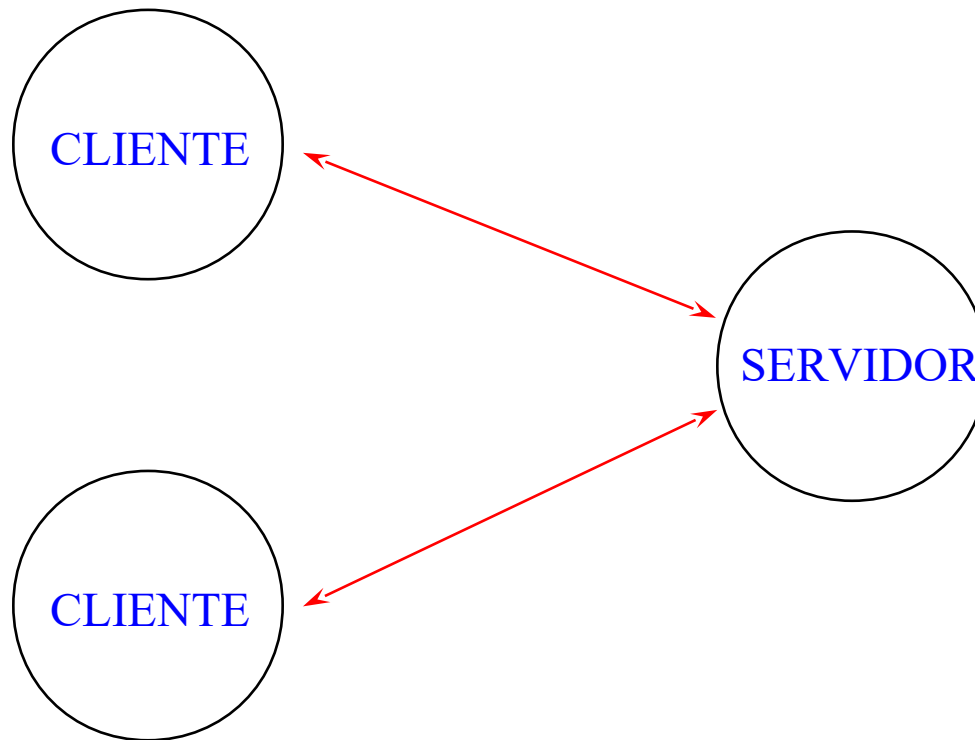
- Verifica identidade do cliente.
- Verifica se acesso é permitido.
- Pode exigir direitos especiais para ser executado.
- Pode ser mais difícil de implementar do que cliente.

## ✓ Características de cliente:

- Pode acessar serviços padronizados.
- Pode acessar serviços específicos.
- Pode permitir especificação de parâmetros.
- Pode ser mais simples de implementar do que servidor.



# Distribuição e concorrência



# Distribuição e concorrência



## ✓ Alternativas de comunicação:

- Comunicação orientada à conexão.
- Comunicação não orientada à conexão.

## ✓ Alternativas quanto ao estado do servidor:

- Servidor com estado (*statefull*).
- Servidor sem estado (*stateless*).

## ✓ Concorrência

- Pode ser real ou aparente.
- Pode ser implementada por meio de vários processos.

# Distribuição e concorrência



## ✓ Exemplos de classes de servidores:

- Servidor interativo.
- Servidor concorrente.
- Servidor não orientado à conexão.
- Servidor orientado à conexão.

# Distribuição e concorrência



## ✓ Servidor interativo:

- Processa uma requisição por vêz.
- Pode resultar em baixo desempenho.
- Clientes tem que aguardar pela prestação do serviço.

## ✓ Servidor concorrente:

- Lida com múltiplas requisições simultaneamente.
- Podem existir múltiplos processos.
- Implementação tipicamente mais complexa.

# Distribuição e concorrência



## ✓ Servidor não orientado à conexão:

- Protocolo de transporte não garante a entrega segura.
- Confiabilidade provida por código da aplicação.
- Pode implementar *timeouts*.
- Pode implementar retransmissões.
- Típico quando necessário *broadcast*.

## ✓ Servidor orientado à conexão:

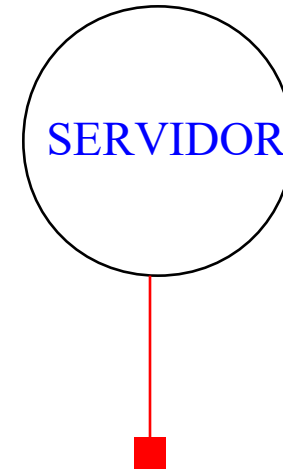
- Usa conexões.
- Gerencia conexões.

# Distribuição e concorrência



## ▼ Interativo e não orientado à conexão:

- Criar descritor.
- Ligar endereço do serviço.
- Receber requisição.
- Executar serviço.
- Devolver resposta.



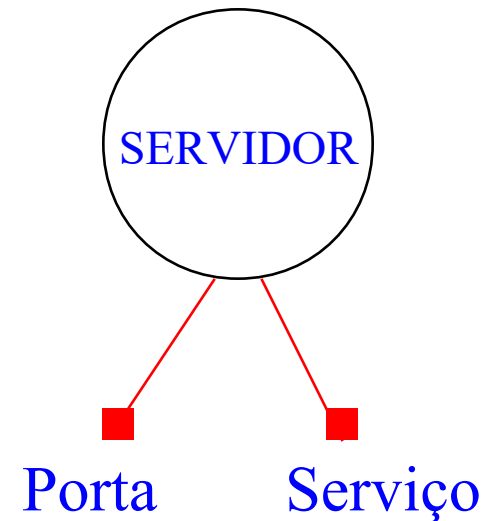
Porta de comunicação

# Distribuição e concorrência



## ▼ Interativo e orientado à conexão:

- Criar descritor.
- Ligar endereço do serviço.
- Aguardar solicitação de conexão.
- Aceitar conexão.
- Receber solicitação.
- Executar serviço.
- Enviar resposta ao cliente.
- Fechar conexão.

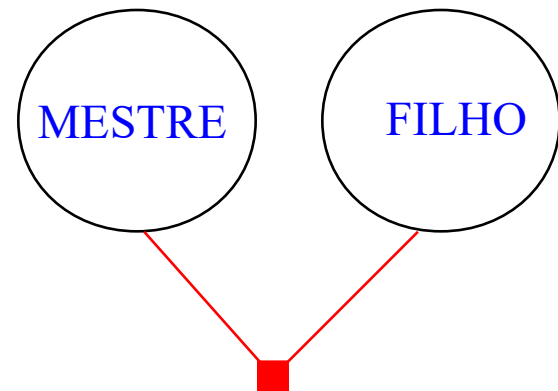


# Distribuição e concorrência



## ✓ Concorrente e não orientado à conexão:

- Criar descritor.
- Ligar endereço do serviço.
- Receber dado.
- Criar processo filho.
- Filho presta serviço.
- Filho envia resposta.
- Filho termina.



Solicitação / Serviço



# Distribuição e concorrência

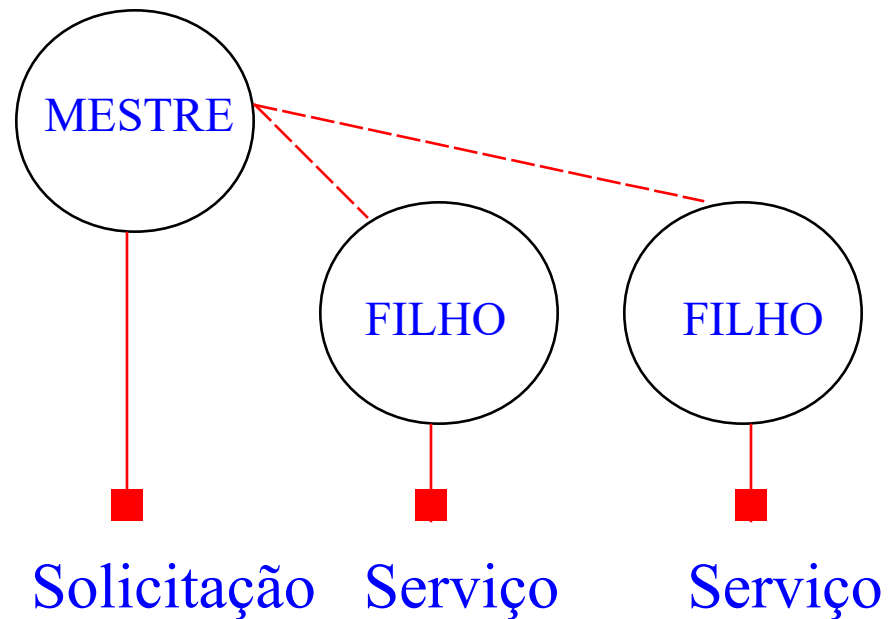


## ✓ Concorrente e orientado à conexão:

- Criar descritor.
- Ligar endereço do serviço.
- Aguardar solicitação de conexão.
- Alocar novo descritor.
- Aceitar pedido de conexão.
- Criar processo filho para prestar serviço.
- Mestre volta a aguardar solicitação de conexão.
- Filho recebe do mestre o descritor da conexão.

# Distribuição e concorrência

- Filho interage com o cliente.
- Filho envia resposta.
- Filho fecha conexão.
- Filho termina.



# Distribuição e concorrência



## ✓ Cliente não orientado à conexão:

- Identificar endereço do *host* e da porta do servidor.
- Alocar descritor.
- Permitir que pilha de protocolos escolha porta disponível.
- Comunicar com servidor.
- Liberar descritor.

# Distribuição e concorrência



## ✓ Cliente orientado à conexão:

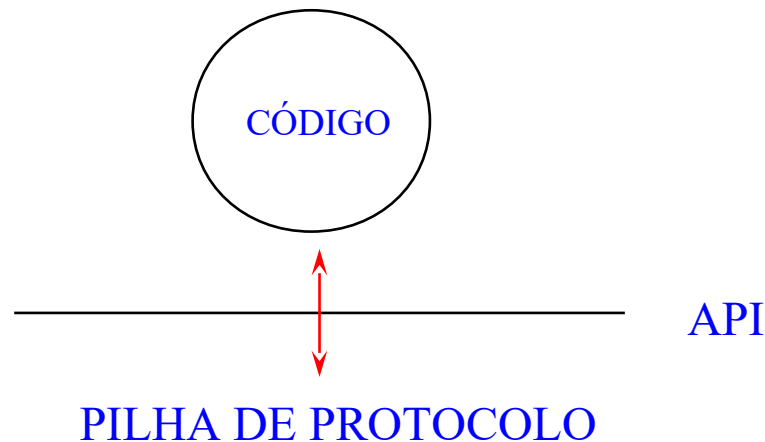
- Identificar endereço do *host* e da porta do servidor.
- Alocar descritor.
- Permitir que pilha de protocolos escolha porta disponível.
- Conectar ao servidor.
- Comunicar com servidor.
- Liberar conexão.

# Distribuição e concorrência



## ✓ Interface de programação:

- Código da aplicação interage com a pilha de protocolo por meio de interface de programação.



# Distribuição e concorrência



## ✓ Exemplos de funcionalidades providas:

- Alocar recursos para comunicação.
- Estabelecer conexão.
- Aguardar conexão.
- Especificar pontos de comunicação.
- Enviar dados.
- Receber dados.
- Terminar ou abortar conexão.
- Tratar erro.

# Distribuição e concorrência



## ✓ Interface de programação Sockets:

- Inicialmente provida no BSD Unix.
- Interface consistente.
- Cada socket tem tipo associado.
- Função *socket* cria ponto de comunicação.

```
int socket ( int family, int type, int protocol )
```

family      - identifica o domínio de comunicação

type        - tipo de socket

protocol    - seleciona protocolo a ser usado

# Distribuição e concorrência



## v Tipos de sockets:

- *Stream.*
- *Datagram.*

## v Associando endereço:

- Necessário associar endereço ao *socket*.
- Função *bind*.

`int bind ( int fd, struct sockaddr *addrp, int alen )`

<code>fd</code>	- descritor do socket
<code>addrp</code>	- ponteiro para o endereço do socket
<code>alen</code>	- tamanho do endereço em bytes



# Distribuição e concorrência



## ✓ Enviar dados:

- Interfaces usadas em qualquer tipo de *socket*.
- Existem três interfaces para envio de dados.

`int send ( int fd, char *buf, int len, int flags )`

`int sendto ( int fd, char *buf, int len, int flags, struct sockaddr *addrp, int alen )`

`int sendmsg ( int fd, struct msghdr *msgp, int flags )`

`fd` - descritor do socket

`buf` - endereço do buffer contendo os dados a transmitir

`len` - tamanho do buffer

`addrp` - endereço

`alen` - tamanho do endereço

`msgp` - endereço de estrutura com as características da mensagem

# Distribuição e concorrência



## ✓ Receber dados:

- Interfaces usadas em qualquer tipo de *socket*.
- Existem três interfaces para envio de dados.

`int recv ( int fd, char *buf, int len, int flags )`

`int recvfrom ( int fd, char *buf, int len, int flags, struct sockaddr *addrp, int alen )`

`int recvmsg ( int fd, struct msghdr *msgp, int flags )`

`fd` - descritor do socket

`buf` - endereço do buffer a ser usado para armazenar os dados

`len` - tamanho do buffer

`addrp` - endereço

`alen` - tamanho do endereço

`msgp` - endereço de estrutura com as características da mensagem

# Distribuição e concorrência



socket

bind

sendto

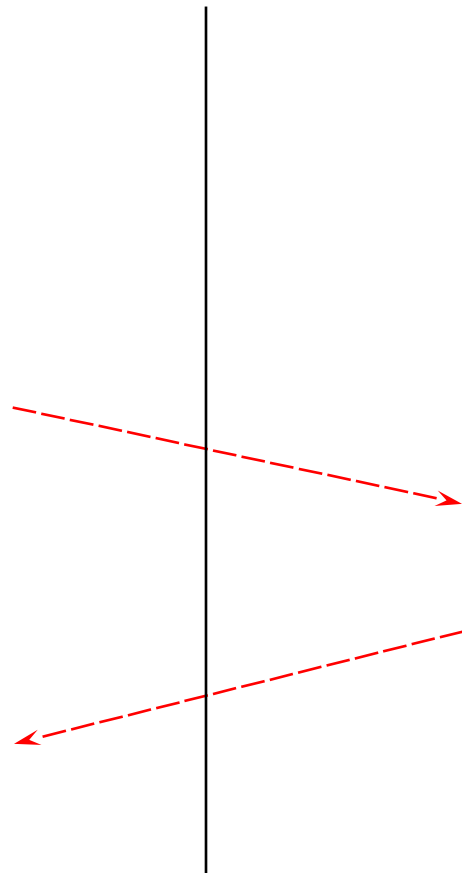
recvfrom

socket

bind

recvfrom

sendto



# Distribuição e concorrência



## ✓ Estabelecer conexão:

- Função *connect* para estabelecer conexão.
- Funções *listen* e *accept* para aceitar conexão.

```
int connect ( int fd, struct sockaddr *addr, int alen )  
int listen  ( int fd, int backlog )  
int accept (int fd, struct sockaddr *addr, socklen_t *addrlen)
```

fd	- descritor do socket
addr	- endereço
alen	- tamanho do endereço em bytes
backlog	- número de conexões que podem ficar pendentes
addrlen	- ponteiro para tamanho do endereço

# Distribuição e concorrência



## ✓ Processo cliente:

- Criar socket e liga endereço ao *socket*.
- Tentar iniciar conexão com o servidor.
- Enviar e receber dados.

## ✓ Processo servidor:

- Criar *socket* e liga endereço ao *socket*.
- Informar desejo de aceitar conexão.
- Aguardar indicação de conexão.
- Enviar e receber dados.

# Distribuição e concorrência



socket

bind

connect

send

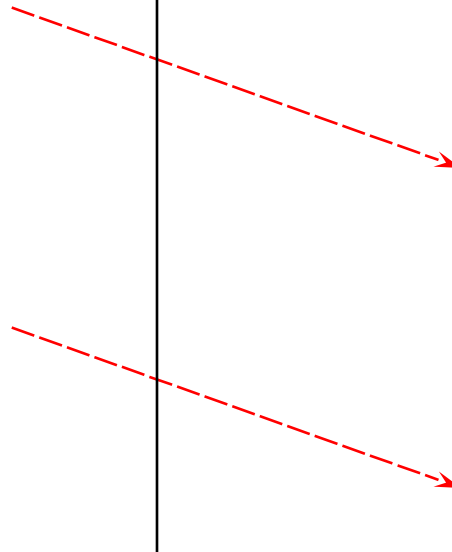
socket

bind

listen

accept

rcv



# Distribuição e concorrência



## ✓ *Remote Procedure Call:*

- Permite chamar procedimentos em computadores remotos.
- Chamada ao procedimento resulta no envio de mensagem.
- Procedimento é executado.
- Mensagem de resposta é enviada.
- Desconsiderados detalhes associados à comunicação.

# Distribuição e concorrência



## ▼ Passos:

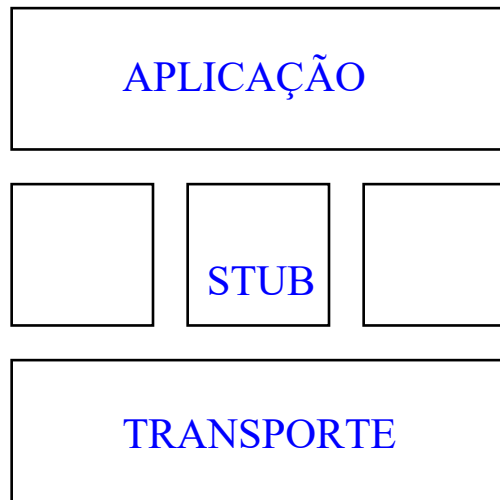
- Código cliente executa chamada ao procedimento.
- Procedimento *stub* é executado.
- Mensagem é enviada para o servidor.
- Servidor recebe mensagem.
- Procedimento a ser executado é identificado.
- Procedimento é chamado.
- Mensagem é enviada para o cliente.



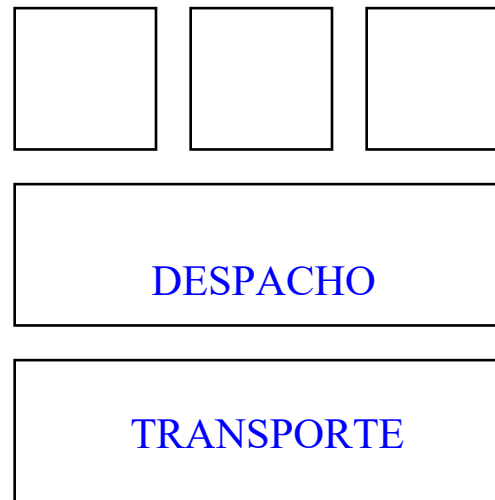
# Distribuição e concorrência



CLIENTE



SERVIDOR



PROCEDIMENTOS



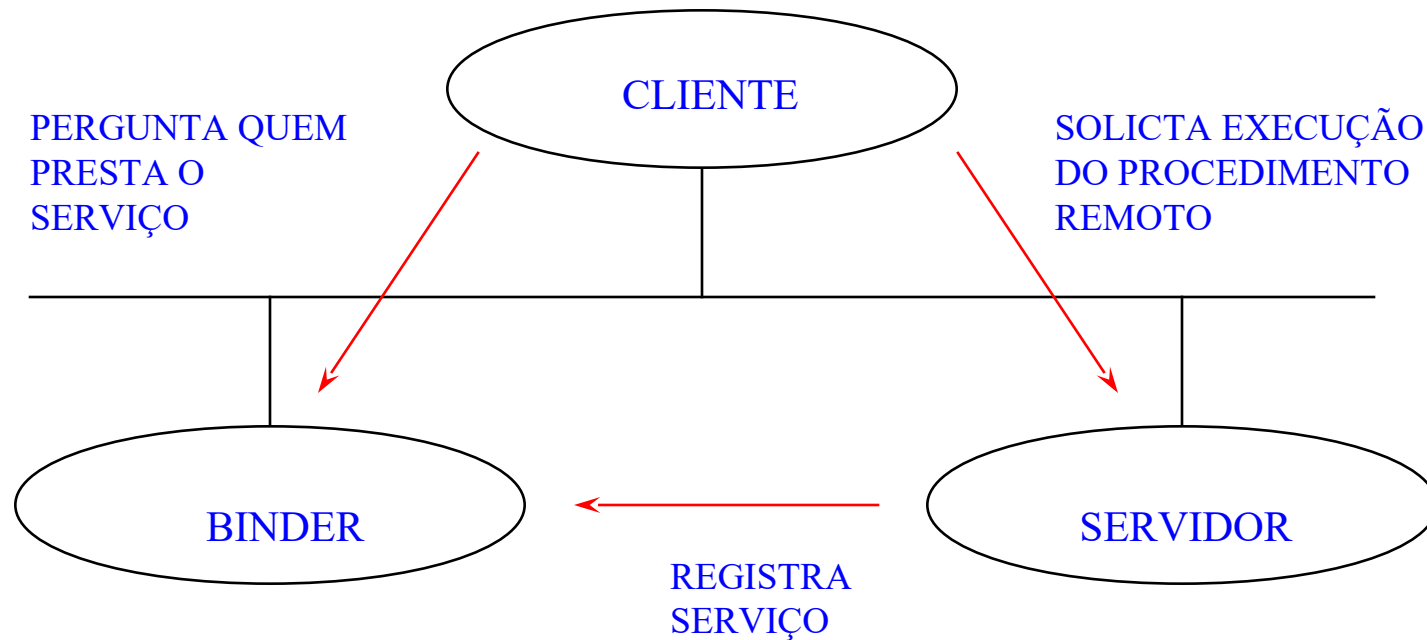
# Distribuição e concorrência



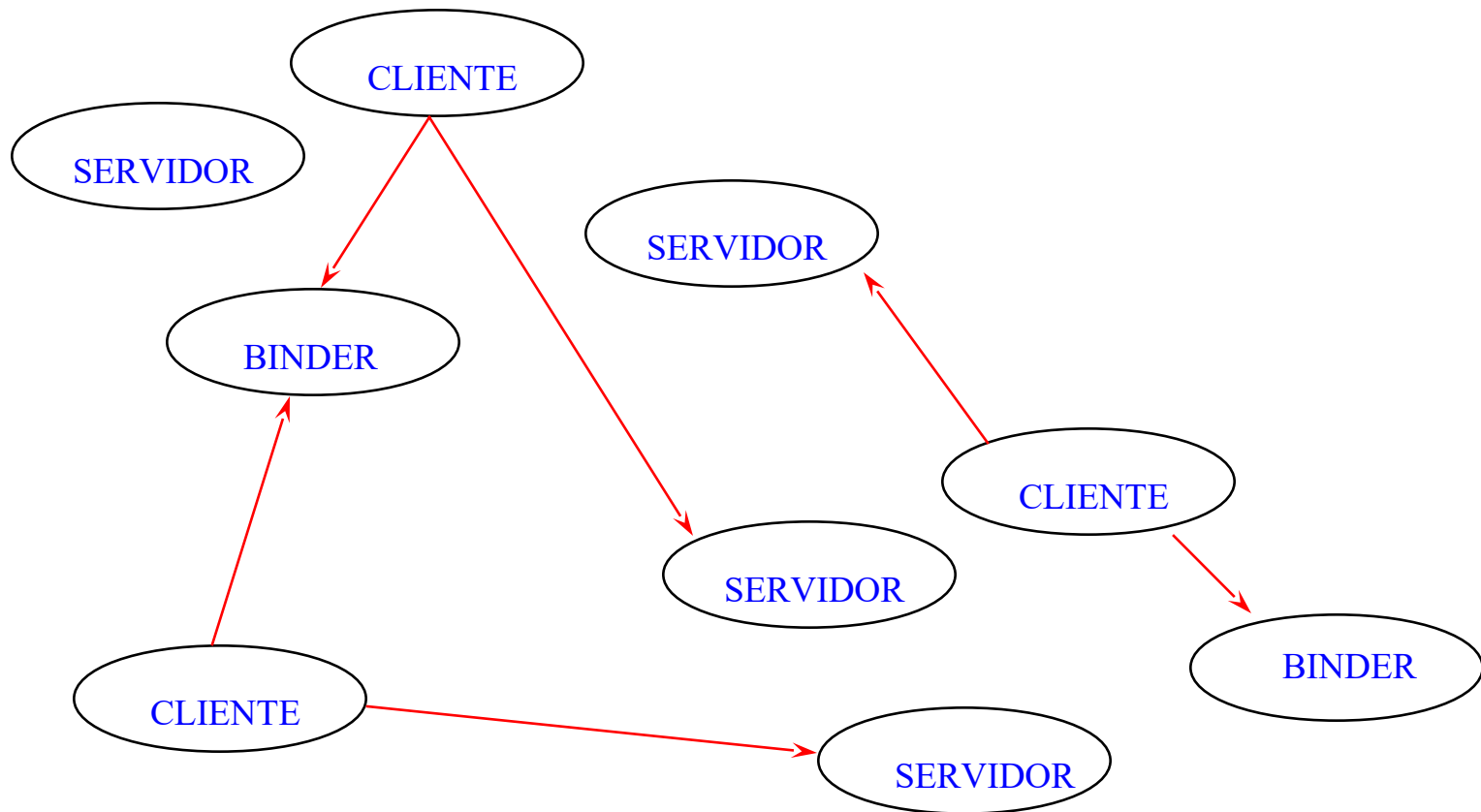
## ✓ Localização do servidor:

- Servidor localiza *binder*.
- Servidor registra serviço com *binder*.
- Cliente identifica *binder*.
- *Binder* informa endereços de servidores ao cliente.
- Cliente contacta servidor.

# Distribuição e concorrência



# Distribuição e concorrência



# Distribuição e concorrência



## ▼ Componentes na geração de código:

- Programa cliente.
- Programa servidor.
- Arquivo com definição de interface.
- Pré-processador.
- Compilador para a linguagem de programação adotada.

## ▼ Arquivo com definição de interface:

- Relaciona procedimentos remotos.
- Especifica tipos e quantidades de parâmetros.

# Distribuição e concorrência



CÓDIGO CLIENTE      ESPECIFICAÇÃO RPC



RPCGEN



COMPILADOR



PROGRAMA CLIENTE

CÓDIGO SERVIDOR      ESPECIFICAÇÃO RPC



RPCGEN



COMPILADOR



PROGRAMA SERVIDOR



# LINGUAGEM C++

# Tópicos



- ✓ Introdução.
- ✓ Ambientes de desenvolvimento.
- ✓ Processo de compilação.
- ✓ Espaço de nomes.
- ✓ Escopo.
- ✓ Diretivas de compilação.
- ✓ Comentários.
- ✓ Decomposição em funções.
- ✓ Biblioteca padrão.
- ✓ Tipos básicos.
- ✓ Variáveis e constantes.
- ✓ Cadeia de caracteres.
- ✓ Operadores e expressões.
- ✓ Controle de fluxo.
- ✓ Controle de iteração.
- ✓ Matrizes.
- ✓ Ponteiros.
- ✓ Classes.
- ✓ Atributos.
- ✓ Métodos.
- ✓ Objetos.
- ✓ Construtores e destrutores.
- ✓ Tratamento de exceção.
- ✓ Sobrecarga de operadores.
- ✓ Herança.
- ✓ Modularização.




# Introdução



## ✓ Introdução:

- Desenvolvida por Bjarne Stroustrup na AT&T.
- Inicialmente uma extensão da linguagem C.
- Linguagem de programação genérica.
- Linguagem de programação híbrida.
- Possibilita introdução gradual de novos conceitos.
- Linguagem sem proprietário.
- Linguagem com forte verificação de tipos.
- Linguagem com sintaxe considerada complexa.

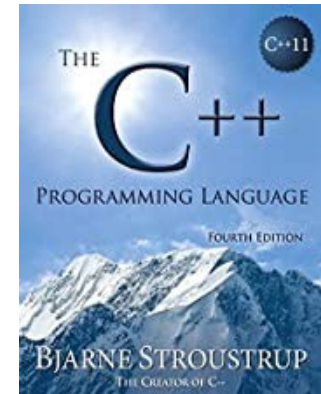
# Introdução

- 
- Interfaces de programação parcialmente padronizadas.
  - Mudança em interface requer manutenção.
  - Linguagem de programação compilada.
  - Existem diversos fornecedores de compiladores.
  - Existem compiladores para diversas plataformas.
  - Migração entre plataformas pode requerer recompilação.
  - Compilador gera código para plataforma alvo específica.
  - Código gerado tipicamente possibilita alto desempenho.

# Introdução

## ✓ The C++ Programming Language:

- Primeiro livro a descrever o C++.
- A documentação *de facto* por vários anos.
- Primeira edição em 1985.
- Escrito por Bjarne Stroustrup.



# Introdução

## ✓ Padronização:

- Padronização pelo ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21).

ANO	PADRÃO	NOME
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2011	ISO/IEC 14882:2011	C++11, C++0x
2014	ISO/IEC 14882:2014	C++14, C++1y
2017	ISO/IEC 14882:2017	C++17, C++1z
2020	A ser determinado.	C++20, C++2a

# Introdução

## ✓ Standard C++ Foundation:

- Organização que provê suporte a desenvolvedores de software em C++.

Get Started! Tour Core Guidelines Super-FAQ Standardization About

## News, Status & Discussion about Standard C++

Follow All Posts

The home of Standard C++ on the web — news, status and discussion about the C++ standard on all compilers and platforms.

### Recent Highlights

**C++17 - La guía completa - Nico Josuttis**  
By Nico Josuttis | Jan 23, 2020 02:17 AM

**How and why overloading, templates, and auto deduction were invented? - Milad Kahsari**  
By Milad Kahsari Alhadi | Jan 23, 2020 02:05 AM

**C++ - Initialization of Static Variables-Pablo Arias**  
By Adrien Hamelin | Jan 21, 2020 12:38 PM

**C++20: Define Concepts-Rainer Grimm**  
By Adrien Hamelin | Jan 21, 2020 12:32 PM

**2 Lines Of Code and 3 C++17 Features - The overload Pattern-Bartłomiej Filipek**  
By Adrien Hamelin | Jan 20, 2020 11:35 AM

### Recent CppCast Podcasts

**Conference Organizing**  
Date: Thu, 16 Jan 2020 00:00:00 +0000

**Clang Hacking**  
Date: Thu, 09 Jan 2020 00:00:00 +0000

**C++ 2020 News**  
Date: Thu, 02 Jan 2020 00:00:00 +0000

**OpenVDB**  
Date: Thu, 19 Dec 2019 00:00:00 +0000

### Recent Cpp.Chat Podcasts

**Set a Breakpoint in the Past**  
Date: Fri, 17 Jan 2020 09:00:00 +0000

**FEATURES**  
Current ISO C++ status  
Upcoming ISO C++ meetings  
Upcoming C++ conferences  
Compiler conformance status

**TAGS**  
basics intermediate  
advanced experimental

**UPCOMING EVENTS**  
emBo++ 2020  
Mar 13-15, Bochum, Germany  
ACCU 2020

# Introdução



## ✓ Exemplos de mudanças em relação ao C:

- Suporte para classes e objetos.
- Suporte para construção de hierarquias de classes.
- Sobrecarga de funções.
- Passagem de parâmetros por referência.
- Funções podem ter quantidade variável de parâmetros.
- Funções podem ser *inline*.
- Resolução de escopo flexível via operador `::`.

# Ambientes de desenvolvimento



## ✓ Características:

- Ambiente integrado de desenvolvimento (IDE).
- Existem IDEs para diversas plataformas.
- Existem IDEs comerciais e IDEs não comerciais.

## ✓ Exemplos:

- |                            |           |
|----------------------------|-----------|
| • Eclipse.                 | CLion.    |
| • Code::Blocks.            | CodeLite. |
| • Dev-C++.                 |           |
| • Microsoft Visual Studio. |           |

# Ambientes de desenvolvimento

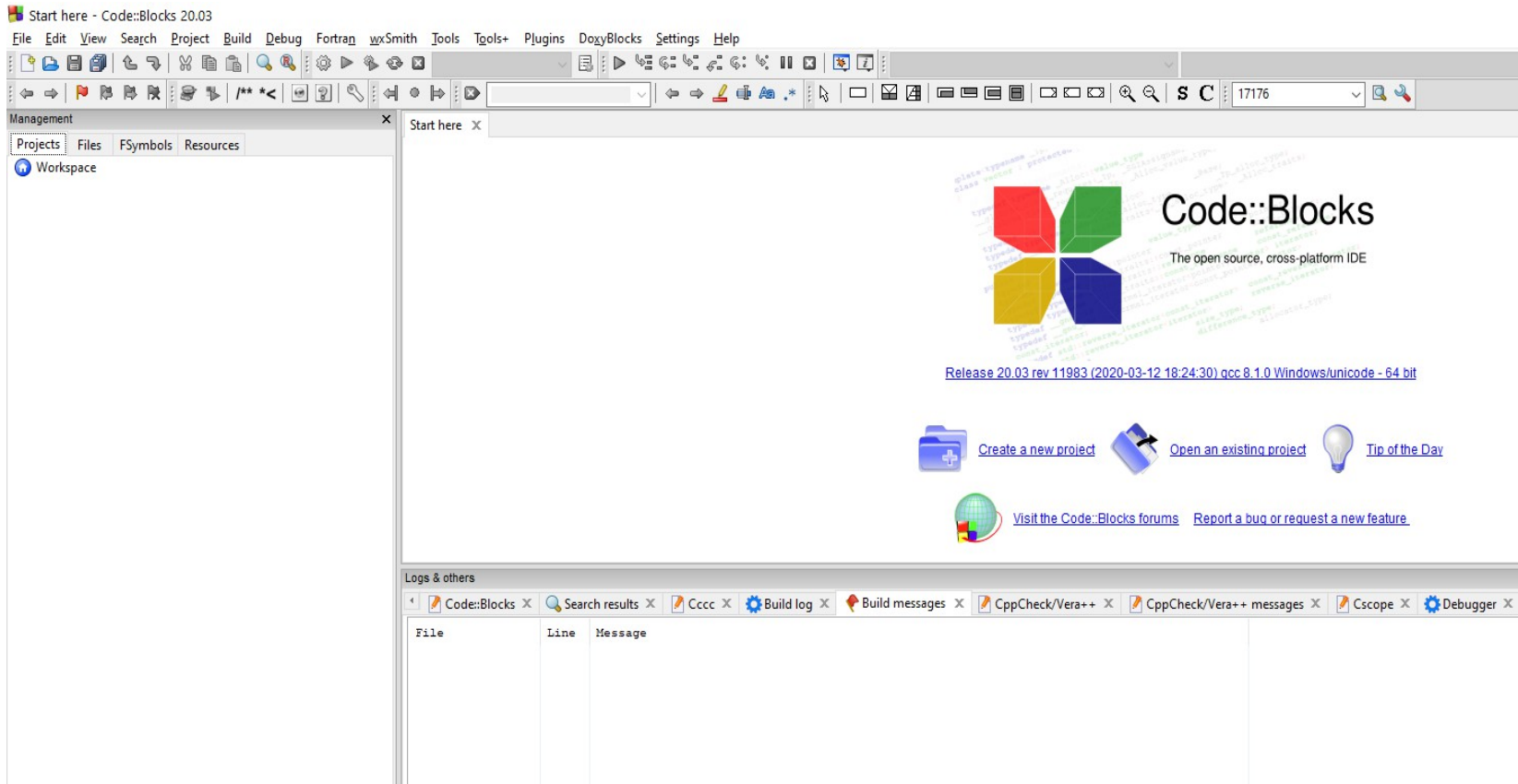


## ✓ Code::Blocks:

- Software aberto (*open source*).
- Codificado em C++.
- Provê suporte a múltiplas plataformas.
- Pode ser estendido por meio de *plugins*.
- Provê suporte a múltiplos compiladores.
- Elevado desempenho.
- Provê depurador.
- Ambiente configurável.
- Interface com o usuário é de fácil entendimento.

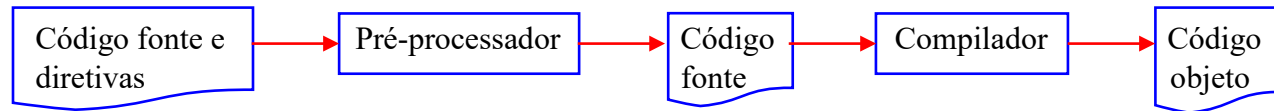


# Ambientes de desenvolvimento



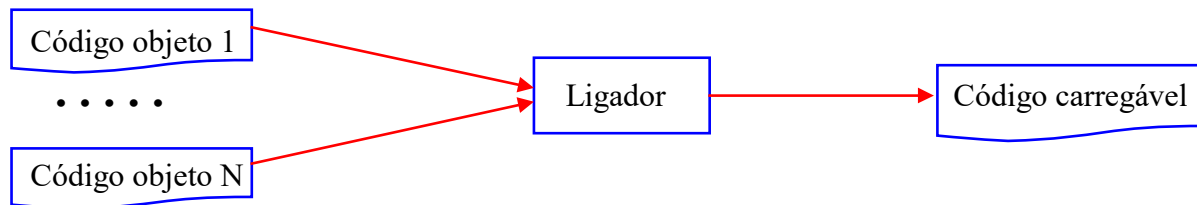
# Processo de compilação

## ✓ Pré-processamento e compilação:



## ✓ Ligação (*linking*):

- Combina códigos em formato objeto.
- Produz código que será carregado para execução.
- Atribuídos endereços e resolvidas referências externas.



# Processo de compilação



## ✓ Carregador (*loader*):

- Pode ser parte de sistema operacional.
- Etapa de processo requerido para execução de código.
- Determina endereço onde código será armazenado.
- Verifica se há memória suficiente.
- Atualiza endereços relocáveis.
- Copia código para memória.
- Desloca fluxo para instrução a ser inicialmente executada.

# Espaço de nomes



## ✓ Conceitos:

- Particiona o espaço de nomes.
- Evita a ocorrência de conflitos entre nomes.
- Agrupa elementos relacionados.
- Podem existir declarações e definições em um *namespace*.
- Definições podem ser feitas fora do corpo do *namespace*.
- Usado o operador de resolução de escopo `::`.
- Espaços de nomes podem ser aninhados.
- Podem existir múltiplas ocorrências de um *namespace*.
- Diretiva *using* identifica o *namespace* usado.

# Espaço de nomes



```
namespace string {  
    void strcpy(char* source, char* from);  
    void strcat(char* source, char* from);  
    int  strlen(char* source);  
}  
  
void string::strcpy(char* source, char* from) {  
    ...  
}  
  
void string::strcat(char* source, char* from) {  
    ...  
}  
  
int string::strlen(char* source) {  
    ...  
}
```

# Espaço de nomes

```
namespace nameSpaceA {  
    int var = 5;  
}  
  
namespace nameSpaceB {  
    double var = 3.1416;  
}  
  
...  
cout << nameSpaceA::var;  
cout << nameSpaceB::var;  
...
```

```
namespace nameSpaceA {  
    int var = 5;  
}  
  
namespace nameSpaceB {  
    double var = 3.1416;  
}  
  
...  
using namespace nameSpaceB;  
  
cout << var;  
cout << (var*2);  
...
```

# Espaço de nomes



## ▼ Declaração de *namespace* :

```
namespace Fila {  
  
    typedef struct passageiro {  
        char *nome;  
        int  cpf;  
        struct passageiro *proximo;  
    } tipo_passageiro;  
  
    void inserir(tipo_passageiro*);  
    tipo_passageiro* remover();  
}
```

# Espaço de nomes



## ✓ Definição de *namespace* :

```
namespace Fila {  
    tipo_passageiro *primeiro = 0, *ultimo = 0;  
}  
  
void Fila::inserir(tipo_passageiro *ptr)  
{  
    if(primeiro == 0)  
        primeiro = ptr;  
    else  
        ultimo->proximo = ptr;  
  
    ultimo = ptr;  
}
```



# Espaço de nomes



## ✓ Informando o *namespace* :

```
using namespace std;

int main(int argc, char *argv[])
{
    using Fila::tipo_passageiro;
    using Fila::inserir;
    using Fila::remover;
```

## ✓ Declarando sinônimo:

```
// Exemplo de sinônimo.
```

```
namespace GR = Grafico;
```

# Espaço de nomes



## ✓ Informando o *namespace* :

```
using namespace std;

int main(int argc, char *argv[])
{
    using Fila::tipo_passageiro;
    using Fila::inserir;
    using Fila::remover;
```

## ✓ Declarando sinônimo:

```
// Exemplo de sinônimo.
```

```
namespace GR = Grafico;
```

# Espaço de nomes



## ▼ O *namespace std* :

- Espaço de nomes da biblioteca padrão.
- Funções, classes e *templates* declarados no *namespace*.
- Existem várias formas de informar o espaço de nomes.

```
#include <iostream>
```

```
...
```

```
using namespace std;
```

```
#include <iostream>
```

```
...
```

```
using std::cin;  
using std::cout;
```

```
#include<iostream>
```

```
...
```

```
std::cin  >> i;  
std::cout << j;
```

# Escopo



## ✓ Regras de escopo:

- Determinam onde identificador é conhecido.
- Escopo de um nome inicia em seu ponto de declaração.

## ✓ Escopo de nome:

- Global.
- Classe.
- Função.
- Bloco delimitado por `{` e `}`.

# Diretivas de compilação



## ▼ Diretiva de compilação:

- Controla ação do pré-processador.
- São executadas operações sobre o código fonte.
- Identificadas através do caractere #.
- Terminadas ao final da linha.
- Se precisar continuar após final da linha é usado \ .
- Pode aparecer em qualquer parte do código fonte.
- Efeito dura até que nova diretiva cancele a operação.

# Diretivas de compilação



## ▼ Diretiva *#define* :

- Informa identificador que deve ser substituído.
- Pode resultar em código mais fácil de entender e adaptar.
- Pode ser usada na definição de macro.
- Macro é expandida pelo pré-processador.
- Uso de macro pode resultar em ganho de desempenho.
- Uso de macro pode resultar em maior gasto de memória.
- Definição é cancelada por meio da diretiva *#undef*.

# Diretivas de compilação



## ▼ Definição de constante:

```
#define LIMITE 100  
if ( valor > LIMITE )  
equivale à  
if ( valor > 100 )
```

## ▼ Definição de macro:

```
#define SOMA( a, b ) ( a + b )  
resultado = SOMA( valor_a, valor_b );  
equivale a  
resultado = ( valor_a + valor_b );
```

# Diretivas de compilação

```

/*****
    MACROS USADAS NO KERNEL
*****/

#define    QUEUE_IS_NOT_OK ((queue<0) || (queue>(NQUEUES-1)))
#define    PID_IS_NOT_OK   ((pid<0) || (pid>(MAXUSER+MAXSERVER+MAXTASK-1)))
#define    NOT_USER        (process[pid].type!=USER)
#define    NOT_SERVER      (process[pid].type!=SERVER)
#define    NOT_TASK        (process[pid].type!=TASK)
#define    TYPE_IS_NOT_OK  (NOT_USER && NOT_SERVER && NOT_TASK)
#define    SEM_IS_NOT_OK   ((sem<0) || (sem>(MAXSEM + N_LEVEL_INT - 1)))
#define    ST_NOT_USER     (process[pid].status!=USER)
#define    ST_NOT_SERVER   (process[pid].status!=SERVER)
#define    ST_NOT_TASK     (process[pid].status!=TASK)
#define    NOT_READY       (ST_NOT_USER && ST_NOT_SERVER && ST_NOT_TASK)
#define    MSEC_IS_NOT_OK  ((msec<0) || (msec>MAXMSEC))

/*****
    MACROS USADAS NO SERVIDOR DE PROCESSOS E DE MEMORIA
*****/

#define    ENTRY_NOT_LVL_1  ((level!=LEVEL_1) || (entry<0) || (entry>=256))
#define    ENTRY_NOT_LVL_2  ((level!=LEVEL_2) || (entry<0) || (entry>=128))
#define    ENTRY_OR_LEVEL_NOK (ENTRY_NOT_LVL_1 && ENTRY_NOT_LVL_2)

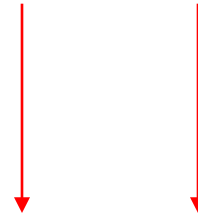
```



# Diretivas de compilação



Parênteses evitam problemas na expansão.



```
#define SOMA( a, b ) ( (a) + (b) )  
resultado = SOMA( valor_a, valor_b );  
equivale a  
resultado = ( valor_a + valor_b );
```

# Diretivas de compilação



## ▼ Contrastando *const* e *inline* com *#define* :

- Nomes definidos são substituídos antes da compilação.
- Compilador tipicamente não conhece os nomes.
- Isso pode acarretar dificuldades na depuração.
- Possível alternativa é usar *const*.

```
const double TAXA = 2.5;
```

```
const char * const CIDADE = "Brasilia";
```

# Diretivas de compilação

- Atentar para a expansão.

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

- Exemplos de dificuldades.

```
int valor = 10, limite = 0;  
max(++valor, limite);  
max(++valor, limite+20);
```

- Alternativa mais segura é usar *inline* .

```
inline int max(int a, int b) {return a > b ? a:b;}
```

# Diretivas de compilação



## ▼ Diretiva *#include* :

- Funções são tipicamente agrupadas em bibliotecas.
- Declarações são importadas quando da compilação.
- Diretiva inclui arquivo com declarações.
- Insere conteúdo do arquivo.
- Nome de arquivo pode se encontrar entre `< >` ou `" "`.
- Podem ser aninhadas diretivas *#include*.

```
#include <stdio.h>

# include "constantes"

main ( ) {

    printf ( "Teste" );

}
```

# Diretivas de compilação



## ▼ Diretivas de compilação condicional:

- # if expressão
- # elif expressão
- # ifdef identificador
- # ifndef identificador
- # else
- # endif

# Diretivas de compilação



## v Exemplo:

```
# if VALOR > 10
dado = dado * 3;
# elif VALOR > 5
dado = dado * 2
# endif
```

## v Exemplo:

```
# if CONTAGEM < 10
val = val + 7;
# else
val = val + 3;
# endif
```

## v Exemplo:

```
# ifdef DEBUG
printf ("%d", cont );
# endif
```

# Comentários

## v Comentário:

- Texto entre `/*` e `*/`.
- Comentário de uma linha pode ser precedido por `//`.
- Não é permitido aninhar comentários.
- Deve ser adotada formatação adequada.
- Deve ser adotado padrão para comentários.

## v Exemplos:

```
/*  
/*  Primeiro exemplo de texto de comentário.  
*/  
/*  Segundo exemplo de texto de comentário */  
//  Terceiro exemplo de texto de comentário.
```

# Decomposição em funções



## ▼ Função:

- Porção auto-contida de código.
- Pode ser usada para decompor código de programa.
- Possui interface.
- Possui corpo.
- Corpo composto por código de implementação da função.
- Pode ter parâmetros.
- Pode retornar dado.
- Pode integrar biblioteca e ser reusada.



# Decomposição em funções



## ▼ Informação em especificação de função:

- Descrição de objetivo da função.
- Acoplamento.
- Interface com o usuário.
- Pré-condições.
- Pós-condições.
- Requisitos.
- Hipóteses.

# Decomposição em funções



## ✓ Elementos em declaração de função:

- Nome.
- Tipos de parâmetros.
- Tipo de dado retornado.

## ✓ Elementos em definição de função:

- Detalhamento da declaração (nomes de parâmetros etc.).
- Código que constitui corpo da função.

# Decomposição em funções



## ▼ Declaração de função:

```
tipo_retorno  nome_da_função ( tipo, ... );
```

## ▼ Definição de função:

```
tipo_retorno nome_da_função ( tipo nome_par_1, ... ){  
    corpo da função  
    return dado;  
}
```

# Decomposição em funções

```
extern void wr_device();  
extern void wr_mmu();  
extern void dis_int();  
extern void en_int();  
extern void begin();  
extern void send();  
extern void receive();
```

```
unsigned short outwait();  
unsigned short set_time();  
unsigned short get_time();  
unsigned short inc_time();  
unsigned short sleep();  
unsigned short initialise();  
unsigned short outsleep();
```

```
void select_proc();  
void hw_init();  
void var_init();  
void mem_init();  
void copy_mess();  
void free_buffer();  
void scheduler();
```

# Decomposição em funções

```
void    free_buffer(unsigned short id)
{
    if(head[BUFFQUEUE]==EOQ)
    {
        buffer[id].next=EOQ;
        head[BUFFQUEUE]=id;
    }
    else
    {
        buffer[tail[BUFFQUEUE]].next=id;
        buffer[id].next=EOQ;
    }
    tail[BUFFQUEUE]=id;
    return;
}
```

```
unsigned short  suspend( unsigned short pid)
{
    unsigned short  status;

    dis_int();

    if(NOT_READY)
        status=FALSE;

    else
    {
        if(head[SUSPENDED]==EOQ)
        {
            suspending[pid].next=EOQ;
            head[SUSPENDED]=pid;
        }
        else
        {
            suspending[tail[SUSPENDED]].next=pid;
            suspending[pid].next=EOQ;
        }
        tail[SUSPENDED]=pid;
        status=TRUE;
    }
    en_int();
    return status;
}
```

# Decomposição em funções



- ✓ Valor *default* para argumento:

Valor *default* colocado à direita.

```
float divide ( float x, float y = 1 ) ;
```

```
float divide ( float x, float y ){  
    return ( x / y ) ;  
}
```

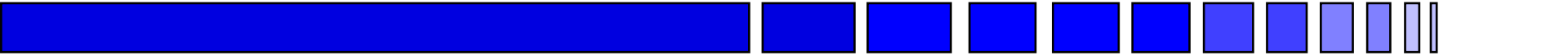
# Decomposição em funções



## ✓ Função *main* () :

- Todo programa contém uma função *main()*.
- Normalmente presente no início do código do programa.
- Contém código a ser inicialmente executado.
- Automaticamente chamada no início da execução.
- Aguarda parâmetros *argc* e *argv*.
- Parâmetro *argc* indica quantidade de argumentos.
- Parâmetro *argv* é um ponteiro para uma matriz.
- Pelo menos o nome do programa é passado para a função.

# Decomposição em funções



```
Sistema biblioteca.cpp
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     system("PAUSE");
9     return EXIT_SUCCESS;
10 }
11
```



# Decomposição em funções



## ✓ Elementos em chamada de função:

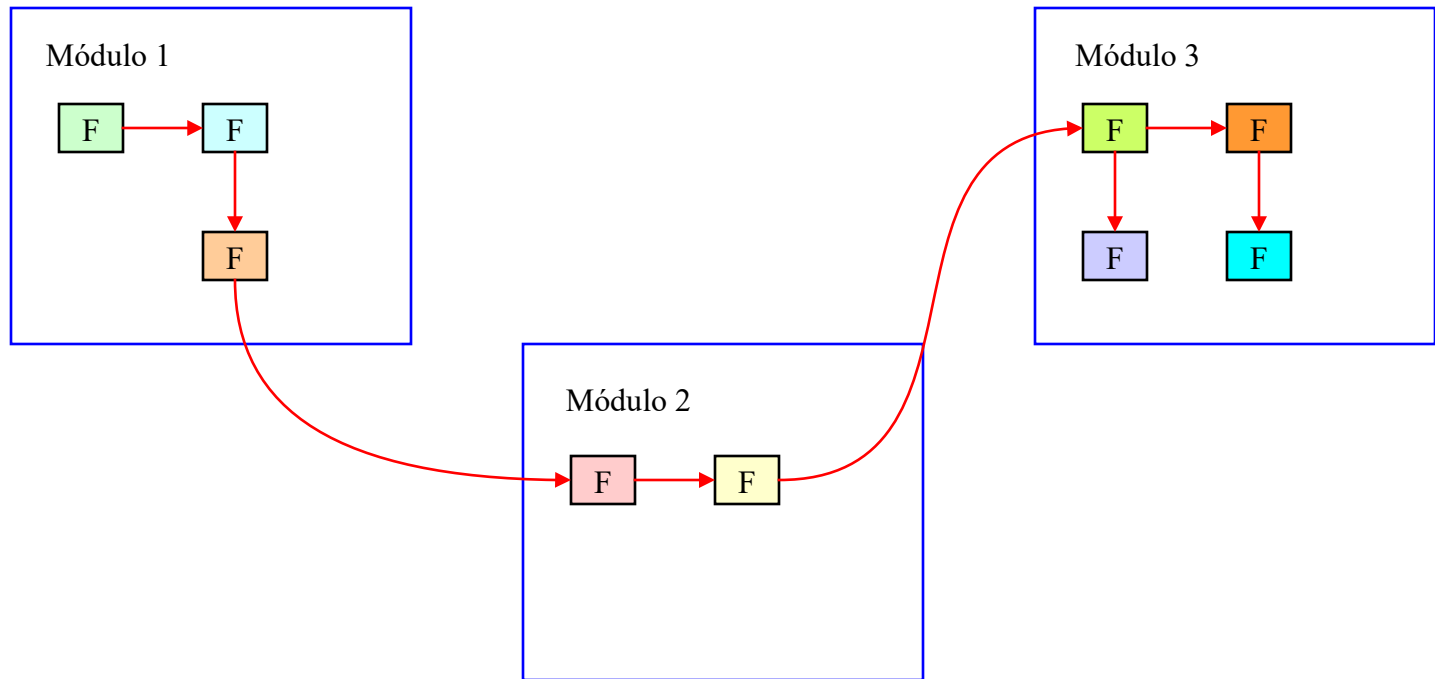
- Argumentos a serem usados pelos parâmetros.
- Parâmetros atuais (argumentos).
- Parâmetros formais.

## ✓ Estrutura de chamada:

- Estrutura definida pela chamada de uma função por outra.
- Estrutura pode envolver funções de diferentes módulos.

# Decomposição em funções

## ✓ Exemplo de estrutura de chamada:



# Biblioteca Padrão



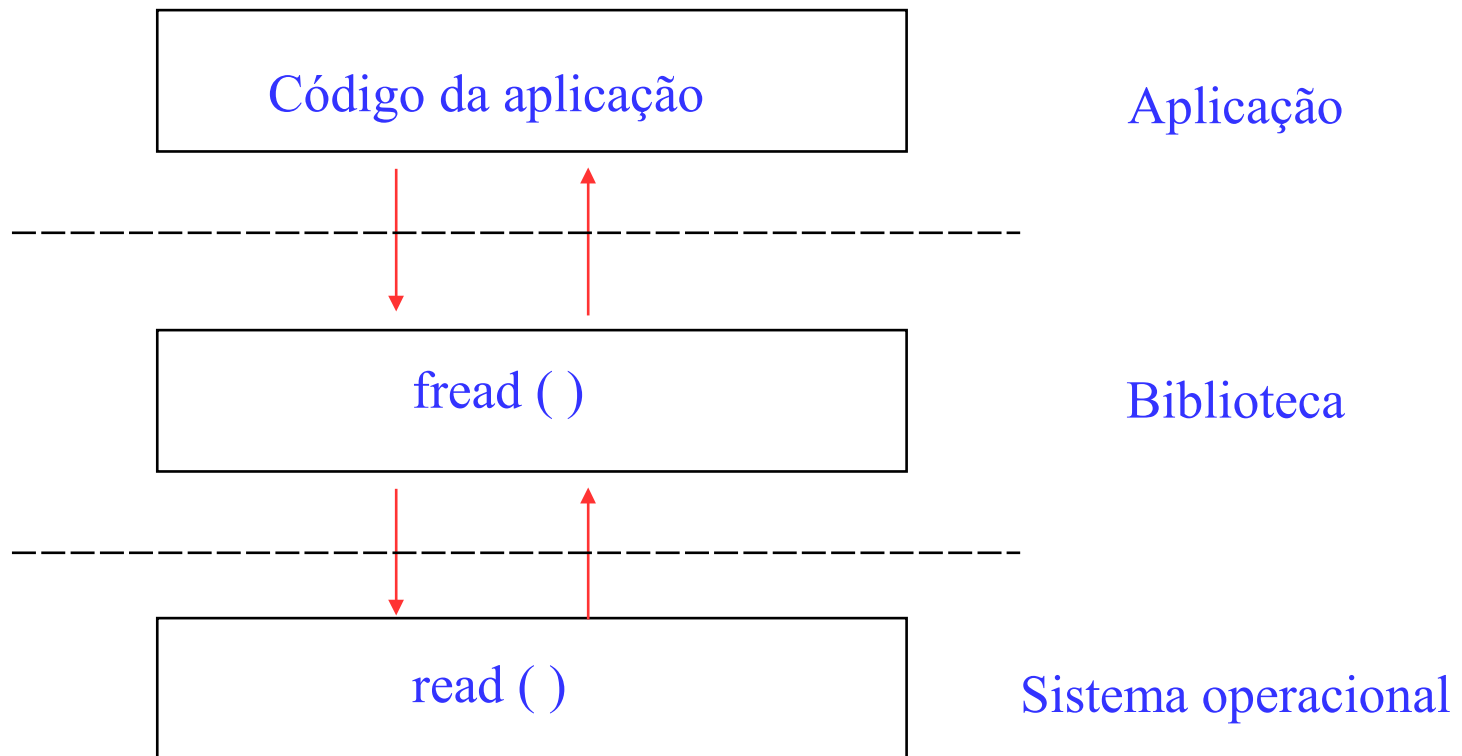
## ▼ Introdução:

- Biblioteca em C.
- Definida no ambiente de nomes *std*.
- Existem vários cabeçalhos padronizados.
- Provê suporte para execução de programas.
- Provê *streams* para entrada e saída de dados.
- Provê suporte para computação numérica.

# Biblioteca Padrão



- ▼ Elementos quando do uso da biblioteca:



# Biblioteca Padrão

## v Funções da biblioteca padrão:

abort	abs	acos	asctime	asin	atan	atan2
atexit	atof	atoi	atol			
bsearch						
ceil	calloc	clearerr	clock	cos	cosh	ctime
difftime	div					
exit	exp					
fabs	fclose	feof	ferror	fflush	fgetc	fgetpos
fgets	floor	fmod	fopen	fprintf	fputc	fputs
fread	free	freopen	frexp	fscanf	fsetpos	ftell
fwrite						
getc	getchar	getenv	gets	gmtime		
isalnum	isalpha	iscntrl	isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit			
labs	ldexp	ldiv	localeconv	localtime	log	log10
longjmp						
malloc	mblen	mbstowcs	mbtowc	memchr	memcmp	memcpy
memmove	memset	mktime	modf			
perror	printf	putc	putchar	puts		
qsort						
raise	rand	realloc	remove	rename	rewind	

# Biblioteca Padrão



scanf	setbuf	setlocale	setvbuf	sin	sprintf	sqrt
srand	strcmp	sscanf	strcat	strchr	strcoll	strcpy
strcspn	strerror	strftime	strlen	strncat	strncmp	strncpy
strbrk	strchr	strspn	strstr	strtod	strtok	strtol
strtoul	strxfrm	system				
tanh	time	tmpfile	tmpnam	tolower	toupper	
ungetc						
vprintf	vsprintf					
wcstombs	wctomb					

# Biblioteca Padrão



## ▼ Imprimir mensagem de erro:

```
void perror ( const char *s ) ;
```



Cadeia de caracteres a ser apresentada após a mensagem de erro.

## ▼ Escrever texto formatado na saída padrão:

```
int printf ( const char *format, . . . . ) ;
```



Ponteiro para cadeia de formatação.



Variáveis a serem escritas.

# Biblioteca Padrão



## ▼ Exemplos de formatação para valor de $Pi$ :

<code>%f</code>	3.141593
<code>%e</code>	3.141593e+000
<code>%-15.2f</code>	3.14
<code>%010.1f</code>	00000003.1
<code>%#010.1f</code>	00000003.1
<code>%+10g</code>	+3.141593
<code>%+10e</code>	+3.141593e+000
<code>%-20e</code>	3.141593e+000
<code>%025e</code>	00000000000003.141593e+000
<code>%025.20e</code>	3.14159265358979300000e+000



# Biblioteca Padrão



- ✓ Escrever caractere na saída padrão:

```
int putchar ( int c ) ;
```

- ✓ Escrever cadeia na saída padrão:

```
int puts ( const char *s ) ;
```

- ✓ Ler texto formatado da entrada padrão:

```
int scanf ( const char *format, . . . ) ;
```



Ponteiro para cadeia  
de formatação.



Endereço das variáveis  
onde armazenar.

# Biblioteca Padrão



## ✓ Cadeia de formatação *format* :

- Controla a leitura dos dados.
- Contém texto e especificadores de conversão.
- Contém zero ou mais diretivas.
- Cada diretiva é iniciada com caractere %.

## ✓ Após caractere % na cadeia de formatação:

- |                 |   |
|-----------------|---|
| • <i>star</i>   | - caractere * opcional                                    |
| • <i>width</i>  | - tamanho máximo do campo, opcional                       |
| • <i>type</i>   | - opcional ( <i>h</i> , <i>l</i> , <i>k</i> ou <i>L</i> ) |
| • <i>format</i> | - tipo da conversão a ser realizada                       |

# Biblioteca Padrão



## ✓ Exemplos de diretivas:

- `%c` - Lê um byte - argumento deve ser `char *`
- `%d` - Lê seqüência de dígitos decimais - argumento deve ser `int *`
- `%e` - Lê um ponto flutuante - argumento deve ser `float*`
- `%f` - O mesmo que `%e`
- `%g` - O mesmo que `%e`
- `%i` - Lê seqüência de dígitos decimais - argumento deve ser `int*`
- `%o` - Lê seqüência de dígitos octais - argumento `unsigned int*`
- `%p` - Lê um ponteiro - argumento deve ser `void*`
- `%s` - Lê cadeia sem espaços - argumento deve ser `char*`
- `%u` - Lê cadeia de dígitos decimais - argumento `unsigned int*`
- `%x` - Lê seqüência de dígitos hexadecimais - argumento `unsigned int*`

# Biblioteca Padrão



## ▼ Observações:

- Tamanho especificado com *%s* caso contrário pode ocorrer *overflow* de memória.
- Tamanho deve incluir espaço para caractere de terminação.

## ▼ Ler caractere da entrada padrão:

```
int getchar ( void ) ;
```

## ▼ Ler cadeia de caracteres da entrada padrão:

```
char *gets ( char *s ) ;
```

# Biblioteca Padrão



## v Observações:

- Função *printf()* aguarda uma quantidade variável de parâmetros de tipos indeterminados.
- Compiladores podem não indicar erro em enunciados como:

```
double d ;  
printf ( "Resposta = %s", d ) ;
```

- Saída de grande quantidade de dados deve ser realizada por várias chamadas à *printf()* e não por uma única chamada.

# Biblioteca Padrão



## ✓ Teste e conversão entre tipos:

- |  |  |
|--|--|
| • <code>int isalnum ( int c );</code>  | -Testa se é caractere numérico ou alfabético |
| • <code>int isalpha ( int c );</code>  | -Testa se é caractere alfabético             |
| • <code>int isctrl ( int c );</code>   | -Testa se é caractere de controle            |
| • <code>int isdigit ( int c );</code>  | -Testa se é dígito de 0 a 9                  |
| • <code>int islower ( int c );</code>  | -Testa se é letra minúscula                  |
| • <code>int isprint ( int c );</code>  | -Testa se é caractere que pode ser impresso  |
| • <code>int ispunct ( int c );</code>  | -Testa se não é espaço, letra ou dígito      |
| • <code>int isspace ( int c );</code>  | -Testa se é espaço                           |
| • <code>int isupper ( int c );</code>  | -Testa se é letra maiúscula                  |
| • <code>int isxdigit ( int c );</code> | -Testa se é dígito hexadecimal               |
| • <code>int tolower ( int c );</code>  | - Converte de maiúscula para minúscula       |
| • <code>int toupper ( int c );</code>  | - Converte de minúscula para maiúscula       |

# Tipos básicos



## v Tipo lógico:

- `bool nome;`

- *true* ou *false*

## v Caractere:

- `char nome;`
- `signed char nome;`
- `unsigned char nome ;`

- caractere sinalizado  
- caractere sinalizado  
- caractere não sinalizado

## v Inteiro:

- `int nome;`
- `short nome;`
- `short int nome;`
- `unsigned int nome;`

- short sinalizado  
- short sinalizado  
- short sinalizado  
- short não sinalizado

# Tipos básicos



- |                            |                        |
|----------------------------|------------------------|
| • unsigned short int nome; | - short não sinalizado |
| • long int nome;           | - long sinalizado      |
| • long nome;               | - long sinalizado      |
| • unsigned long nome;      | - long não sinalizado  |

## ✓ Ponto-flutuante:

- |                     |                      |
|---------------------|----------------------|
| • float nome;       | - precisão simples   |
| • double nome;      | - precisão dupla     |
| • long double nome; | - precisão estendida |

## ✓ Tipo *void* :

- |              |                                   |
|--------------|-----------------------------------|
| • void f( ); | - função não devolve valor        |
| • void* ptr; | - ponteiro para tipo desconhecido |



# Variáveis e constantes



## ✓ Componentes de declaração:

```
modificador tipo nome = valor_inicial;
```

## ✓ Modificadores de tipos:

- Automática
- Registrador
  - `register` tipo nome;
- Externa
  - `extern` tipo nome;
- Estática
  - `static` tipo nome;
- Constante
  - `const` tipo nome;
- Volátil
  - `volatile` tipo nome;

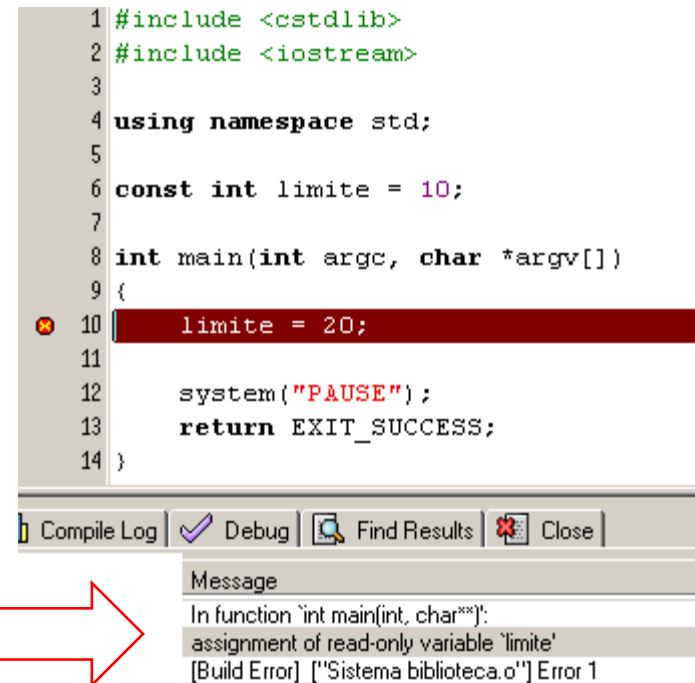
# Variáveis e constantes

## ✓ Palavra-chave *const* :

- Informa que elemento não será modificado.

## ✓ Exemplo:

```
const int limite = 10;
```



```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 const int limite = 10;
7
8 int main(int argc, char *argv[])
9 {
10     limite = 20;
11
12     system("PAUSE");
13     return EXIT_SUCCESS;
14 }
```

Indicação de erro

Message  
In function 'int main(int, char\*\*):  
assignment of read-only variable 'limite'  
[Build Error] ["Sistema biblioteca.o"] Error 1

# Variáveis e constantes



## ✓ Constantes:

- Tipo caracter - 'a' 'B' '\ n'
- Tipo inteiro - 30 30u 50000 50000u
- Ponto-flutuante - 12.2234e2 1444. 0.345

## ✓ Exemplos:

30	signed integer	25L	long
30u	unsigned integer	25uL	unsigned long
50000	signed long	34.45e12	double
50000u	unsigned long	222.3e10f	float

# Variáveis e constantes



## v Exemplos de sequências de escape:

<code>\a</code>	- sinal de áudio
<code>\b</code>	- retrocede
<code>\f</code>	- alimenta folha
<code>\n</code>	- nova linha
<code>\r</code>	- retorno do carro
<code>\t</code>	- tabulação horizontal
<code>\v</code>	- tabulação vertical
<code>\0dd</code>	- valor octal
<code>\0xdd</code>	- valor hexadecimal
<code>\0</code>	- nulo

# Variáveis e constantes



## ✓ Palavra-chave *volatile* :

- Informa ao compilador que não devem ser feitas otimizações envolvendo o identificador.

```
volatile int a, b;  
int i, tbl [100];  
.  
.  
.  
a = 5;  
b = 3;  
for ( i = 0 ; i <= 99 ; i++ ) tbl [i] = a + b ;
```

Garante execução do *loop* e evita otimizações.

Two red arrows originate from the text 'Garante execução do loop e evita otimizações.' One arrow points to the 'volatile' keyword in the first line of code, and the other points to the 'tbl' array identifier in the 'for' loop.

# Cadeia de caracteres



## ▼ Cadeia de caracteres em memória:

Cadeia	Memória	Caracter	ASCII
"Dia"	n	D	0x44
	n + 1	i	0x69
	n + 2	a	0x61
	n + 3	\0	0x00

```
mens = "Mensagem de aviso" ;  
mens = "Mensagem muito longa\  
        para uma linha" ;
```

← Declarações.

# Cadeia de caracteres



- ✓ Concatenar cadeias de caracteres:

```
char *strcat ( char *s1, const char *s2 );
```

- ✓ Procurar por caractere:

```
char *strchr ( const *s, int c ) ;
```

- ✓ Comparar cadeias de caracteres:

```
int strcmp ( const char *s1, const char *s2 ) ;  
int strcoll ( const char *s1, const char *s2 );
```

- ✓ Copiar cadeia de caractere:

```
char *strcpy ( char *s1, const char *s2 ) ;
```

# Cadeia de caracteres



- ▼ Tamanho da maior sequência diferente:

```
size_t  strcspn ( const char *s1, const char *s2 );
```

- ▼ Calcular tamanho de cadeia de caracteres:

```
size_t  strlen ( const char *s );
```

- ▼ Concatenar cadeias de caracteres:

```
char *  strcat ( char *s1, const char *s2, size_t n );
```

- ▼ Comparar cadeias de caracteres:

```
char *  strcmp ( char *s1, const char *s2, size_t n );
```



# Cadeia de caracteres



- ✓ Copiar cadeia de caracteres:

```
char *strncpy (char *s1, const char *s2, size_t n) ;
```

- ✓ Pesquisar cadeia por conjunto de caracteres:

```
char *strpbrk ( const char *s1, const char *s2 ) ;
```

- ✓ Localizar última ocorrência de caractere:

```
char *strrchr ( const char *s, int c ) ;
```

- ✓ Pesquisar cadeia por conjunto de caracteres:

```
size_t strspn ( const char *s1, const char *s2 ) ;
```

# Cadeia de caracteres



- ▼ Localizar sub-cadeia de caracteres:

```
char *strstr ( const char *s1, const char *s2 ) ;
```

- ▼ Dividir cadeia de caracteres em *tokens*:

```
char *strtok ( char *s1, const char *s2 );
```

- ▼ Converter cadeia de caracteres em outro tipo:

```
double strtod (const char *nptr, char **endptr);  
long int strtol (const char *nptr, char **endptr, int base);  
unsigned long int strtoul (const char *nptr, char **endptr, int base);  
double atof (const char *nptr);  
int atoi (const char *nptr);  
long int atol (const char *nptr);
```

# Operadores e expressões

## v Expressões e operadores unários:

- Tamanho de uma variável `sizeof nome`
- Tamanho de um tipo `sizeof ( nome )`
- Incremento `++`
- Decremento `--`
- Positivo `+`
- Negativo `-`
- Negação no nível de bit `~`
- Endereço `&`
- Indireção `*`
- Força tipo (type cast) `( tipo ) expressão`

# Operadores e expressões



## v Exemplos:

```
sizeof cidade ;  
sizeof ( int );  
saldo = saldo++ + 2;  
saldo = ++saldo + 2;  
saldo = saldo-- + 2;  
saldo = --saldo + 2;  
dado = + dado;  
dado = + 10;
```

```
dado = - dado;  
dado = - 20;  
mascara = ~mascara;  
ptr = &valor_maximo;  
*ptr = 15;
```

# Operadores e expressões



## ✓ Aritméticos:

- Multiplicação \*
- Divisão /
- Módulo %
- Adição +
- Subtração -

## ✓ Relacionais:

- Igual a ==
- Não igual a !=
- Menor que <
- Maior que >
- Menor ou igual <=
- Maior ou igual >=

# Operadores e expressões



## ✓ Lógicos:

- E            &&
- OU          ||
- Não        !

## ✓ No nível de bits:

- E            &
- OU          |
- Não        ~
- XOR        ^
- Shift       << e >>

## ✓ Exemplos:

```
if (a == b && c < 10)
if (c != d)
if (e == a || b == c)
```

## ✓ Exemplos:

```
c = a & f ;
e = b | g ;
b = a ^ f ;
```

# Controle de fluxo



## ✓ Enunciado *if–else* :

- Possibilita decidir qual enunciado executar.
- Teste é verdadeiro se valor diferente de zero.

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




Enunciados executados se teste for verdadeiro.



Enunciados executados se teste for falso.

# Controle de fluxo



```
if (QUEUE_IS_NOT_OK || PID_IS_NOT_OK)
    return SYSERROR;
else
{
    if (process[pid].status != FREEPROC)
        return SYSERROR;
    process[pid].status = queue;
    if (head[queue] == EOQ)
    {
        process[pid].next = EOQ;
        head[queue] = pid;
    }
    else
    {
        process[tail[queue]].next = pid;
        process[pid].next = EOQ;
    }
    tail[queue] = pid;
}

return SYSOK;
```



# Controle de fluxo



## ✓ Enunciado *switch – case* :

- Possibilita substituição de enunciados *if-else* aninhados.
- Possibilita seleção de conjunto de enunciados.

```
switch ( expressão ){  
    case constante_1: enunciado_1;  
                    break;  
    case constante_2: enunciado_2;  
                    break;  
    default:        enunciado_3;  
                    break;  
}
```

# Controle de fluxo

```
switch(field)
{
    case R:
        data=data&R_MASK;
        data=data>>R_SHIFT;
        break;
    case PL:
        data=data&PL_MASK;
        data=data>>PL_SHIFT;
        break;
    case V:
        data=data&V_MASK;
        data=data>>V_SHIFT;
        break;
    case PFN:
        data=data&PFN_MASK;
        data=data>>PFN_SHIFT;
        break;
    default:
        return SYSERROR;
    break;
}
```

```
switch(field)
{
    case PL:
        data=PL_MASK;
        info=info<<PL_SHIFT;
        break;
    case V:
        data=V_MASK;
        info=info<<V_SHIFT;
        break;
    case PFN:
        data=PFN_MASK;
        info=info<<PFN_SHIFT;
        break;
    default:
        return SYSERROR;
    break;
}
```

# Controle de fluxo



## v Operador condicional ?:

- Possibilita substituir enunciado *if* em certas situações.

```
nome = condição ? expressão_1: expressão_2 ;
```

- Equivale aos seguintes enunciados.

```
if ( condição )  
    nome = expressão_1;  
else  
    nome = expressão_2;
```

# Controle de iteração



## ✓ Enunciado *while* :

- Possibilita execução repetida de código.


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

## ✓ Enunciado *do-while* :

- Possibilita execução repetida de código.
- Execução de código ocorre pelo menos uma vez.

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

# Controle de iteração



```
count=0;
while ( (n<(MEM_SIZE/PAGE_SIZE)) &&(count<n_pages) )
{
    if (type==LEVEL_1)
    {
        if ( (memory[n]==FREE) &&(memory[n+1]==FREE) )
            count=2;

        n=n+2;
    }
    else
    {
        if (memory[n]==FREE)
            count=1;

        n=n+1;
    }
}
```

# Controle de iteração



## ▼ Enunciado *for* :


- Possibilita execução enquanto valor está dentro de faixa.

```
for ( expressão_1 ; expressão_2 ; expressão_3 )  
    enunciado;
```

- Equivale aos seguintes enunciados:

```
expressão_1;  
while ( expressão_2 ) {  
    enunciado_1;  
    expressão_3;  
}
```

# Controle de iteração



```
for (count1=0;count1<NQUEUES;count1++)
{
    head[count1]=EOQ;
    tail[count1]=EOQ;
}
head[FREEPROC]=0;
for (count1=0;count1<(MAXUSER+MAXSERVER+MAXTASK);count1++)
{
    process[count1].pid      =   count1;
    process[count1].status   =   FREEPROC;
    process[count1].pmt      =   0;
    process[count1].next     =   count1 + 1;
}
process[count1-1].next=EOQ;
tail[FREEPROC]=count1-1;
count1=0;
for (count2=NSTATES;count2<NQUEUES;count2++)
{
    semaphore[count1].queue=count2;
    semaphore[count1].status=FREE;
    count1=count1+1;
}
for (count1=0;count1<(MEM_SIZE/PAGE_SIZE);count1++)
    memory[count1]=FREE;
```

# Controle de iteração



## ✓ Enunciado *break* :

- Possibilita desvio de fluxo para enunciado seguinte a *loop* controlado por *switch*, *while*, *do-while* ou *for*.

## ✓ Enunciado *continue* :

- Finaliza apenas a iteração em execução.
- Desvia o fluxo de volta para expressão de condição.

## ✓ Enunciado *return* :

- Retorna da execução de função.
- Quando do retorno pode ser retornado ou não um valor.



# Matrizes



## v Declaração e inicialização:

```
int valores  [4] = { 2, 5, 1, 7 };  
int dados    [ ] = { 6, 8, 0 } ;  
int valores  [5] = { 3, 5, 1 } ;  
char nomes   [ ] = { 'J', 'O', 'A', 'O' };  
char nomes   [ ] = { "JORGE" } ;
```

## v Acesso a elemento:

```
dado = texto [0][2];
```

# Matrizes



## ✓ Armazenamento em memória:

Posição em memória	Elemento da matriz
$n$	$[0][0]$
$n + 1$	$[0][1]$
$n + 2$	$[1][0]$
$n + 3$	$[1][1]$



# Ponteiros



## v Declaração e atribuição de valor:

```
int  *ptr, dado ;  
ptr = &dado;
```

## v Acesso a valor apontado:

```
dado = *ptr ;
```

## v Espaço para armazenamento:

- Armazenamento em determinada quantidade de bytes.
- Quantidade de bytes independe do tipo apontado.

# Ponteiros



## ✓ Operações aritméticas:


- Soma, subtração, incremento e decremento.
- Comparação via operadores relacionais.
- Operações aritméticas sofrem processo de escala.
- Não são válidas no caso de ponteiros para funções.
- Subtração apenas entre ponteiros de mesmo tipo.
- Subtração resulta em quantidade de variáveis do tipo.

# Ponteiros



## ✓ Argumentos por valor ou por referência:

Por valor.      Por referência.



```
void raizes (float a, float b, float c, float *x1,  
            float *x2){  
    float d ;  
    d = sqrt ( b * b - 4 * a * c ) ;  
    *x1 = - ( b + d ) / ( 2 * a ) ;  
    *x2 =   ( d - b ) / ( 2 * a ) ;  
}
```

# Ponteiros



## v Passando matriz:

```
nome_da_função (nome_da_matriz);
```

## v Ponteiro para função:

```
tipo (*nome_ponteiro) ();
```

## v Obtendo endereço de função:

```
nome_ponteiro = nome_função ;
```

## v Chamando função usando ponteiro:

```
(*nome_ponteiro) (argumentos) ;
```

# Ponteiros



## ▼ Usando *const* com ponteiros:

- Informa que o elemento não será modificado.
- Uso típico na passagem por referência.
- Passagem por referência apresenta boa performance.
- Pode não ser desejável a alteração do valor original.

## ▼ Exemplo:

```
int copy (const char *from, char *to);
```



# Ponteiros



## ✓ Cuidados com a declaração:

`const char *sp;`

*sp* aponta para caracteres que não serão modificados quando apontados via *sp*.

`char *const sp ;`

*sp* não pode ser modificado mas valor apontado pode.

`const char *const sp;`

nem *sp* nem o valor apontado podem ser modificados.

# Classes

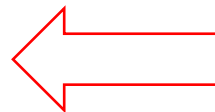
## v Declaração:

- Palavra-chave *class*.
- Nome da classe.
- Lista de atributos e métodos colocados entre chaves.

## v Visibilidade:

- Acesso aos métodos e aos atributos pode ser público, privado ou protegido.
- A visibilidade *default* é *private*.

`private`  
`public`  
`protected`



Especificadores  
da visibilidade de acesso.

# Classes



Palavra-chave.



```
class nome_da_classe {  
  
    especificador_acesso_1:  
        membro1;  
        ...  
    especificador_acesso_2:  
        membro2;  
        ...  
};
```

# Classes



## ✓ Classe aninhada:

- Classe pode ser declarada dentro de outra classe.
- Classe aninhada é visível dentro da classe externa.

```
class Retangulo {  
  
    class Coordenada {  
        int x, y;  
    };  
  
    Coordenada pt1, pt2, pt3, pt4;  
};
```

# Atributos



## ▼ Tipos dos atributos:

- Valor inicial dos atributos é aleatório.
- Atributos podem ser de instância ou de classe.
- Atributo de classe identificado pelo modificador *static*.
- Atributo constante identificado pelo modificador *const*.

```
class nome_da_classe {  
    tipo    nome_1;  
    static tipo    nome_2;  
    const  tipo    nome_3;  
};
```

# Atributos

- Atributo de instância.

```
class nome_da_classe {  
    tipo nome;  
};
```

- Atributo de classe via modificador *static*.

```
class nome_da_classe {  
    static tipo nome;  
};
```

- Atributo constante via modificador *const*.

```
class nome_da_classe {  
    const tipo nome;  
};
```

# Atributos

```
class Aluno {  
    // Atributos.  
    string nome;  
    int     matricula;  
    int     telefone;  
};
```

```
class Disciplina {  
    // Atributos.  
    private:  
        string nome;  
        int     codigo;  
};
```

# Atributos

- Constante com escopo de classe.

// Declaração de uma constante.

```
class CntrComunicacao {  
    static const int LIMITE;  
    . . .  
};
```

// Definição de uma constante.

```
const int CntrComunicacao::LIMITE = 10;
```



# Atributos

- Alternativamente para *int*, *bool*, *char* etc.

```
// Declaração de uma constante.
```

```
class CntrComunicacao {  
    static const int LIMITE = 10;  
    . . .  
};
```

```
// Definição de uma constante.
```

```
const int CntrComunicacao::LIMITE;
```



## ✓ Declaração e definição:

- Declaração informa a assinatura do método.
- Definição pode ocorrer quando da declaração.
- Definição pode ocorrer posteriormente.
- Definição posterior usa operador de resolução de escopo.

# Métodos



```
class Aluno {  
    // Lista dos atributos.  
    private:  
        string        nome;  
        int            matricula;  
        int            telefone;  
  
    // Lista dos métodos.  
    public:  
        void setNome(string nome) {  
            // Código do método.  
        }  
};
```

Método declarado  
e definido.



# Métodos



```
class Aluno {  
    // Lista dos atributos.  
    private:  
        string        nome;  
        int           matricula;  
        int           telefone;  
  
    // Lista dos métodos.  
    public:  
        void setNome(string nome);  
        void setMatricula(int matricula);  
        void setTelefone(int telefone);  
};
```

Métodos  
declarados  
mas não  
definidos.



# Métodos



```
// Definições dos métodos.
```

```
void Aluno::setNome(string nome) {  
    this->nome = nome;  
}
```

Operador  
de resolução  
de escopo.

```
void Aluno::setMatricula(int matricula) {  
    this->matricula = matricula;  
}
```

Ponteiro  
para o  
objeto.

```
void Aluno::setTelefone(int telefone) {  
    this->telefone = telefone;  
}
```

# Métodos

## ✓ Exemplo:

```
class Usuario {  
    private:  
        Nome nome;  
        Matricula matricula;  
        Senha senha;  
        Telefone telefone;  
  
    public:  
        Usuario(Nome, Matricula, Senha, Telefone);  
        Nome getNome();  
        void setNome(Nome);  
        Matricula getMatricula();  
        void setMatricula(Matricula);  
        Senha getSenha();  
        void setSenha(Senha);  
        Telefone getTelefone();  
        void setTelefone(Telefone);  
};
```

# Métodos

```
Usuario::Usuario(Nome nome, Matricula matricula,  
                Senha senha, Telefone telefone): nome(nome),  
                matricula(matricula), senha(senha), telefone(telefone) {  
}  
  
Nome Usuario::getNome() {  
    return nome;  
}  
  
void Usuario::setNome(Nome) {  
    this->nome = nome;  
}  
  
Matricula Usuario::getMatricula() {  
    return matricula;  
}
```



## ✓ Sobrecarga e polimorfismo:

- Métodos podem ser sobrecarregados.
- Métodos sobrecarregados tem o mesmo nome.

```
class Horario {  
    . . .  
    void obterHorario ( long* ticks );  
    void obterHorario ( int* hours,  
                        int* minutos, int* segundos );  
};
```



# Métodos

- Facilita desenvolvimento de aplicações portáteis.
- Facilita entendimento de aplicações.
- No nível de método.

```
class nome_da_classe {  
    tipo nome_do_método ( parâmetros );  
    tipo nome_do_método ( parâmetros );  
};
```

- No nível de operador.

```
tipo nome_da_classe::operator  
    símbolo (param) { . . .  
}
```

# Métodos



## ✓ Argumento *default*:

- Usado quando parâmetros atuais não são informados.
- Informado nas declarações de métodos.

```
class SymTable {  
    . . .  
public:  
    SymTable ( int sz = 16 );  
};
```



Construtor com  
valor *default* para  
argumento.

# Métodos

## v Método *inline*:

- Método pequeno e frequentemente invocado.
- Código inserido quando o método é invocado.
- Usada palavra-chave *inline* ou o método é definido na declaração da classe.

```
class ContaCorrente {  
    // . . .  
    public: void sacar ( float valor ) {  
        // . . .  
    }  
};
```

Método definido na  
declaração da classe.



# Métodos



```
class ContaCorrente {
```

```
    public:
```

```
        void sacar (float valor);
```

```
        ...
```

```
};
```

```
inline void ContaCorrente::sacar(float valor) {
```

```
    ...
```

```
}
```

Usada a  
palavra-chave.



# Métodos

## ▼ Método amigo:

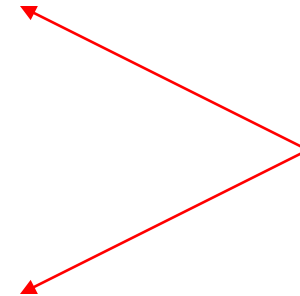
- Pode acessar membros privados de uma classe.
- Útil em alguns padrões de projeto.
- Usada a palavra-chave *friend*.
- Uma classe pode ser declarada como amiga de outra.

```
class nome_da_classe {  
  
    public:  
  
        friend tipo nome_da_classe::nome_método(param);  
        friend class nome_da_classe;  
  
    ...  
};
```

# Métodos



```
class IntSet {  
    // . . .  
  
public:  
    friend void RealSet::SetToInt ( IntSet * );  
};  
  
class RealSet {  
    // . . .  
  
public:  
    friend void IntSet::SetToReal ( RealSet * );  
};
```



Métodos  
amigos.

# Métodos


## ✓ Ejemplos:

```
class Horario {  
    // . . .  
    void obterHorario ( long* ticks );  
    void obterHorario ( int* hours,  
                        int* minutos, int* segundos );  
};  
  
class Conjunto {  
    // . . .  
public:  
    friend Bool operator & ( int, Conjunto );  
    friend Bool operator < (Conjunto, Conjunto);  
};
```

Sobrecarga de  
métodos



Sobrecarga de  
operadores



# Métodos



## ✓ Método e atributo estático:

- Atributo relacionado a uma classe e não às instâncias.
- Atributo identificado via palavra-chave *static*.
- Atributo estático é declarado na classe.
- Atributo estático é definido fora da classe.
- Atributo pode ser acessado por qualquer método da classe.
- Método estático pode ser invocado sem informar alvo.
- Método estático só pode acessar membros estáticos.
- Método estático é identificado por *static*.

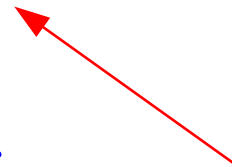


# Métodos

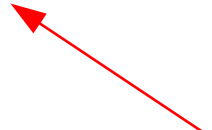


```
class Aluno {  
    // Lista dos atributos.  
    private:  
    static int    numeroAlunos;  
    ...  
}
```

```
int Aluno::numeroAlunos = 0;  
...
```



Declaração.

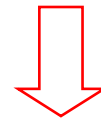


Posterior definição.

# Métodos

```
class Aluno {  
    ...  
private:  
    static int    numeroAlunos;  
    ...  
public:  
    static int getNumeroAlunos();  
    ...  
};  
...  
int Aluno::getNumeroAlunos() {  
    return numeroAlunos;  
}  
...  
int Aluno::numeroAlunos = 0;
```

Modos de  
invocar um  
método estático.



```
n = alunoA.getNumeroAlunos();  
...  
n = alunoB->getNumeroAlunos();  
...  
n = Aluno::getNumeroAlunos();
```

# Métodos



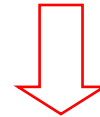
## ✓ Método e objeto constante:

- Objeto constante é identificado por *const*.
- Objeto constante é alterado apenas via construtores.
- Método constante não pode alterar estado do objeto.
- Método constante é identificado por *const*.
- Método constante pode atuar sobre objetos constantes.

# Métodos

```
class Aluno {  
    ...  
    // Lista dos métodos.  
public:  
    string getNome() const;  
    ...  
};  
  
...  
  
string Aluno::getNome() const {  
    return nome;  
}
```

Declaração de  
um objeto constante  
e acesso ao mesmo  
via método constante.



```
const Aluno alunoE("jose");  
  
...  
  
nome = alunoE.getNome();
```

## ✓ Atributo modificável:

- Pode-se ter atributo modificável em objeto constante.
- Atributo modificável é identificado via *mutable*.
- Atributo *mutable* é modificável via método *const*.

```
class Aluno {  
    private:  
        ...  
    mutable string nome;  
    public:  
        ...  
    static int getNumeroAlunos();  
};
```

```
void Aluno::setNome(string nome)  
const {  
    this->nome = nome;  
}  
...  
  
const Aluno alunoE;  
...  
alunoE.setNome("Maria");
```

# Objetos

## v Como criar objeto:

- Quando da declaração da classe.
- Após a declaração da classe.
- Dinamicamente após a declaração da classe.

```
class Aluno {  
    private:  
        string  nome;  
        int     matricula;  
        int     telefone;  
    // Lista dos métodos.  
    ...  
} alunoA, alunoB;
```

Objetos  
criados quando  
da declaração  
da classe.



# Objetos

```
class Aluno {  
    // Lista dos atributos.  
    private:  
        string  nome;  
        int     matricula;  
        int     telefone;  
    // Lista dos métodos.  
    ...  
};
```

```
int main(int argc, char *argv[]) {  
    Aluno alunoA, alunoB;  
    ...  
}
```

Objetos  
posteriormente  
criados.



# Objetos



## ✓ Criar e destruir dinamicamente objeto:

- Usar operadores *new* e *delete*.
- Operador *new* aloca memória e invoca o construtor.
- Operador *delete* invoca o *destrutor* e libera memória.
- Tornar claro se é matriz de objetos.
- Se alocada uma matriz, deve ser destruída uma matriz.
- Funções *malloc* e *free* não invocam construtor e destrutor.

```
ptrA = new NomeClasse;  
ptrB = new NomeClasse(argumentos);  
ptrC = new NomeClasse[tamanho];
```



# Objetos



## v Alocação dinâmica:

```
nome_da_classe *nome_do_objeto =  
                                new nome_da_classe ( );  
delete ( nome_do_objeto );
```

## v Ponteiro:

- Para uma classe.

```
nome_da_classe *nome_do_ponteiro ;
```

- Para membros de uma classe.

```
tipo nome_da_classe::*nome_do_ponteiro;
```

# Objetos



```
class Aluno {  
    // Lista dos atributos.  
    ...  
    // Lista dos métodos.  
    ...  
};  
  
int main(int argc, char *argv[]) {  
  
    Aluno *alunoA, *alunoB;  
  
    alunoA = new Aluno();  
    alunoB = new Aluno();  
    ...  
}
```



Objetos  
dinamicamente  
criados.

# Objetos



```
string *stringPtr1 = new string;  
string *stringPtr2 = new string[100];
```



```
...
```

```
delete    stringPtr;           // Destrói um objeto.  
delete [] stringPtr2;         // Destrói uma matriz de objetos.
```



```
typedef string SociosFirma[2];
```



```
string *socios = new SociosFirma;
```

```
...
```

```
delete [] socios;
```



# Objetos



## v Exemplo:

```
class Coordenada {  
    public:  
        int x,y;  
    ...  
};
```

...

```
Coordenada coordA;  
int valor;  
int Coordenada::*ptr; // Ponteiro para atributo da classe.  
ptr = &Coordenada::x; // Obtém endereço do atributo.  
coordA.*ptr = 10;      // Acessa o membro.  
valor = coordA.*ptr;    // Acessa o membro.
```



Visibilidade  
pública para  
possibilitar  
acesso.

# Objetos

```
class Coordenada {  
    ...  
    public:  
        void setX(int x){this->x = x;}  
        int getX(){return x;}  
};  
  
Coordenada coordA;  
int valor;  
void (Coordenada::*ptrA)(int);           // Ponteiro para método.  
int (Coordenada::*ptrB)();               // Ponteiro para método.  
ptrA = &Coordenada::setX;                // Obtém endereço.  
ptrB = &Coordenada::getX;                // Obtém endereço.  
(coordA.*ptrA)(10);                     // Invoca via ponteiro.  
valor = (coordA.*ptrB)();                 // Invoca via ponteiro.
```



## ✓ Referência para objeto:

- Similar ao uso de ponteiros.
- Usado o símbolo `&` em vez do símbolo `*`.  

```
Point pt1 ( 10, 10 );  
Point& pt2 = pt1;
```
- Pode-se usar referências como argumentos de métodos.
- Argumento é passado por referência e não por valor.

```
Conjunto operator * (Conjunto& cnj1,  
                    Conjunto& cnj2) {  
    // . . .  
}
```

# Objetos



## ▼ Envio de mensagem:

- Usando o nome do objeto.

```
nome_do_objeto.nome_do_método (param);
```

- Usando ponteiro para o objeto.

```
nome_do_ponteiro->nome_do_método (param);
```

## ▼ Exemplo:

```
ContaCorrente *a = new ContaCorrente ( );  
ContaCorrente b;  
a->sacar ( 100 );  
b.depositar ( 200 );
```

# Construtores e destrutores



## ✓ Método construtor:


- Automaticamente invocado quando objeto é criado.
- Garante que inicialização não será esquecida.
- Mesmo nome da classe da qual é um membro.
- Tipo do retorno é implícito.
- Retorno é ponteiro para uma instância da classe.
- Pode ser sobrecarregado e ter argumentos.

## ✓ Construtor *default*:

- Construtor para o qual não há argumentos .
- Construtor para o qual há *defaults* para todos argumentos.



# Construtores e destrutores



```
class Aluno {  
    // Lista dos atributos.  
    private:  
        string  nome;  
        int     matricula;  
        int     telefone;  
  
    // Lista dos métodos.  
    public:  
        Aluno();  
        Aluno(string, int, int);  
        void setNome(string nome);  
        void setMatricula(int matricula);  
        void setTelefone(int telefone);  
};
```

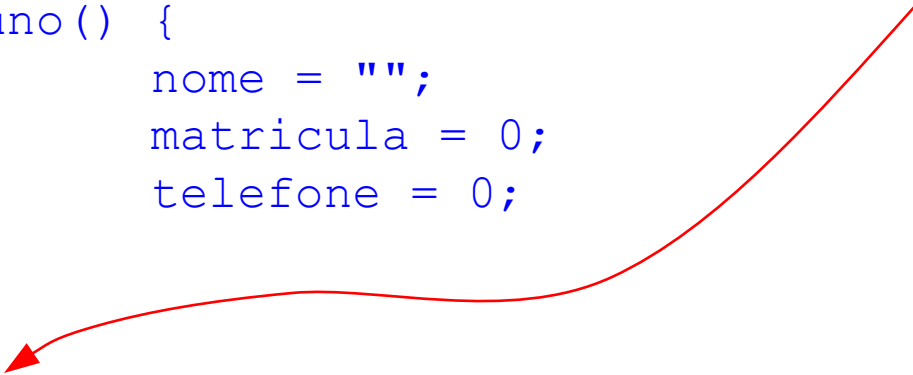
# Construtores e destrutores



// Definições dos construtores.

```
Aluno::Aluno() {  
    nome = "";  
    matricula = 0;  
    telefone = 0;  
}
```

Construtor  
tem o mesmo  
nome da  
classe.



```
Aluno::Aluno(string nome, int matricula, int telefone) {  
    this->nome = nome;  
    this->matricula = matricula;  
    this->telefone = telefone;  
}
```

# Construtores e destrutores

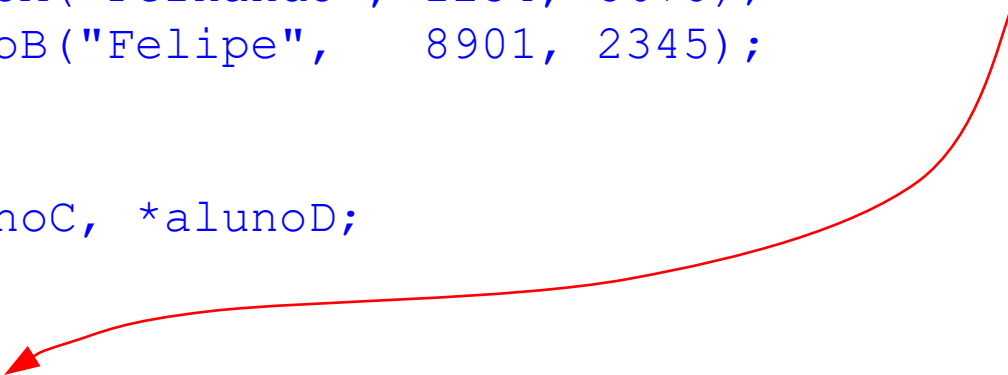


Os objetos são  
inicializados  
quando são  
criados.



```
Aluno alunoA("Fernando", 1234, 5678);  
Aluno alunoB("Felipe", 8901, 2345);
```

```
Aluno *alunoC, *alunoD;
```



```
alunoC = new Aluno("Margarida", 6789, 1234);  
alunoD = new Aluno("Claudia", 5678, 9012);
```

# Construtores e destrutores



## ✓ Lista de inicialização:

- Forma de inicializar os membros.
- Lista separada por vírgulas.

```
class ElementoGrafico {  
    int coordX, coordY;  
public:  
    ElementoGrafico(int, int);  
    ...  
};
```

```
ElementoGrafico::ElementoGrafico(int x, int y):coordX(x), coordY(y){  
}
```

# Construtores e destrutores

```
class Matricula {  
    Aluno &aluno;  
    Disciplina &disciplina;  
public:  
    Matricula(Aluno&, Disciplina&);  
    ...  
};
```

```
Matricula::Matricula(Aluno &aluno,  
    Disciplina &disciplina):  
    aluno(aluno), disciplina(disciplina) {  
}
```

```
Matricula::Matricula(Aluno &aluno, Disciplina &disciplina){  
    this->aluno = aluno;  
    this->disciplina = disciplina;  
}
```

Inicialização de referências deve ser feita em uma lista.

Não é correto inicializar no corpo.

# Construtores e destrutores



## v Método destrutor:

- Programador deve destruir objetos não mais necessários.
- Invocado quando objeto sai do escopo.
- Invocado quando de *delete*.
- Tem o mesmo nome da classe precedido por ~ .
- Não tem argumentos.
- Não pode ser sobrecarregado.

```
class nome_da_classe {  
    public: ~nome_da_classe();  
};
```

# Construtores e destrutores



## ✓ Construtor usado para copiar objetos:

- Possibilita criar cópia de objeto.
- Argumento referencia objeto da classe da qual construtor é membro.
- Construtor provido pelo compilador copia membro a membro, isso causa problemas em certas situações.

## ✓ Quando copiar objetos:

- Ao criar objeto que usa outro objeto para inicialização.
- Em uma atribuição.
- Na passagem de argumentos para uma função.
- No retorno de um objeto por uma função.

# Construtores e destrutores



## ✓ Matriz de objetos:

- Definida tal como uma matriz convencional.
- Podem ser especificados valores para sua inicialização.
- Objetos podem ser criados de forma dinâmica ou não.

```
Aluno graduacao[2]= {Aluno("Maria", 1234, 5678),  
                      Aluno("Roberto", 9012, 3456)};
```

```
Aluno *mestrado = new Aluno [3];
```



Valores iniciais  
não são  
informados.



# Tratamento de exceção



## ✓ Conceito:

- Mecanismo que visa facilitar o tratamento de anomalias ocorridas em tempo de execução.

## ✓ Vantagens:

- Padroniza a forma como erros são notificados.
- Torna explícita a notificação de erro.
- Aproxima a notificação de erro do tratamento do mesmo.
- Melhora a documentação de erro.

# Tratamento de exceção



## ✓ Exemplos de exceções:

- Divisão por zero.
- Argumento inválido.
- *Overflow*.
- Acesso a um elemento fora dos limites de uma matriz.
- Término da memória livre disponível.

# Tratamento de exceção

```
exception
    bad_alloc
    bad_cast
    bad_typeid
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        overflow_error
        range_error
        underflow_error
    ios_base::failure
```



Algumas funções da biblioteca padrão lançam exceções.

A classe base `std::exception` é definida em `<exception>`.

As exceções padrões podem ser lançadas por funções na aplicação ou novas classes podem ser derivadas.

# Tratamento de exceção



## ▼ Tratamento de exceção:

- Bloco *try* - Contém código que pode lançar exceções.
- *throw* - Palavra-chave que lança exceção.
- *catch* - Enunciados que capturam as exceções.

## ▼ Palavra-chave *throw*:

// Lança objeto a ser capturado por valor ou referência.

```
throw EIntegerRange(0, 10, userValue);
```

// Lança objeto a ser capturado por um ponteiro.

```
throw new EIntegerRange(0, 10, userValue);
```

# Tratamento de exceção



## ✓ Capturando e tratando exceção:

```
try {  
    ...  
    if (mystring == NULL) throw "Falha de alocacao";  
    ...  
    if (n > 9) throw n;  
    ...  
}  
catch (int i) { ... }  
catch (char * str) { ... }
```

Two red arrows originate from the right side of the slide. One arrow points towards the closing brace of the try block, and the other points towards the first catch block.

# Tratamento de exceção



- ✓ Captura pode englobar toda uma função:

```
int funcao ()  
try {  
    // ...  
    return 0;  
}  
  
catch ( pushOnFull ) {  
    // ...  
}  
catch ( popOnEmpty ) {  
    // ...  
}
```

O bloco *try* está englobando todo o corpo da função.


A captura está ocorrendo fora do corpo da função.

# Tratamento de exceção

## ✓ Capturando qualquer exceção:

```
try {  
    ...  
}  
catch (...) { ... }
```

Os três pontos  
indicam a captura de  
qualquer classe  
de exceção.



## ✓ Especificando as possíveis exceções:

```
void f1();           // Pode lançar qualquer exceção.  
void f2() throw();   // Não pode lançar exceções.  
void f3() throw( A, B ); // Pode lançar exceções derivadas  
                        // das classes A ou B.
```

# Tratamento de exceção



Classes das exceções que  
podem ser lançadas  
pelos métodos.



```
class iStack {  
public:  
    // ...
```

```
    void pop( int &value ) throw(popOnEmpty);  
    void push( int value ) throw(pushOnFull);
```

```
private:  
    // ...  
};
```



# Tratamento de exceção



## ✓ Aninhando blocos:

- Blocos *try-catch* podem ser aninhados.
- Exceções podem ser relançadas a partir de um *catch*.

```
try {  
    try {  
        ...  
    }  
    catch (int n) {  
        throw;  
    }  
}  
catch (...) {  
    ...  
}
```



Exceção relançada  
via **throw** sem  
operando.

# Tratamento de exceção



## ✓ Exceção não capturada:

- Exceção não capturada chama a função *terminate* .
- A função tipicamente encerra o processo corrente e informa que ocorreu término anormal.

```
void terminate( );
```

## ✓ Lidando com falta de memória:

- *new* retorna ponteiro para *void* em caso de sucesso.
- *new* lança exceção quando não há memória para objeto
- Exceção *bad\_alloc*.

# Tratamento de exceção

- Versões do C++ sem tratamento de exceção retornam 0.
- Pode-se também retornar 0 no C++ padrão.
- Usada a palavra *nothrow*.

```
if (int *ptr = new (nothrow) int[100]){  
  
    // Sucesso na alocação.  
}  
else {  
    // Falha na alocação.  
}
```

# Sobrecarga de operadores



## ✓ Conceitos:

- Possibilita redefinição do comportamento de operadores.
- Uso dos objetos torna-se similar ao dos tipos primitivos.

## ✓ Sobrecarga de operador deve conter:

- Tipo do dado a ser retornado.
- Nome da classe da qual o método é membro.
- Palavra-chave *operator*.
- Símbolo que identifica o operador.
- Tipos e identificadores dos parâmetros.

# Sobrecarga de operadores



## ✓ Número de argumentos:

- 0 se operador for unário e função definida como membro.
- 1 se operador for unário e função definida como amiga.
- 1 se operador binário e função definida como membro.
- 2 se operador binário e função definida como amiga.

## ✓ Operadores que podem ser sobrecarregados:

- Unário `+` `-` `*` `!` `~` `&` `++` `--` `( )` `->` `new` `delete`
- Binário `+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>` `=` `+=`  
`--` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=` `==` `!=`  
`<` `>` `<=` `>=` `&&` `||` `[ ]` `( )`

# Sobrecarga de operadores



## ▼ Restrições quanto à sobrecarga:

- Apenas símbolos existentes podem ser sobrecarregados.
- Não podem ser sobrecarregados `::`, `.*`, `.` e `?:`
- Não podem ser funções globais `=`, `[]`, `()` e `->`
- Operadores `()` `[]` e `->` sobrecarregados como membros.
- Pelo menos um operando é uma classe ou enumeração.
- Não pode-se modificar o número de operandos original.
- Não pode-se mudar a precedência dos operadores.

# Sobrecarga de operadores

```
class Ponto {  
    private:  
        int x, y;  
    public:  
        Ponto operator + (Ponto p);  
        Ponto operator - (Ponto p);  
        ...  
};
```

```
Ponto Ponto::operator + (Ponto p) {  
    return Ponto(x + p.x, y + p.y);  
}
```

```
Ponto Ponto::operator - (Ponto p) {  
    return Ponto(x - p.x, y - p.y);  
}  
...
```

```
Ponto a(1,2), b(3,4), c;
```

```
c = a + b;
```



Uso do operador  
sobrecarregado.



Funções são  
membros  
da classe.

# Sobrecarga de operadores



```
class Ponto {  
    private:  
        int x, y;  
    public:  
        friend Ponto operator + (Ponto, Ponto);  
        friend Ponto operator - (Ponto, Ponto);  
  
    ...  
};
```

```
Ponto operator + (Ponto p1, Ponto p2) {  
    return Ponto(p1.x + p2.x, p1.y + p2.y);  
}
```

```
Ponto operator - (Ponto p1, Ponto p2) {  
    return Ponto(p1.x - p1.x, p1.y - p2.y);  
}
```

...



Funções  
são amigas  
da classe.



# Sobrecarga de operadores

```
class Complexo {
    private:
        float r;
        float i;
    public:
        Complexo operator = (Complexo x);
        Complexo operator + (Complexo x);
};

Complexo Complexo::operator = (Complexo x) {
    r = x.r;
    i = x.i;
    return *this;
}

Complexo Complexo::operator + (Complexo x) {
    Complexo tmp;
    tmp.r = r + x.r;
    tmp.i = i + x.i;
    return tmp;
}
```



Retorno de  
referência  
*this*.



Retorno de  
cópia do  
objeto local.

# Sobrecarga de operadores



## ✓ Sobrecarga de *new* e *delete*:

- Situações onde é necessário controlar a alocação.
- Sobrecarga pode ser relativa a uma classe se a função for membro da classe.
- Sobrecarga pode ser global se não for função membro.

```
void *operator new(long bytes) {  
    ...  
    return ponteiro_para_memória;  
}
```

← Sobrecarga global.

```
void nome_classe::operator delete(void *p)  
    ...  
}
```

← Sobrecarga relativa a uma classe.

# Sobrecarga de operadores



## ✓ Operadores de inserção << e extração >>:

- Preferível esses operadores do que as funções em *stdio.h*.
- Funções *printf* e *scanf* não são extensíveis.
- Funções separam das variáveis as formatações.

```
int i;
```

```
cin >> i;           // Extração.
```

```
cout << i;          // Inserção.
```

# Sobrecarga de operadores



Operador de inserção.



```
class Coordenada {  
    int x, y;  
public:  
    friend ostream& operator << (ostream &str, Coordenada coord);  
    ...  
};  
  
ostream &operator << (ostream &str, Coordenada coord) {  
    str << "Coordenada x: " << coord.x;  
    str << "Coordenada y: " << coord.y;  
}  
...  
cout << coordA;
```

# Herança

## ✓ Conceitos:

- Promove reuso de código.
- Herança pode ser simples ou múltipla.

## ✓ Elementos na declaração de herança:

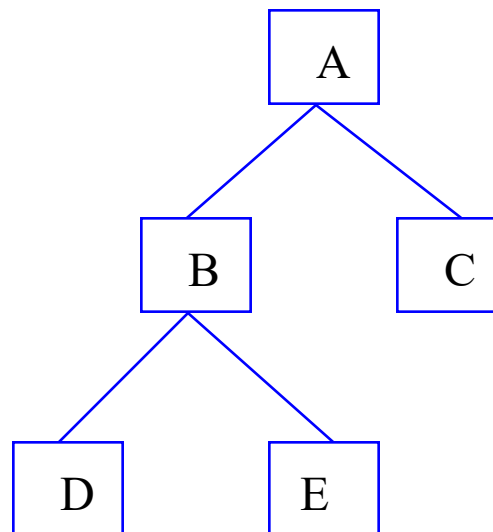
- Palavra-chave *class*.
- Nome da classe.
- Operador *:*
- Nomes das classes herdadas e dos modos de acesso.

```
class nome_classe:modo_acesso nome_classe_base {  
  
};
```

# Herança

## ✓ Herança simples:

```
class nome_da_classe_A { // . . . };  
class nome_da_classe_B:nome_da_classe_A { //... };  
class nome_da_classe_C:nome_da_classe_A { //... };
```



# Herança



## ✓ Modo de acesso:

- No modo *public*, os membros públicos na classe base permanecem públicos na classe derivada e os membros protegidos permanecem protegidos.
- No modo *private*, os membros públicos e protegidos na classe base se tornam privados na classe derivada.
- No modo *protected*, os membros públicos e protegidos podem ser usados somente por membros e amigas da classe derivada e de classes que venham a ser derivadas dessa.

# Herança

- O modo de acesso *default* é *private* na herança entre classes, restringindo automaticamente a visibilidade dos membros nas classes derivadas.
- O modo de acesso *default* é *public* na herança entre *structs*.

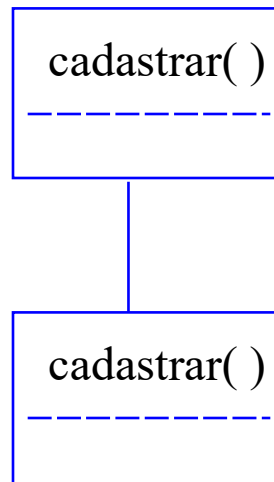
```
class A: public B { ... }  
class C: protected B { ... }  
class D: private B { ... }  
class E: B { ... }           // Privado.
```



# Herança

## ✓ Método sobrescrito (*method overriding*):

- Pode ser acessado a partir das classes derivadas.
- Usado o operador de resolução de escopo.
- Necessário especificar a visibilidade como *protected*.



# Herança

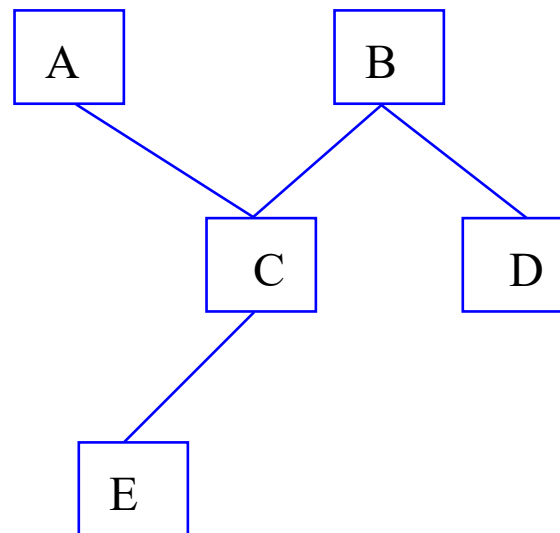


```
class Aluno {  
    ...  
    protected:  
        void cadastrar();  
    ...  
};  
  
class AlunoEspecial:public Aluno {  
    ...  
    public:  
        void cadastrar();  
    ...  
};  
  
void AlunoEspecial::cadastrar() {  
    ...  
    Aluno::cadastrar();  
    ...  
}
```

# Herança

## ✓ Herança múltipla:

```
class nome_da_classe_A { // . . . };  
class nome_da_classe_B { //. . . };  
class nome_da_classe_C:nome_da_classe_A,  
                        nome_da_classe_B {//...};
```



# Herança



## ✓ Ambiguidades na herança múltipla:

- Diferentes classes base tendo membros com mesmo nome.
- Uma mesma classe base mais de uma vez na hierarquia.

## ✓ Solução de ambiguidade:

- Necessário identificar a classe com o membro.
- Especifica-se a classe do membro.
- Usa-se o operador de resolução de escopo.

```
NomeClasseA::NomeMetodo();  
NomeClasseB::NomeMetodo();  
NomeClasseA::NomeAtributo;  
NomeClasseB::NomeAtributo;
```

# Herança



## ✓ Resolução de ambiguidade:

```
class ProfessorOrientador:public Docente, public Funcionario {  
    ...  
};
```

```
class ProfessorCoordenador:public ProfessorOrientador {  
    ...  
    ProfessorOrientador::Docente::cadastrar();  
    Docente::cadastrar();  
    Funcionario::cadastrar();  
    cadastrar();  
    ...  
};
```

Procura em  
ProfessorCoordenador  
depois em  
ProfessorOrientador  
depois em  
Docente  
e depois em Funcionario.

# Herança

## ✓ Herança e declaração *using* :

```
class Docente {
```

```
    ...
```

```
    void cadastrar(int);
```

```
};
```

```
class Funcionario {
```

```
    ...
```

```
    void cadastrar(float);
```

```
};
```

```
class Professor:public Docente, public Funcionario {
```

```
    ...
```

```
    cadastrar(123);
```

```
    ...
```

```
}
```

← As ambiguidades entre métodos de diferentes classes base não são resolvidas com base nos tipos dos argumentos.

← Ambiguidade.

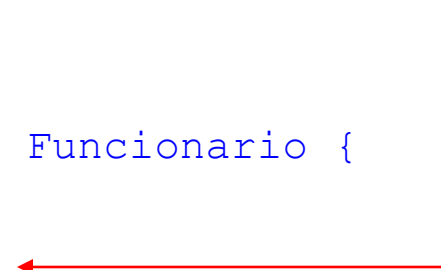
# Herança

```
class Docente {  
    ...  
    void cadastrar(int);  
};
```

```
class Funcionario {  
    ...  
    void cadastrar(float);  
};
```

```
class Professor:public Docente, public Funcionario {  
    ...  
    using Docente::cadastrar;  
    using Funcionario::cadastrar;  
    ...  
    cadastrar(123); // Invoca Docente::cadastrar  
    cadastrar(1.2); // Invoca Funcionario::cadastrar  
    ...  
}
```

Declarações *using*  
trazem os métodos  
para o mesmo escopo  
possibilitando a  
resolução a partir dos tipos  
dos argumentos.



# Herança



## ✓ Replicação de classes base:

```
class Profissional {  
    Profissional *primeiro;  
    ...  
};  
class Docente:public Profissional {  
    ...  
};  
class Funcionario:public Profissional {  
    ...  
    void cadastrar(float);  
};  
class Professor:public Docente, public Funcionario {  
    ...  
}
```

Instâncias da  
classe **Professor**  
terão duas  
listas de  
profissionais.





# Herança

```
class Profissional {  
    Profissional *primeiro;  
    ...  
};
```

```
class Docente:public virtual Profissional {  
    ...  
};
```

```
class Funcionario:public virtual Profissional {  
    ...  
    void cadastrar(float);  
};
```

```
class Professor:public Docente, public Funcionario {  
    ...  
}
```

As instâncias da classe **Professor** terão uma lista de profissionais devido ao uso de classes base virtuais.



## ✓ Construtores e destrutores:

- Podem existir construtores e destrutores em uma classe base e nas suas derivadas.
- Construtor na classe base é invocado primeiro.
- Destrutor de uma classe derivada é executado antes do destrutor da sua classe base.
- Se um construtor de uma classe base requer argumentos, devem ser providos pela classe derivada.
- Se houver herança múltipla, construtores são invocados seguindo-se a ordem da esquerda para a direita e destrutores da direita para a esquerda.

# Herança

```
class ElementoGrafico {  
    ...  
    public:  
        ElementoGrafico(int, int);  
        ~ElementoGrafico();  
};
```

```
class Retangulo:public ElementoGrafico {  
    ...  
    public:  
        Retangulo(int, int, int);  
        ~Retangulo();  
};
```

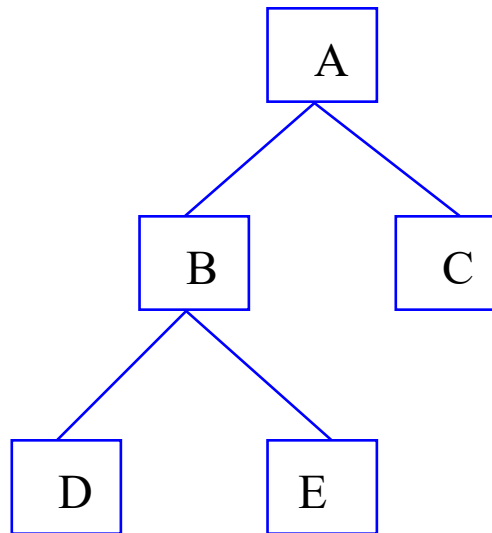
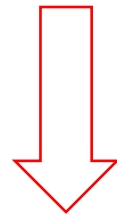
Construtor que requer  
parâmetros sendo  
explicitamente  
invocado.



```
Retangulo::Retangulo(int x, int y, int z):ElementoGrafico(x,y) {  
    ...  
}
```

# Herança

Ordem de  
execução  
dos métodos  
construtores.



Ordem de  
execução  
dos métodos  
destrutores.

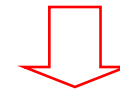
# Herança



```
class ElementoGrafico {  
  
    public:  
        ElementoGrafico();  
        ~ElementoGrafico();  
  
    ...  
};  
...
```

```
class Retangulo:public ElementoGrafico {  
    public:  
        Retangulo();  
        ~Retangulo();  
  
    ...  
};  
...
```

```
{  
    ...  
    ElementoGrafico elemento;  
    Retangulo retangulo;  
    ...  
}
```



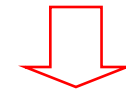
```
    ElementoGrafico()  
    ~ElementoGrafico()  
    ...  
    ElementoGrafico()  
    Retangulo()  
    ~Retangulo()  
    ~ElementoGrafico()
```

# Herança



```
class ElementoGrafico {  
  
    public:  
        ElementoGrafico();  
        ~ElementoGrafico();  
  
    ...  
};  
  
...  
  
class Retangulo:public ElementoGrafico {  
    public:  
        Retangulo();  
        ~Retangulo();  
  
    ...  
};  
  
...
```

```
ElementoGrafico *ptrA;  
...  
ptrA = new ElementoGrafico();  
delete ptrA;  
...  
Retangulo *ptrB;  
ptrB = new Retangulo();  
delete ptrB;  
...  
  
    ElementoGrafico()  
    ~ElementoGrafico()  
    ...  
    ElementoGrafico()  
    Retangulo()  
    ~Retangulo()  
    ~ElementoGrafico()
```





## ✓ Método virtual:

- Definido em uma classe.
- Pode se redefinido em classes derivadas.
- Identificado pela palavra-chave *virtual*.
- Novas declarações são feitas nas classes derivadas.
- Pode facilitar a evolução futura.
- Possibilita a especificação de interfaces.

## ✓ Classe sem métodos virtuais:

- Classe projetada para não ser classe base em hierarquias.

# Herança



Declaração de  
método  
virtual.



```
class Lista {  
    public: virtual Bool inserir (int);  
    ...  
};
```

```
class Conjunto: Lista {  
    public: Bool inserir (int);  
    ...  
};
```



Presença do  
método em  
subclasse.





## ✓ Construtores e destrutores:

- Métodos construtores não podem ser métodos virtuais.
- Métodos destrutores podem ser virtuais.

## ✓ Destrutores não virtuais e virtuais:

- Quando é destruído um objeto de uma classe derivada referenciado por um ponteiro para a classe base, o resultado é indefinido se o destrutor na classe base for não virtual. Tipicamente o destrutor da classe derivada não é invocado.

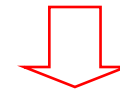
# Herança



```
class ElementoGrafico {  
  
    public:  
        ElementoGrafico();  
        ~ElementoGrafico();  
  
    ...  
};  
  
...
```

```
class Retangulo:public ElementoGrafico {  
    public:  
        Retangulo();  
        ~Retangulo();  
  
    ...  
};  
  
...
```


```
ElementoGrafico *ptr;  
...  
ptr = new ElementoGrafico();  
delete ptr;  
...  
ptr = new Retangulo();  
delete ptr;
```



```
    ElementoGrafico()  
    ~ElementoGrafico()  
    ...  
    ElementoGrafico()  
    Retangulo()  
    ~ElementoGrafico()
```

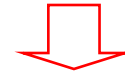
# Herança

```
class ElementoGrafico {  
  
    public:  
        ElementoGrafico();  
        virtual ~ElementoGrafico();  
...  
};  
...
```



```
class Retangulo:public ElementoGrafico {  
    public:  
        Retangulo();  
        ~Retangulo();  
...  
};  
...
```

```
ElementoGrafico *ptr;  
...  
ptr = new ElementoGrafico();  
delete ptr;  
...  
ptr = new Retangulo();  
delete ptr;
```



```
    ElementoGrafico()  
    ~ElementoGrafico()  
  
    ElementoGrafico()  
    Retangulo()  
    ~Retangulo()  
    ~ElementoGrafico()
```

# Herança



## ✓ Polimorfismo em tempo de execução:

- Método a ser invocado identificado quando da execução.
- Possível via métodos virtuais.
- Objetos devem ser referenciados via ponteiros.
- Ponteiro deve ser para uma classe base.

# Herança

```
class ElementoGrafico {
```

```
    public:
```

```
    virtual void desenhar();
```

```
    void apagar();
```

```
};
```

```
...
```

```
class Retangulo:public ElementoGrafico {
```

```
    public:
```

```
    void desenhar();
```

```
    void apagar();
```

```
};
```

```
...
```

```
ElementoGrafico *ptr;
```

```
ptr = new ElementoGrafico();
```

```
ptr->desenhar();
```

```
ptr->apagar();
```

```
...
```

```
ptr = new Retangulo();
```

```
ptr->desenhar();
```

```
ptr->apagar();
```



## ✓ Método virtual puro:

- Facilita a padronização das interfaces em uma hierarquia.
- Se não for definido, é um método virtual puro.

## ✓ Classe abstrata:

- Classe com um ou mais métodos virtuais puros.
- Não é possível instanciar uma classe abstrata.

# Herança



```
class ElementoGrafico {  
    public:  
        virtual void desenhar ( ) = 0;  
    ...  
};  
  
class Circulo: ElementoGrafico {  
    public:  
        void desenhar ( );  
    ...  
};
```

Classe  
abstrata.

Método  
virtual puro.

Classe onde o  
método é  
definido.



- ✓ Classe protocolo (*protocol class*):
  - Classe abstrata especial.
  - Não contém implementação.
  - Tipicamente usada para especificar interface.
  
- ✓ Elementos em uma classe protocolo:
  - Não contém atributos.
  - Não contém construtores.
  - Contém um método destrutor virtual.
  - Contém um conjunto de métodos virtuais puros.



# Herança

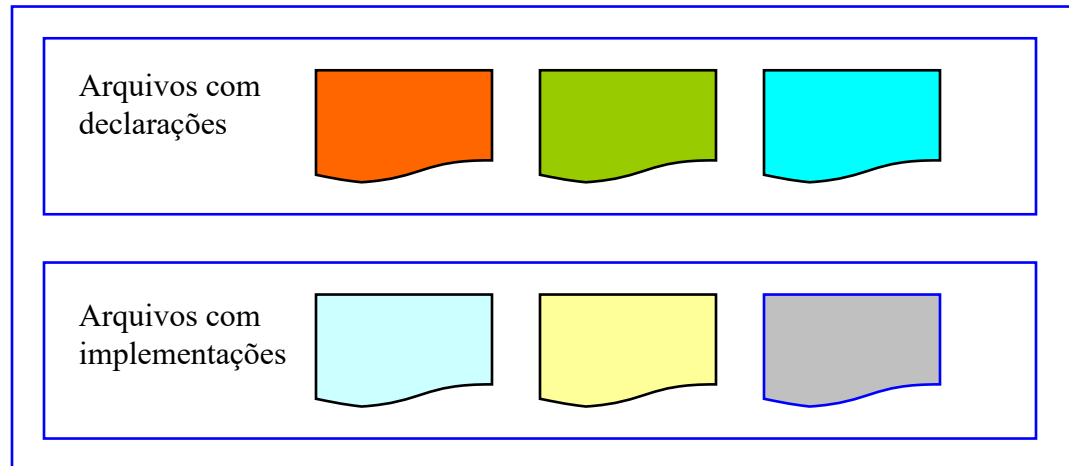
```
class CntrPersistencia {  
public:  
    virtual ~CntrPersistencia(){};  
    virtual void conectar() = 0;  
    virtual void desconectar() = 0;  
    virtual void executar() = 0;  
};  
...  
CntrPersistencia* cntr;  
cntr = criarCntrPersistencia();  
...  
cntr->conectar();  
...  
delete cntr;
```

← Classe protocolo  
com um conjunto  
de métodos virtuais  
puros.

← Clientes de classes  
protocolos são  
programados em  
termos de ponteiros  
e referências para  
a classe.

# Modularização

- ✓ Arquivos com código integrante de módulo:
  - Arquivos com declarações.
  - Arquivos com implementações.



# Modularização



## ✓ Arquivo com declarações:

- Declarações de tipos.
- Declarações de constantes.
- Declarações de variáveis.
- Protótipos para funções integrantes da interface.
- Definições de tipos relevantes ao módulo.

# Modularização



## ✓ Arquivo com implementações:

- Código com a implementação da lógica.
- Definições de tipos usados na implementação.
- Corpos de funções públicas e privadas.
- Deve incluir ( *#include* ) arquivo com declarações.

# Modularização



## ▼ Módulo cliente:

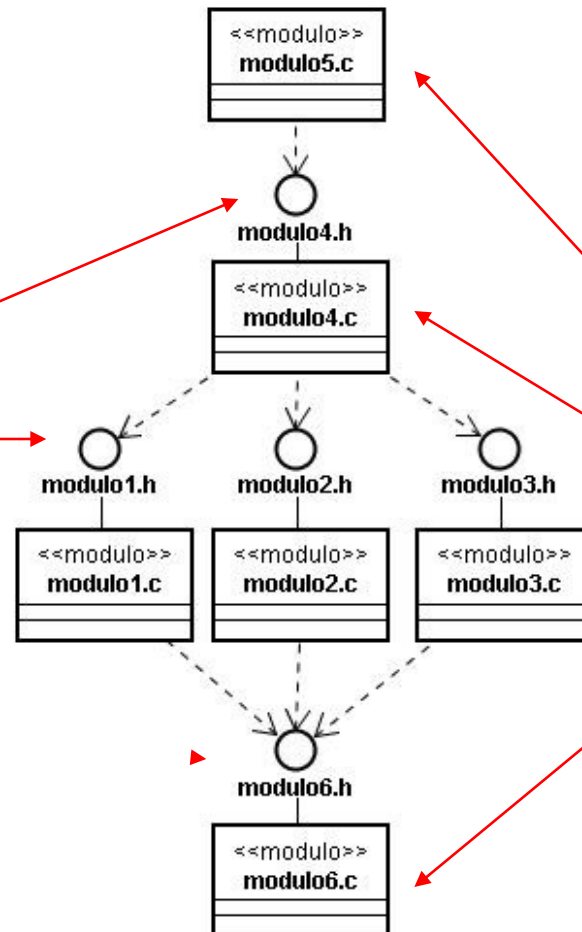
- Inclusão de arquivos com as declarações necessárias.

```
#include "Sets.h"  
  
#define SETSIZE 70  
  
void main ( void ) {  
    ...  
}
```

Inclusão de arquivo com  
declarações.

# Modularização

Arquivos com  
declarações.



Arquivos com  
implementações.



# Encerramento

# Encerramento



## ✓ Exemplos de requisitos profissionais:

- Graduação em Ciência da Computação ou em curso correlato.
- Experiência em engenharia de software e em processos de software.
- Experiência em projeto (*design*) e desenvolvimento orientados a objetos.
- Habilidade de codificação e experiência em linguagens relevantes.
- Experiência no uso de ferramentas de desenvolvimento.
- Habilidade de comunicação escrita e comunicação verbal.
- Habilidade interpessoal e na construção de equipes.
- Criatividade na solução de problemas.
- Foco em detalhes.
- Experiência em gerenciar ciclo de vida de desenvolvimento de software.
- Habilidade de priorizar atividades.
- Habilidade de realizar projetos dentro de prazos e custos definidos.



# Encerramento



## ✓ Exemplos de responsabilidades:

- Realizar captura e análise de requisitos de software.
- Liderar e coordenar ciclo de desenvolvimento de software.
- Planejar e monitorar esforços de desenvolvimento.
- Participar de revisões de software.
- Realizar manutenção de software.
- Determinar necessidade de reestruturação de software.
- Definir padrões de modelos a serem adotados na organização.
- Definir padrões de codificação a serem adotados na organização.
- Disseminar princípios de arquitetura de software na organização.
- Desenvolver documentação de software.
- Gerenciar documentação de software.

# Encerramento



## ✓ Exemplos de tecnologias demandadas:

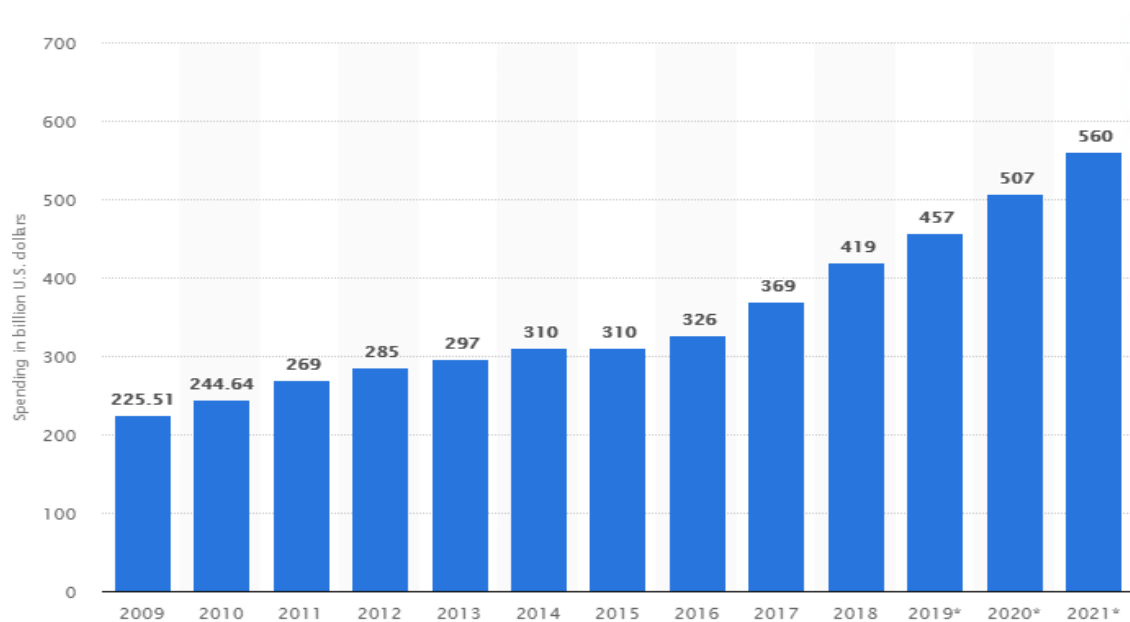
- Objective-C, C, C++, C#, Python, SQL, Java, Visual Basic.NET.
- .NET, ASP.NET, SQL Server, HTML, CSS.
- AJAX
- Visual Studio
- Linux.
- Windows.
- REST API
- Git & Github
- SCRUM
- MVC
- Cloud Based Solutions/Technologies (AWS, Google, Azure).

# Encerramento



## ✓ Gastos globais em software empresarial:

- Sistemas de software são relevantes em diversos contextos.
- Bilhões de dólares americanos são anualmente gastos em software.





## ▼ Fontes de informação:

- BOOCH, G. et al. *Object-Oriented Analysis and Design with Applications*, 3<sup>rd</sup> ed. Addison-Wesley Professional, 2007.
- DEITEL, H. & DEITEL, P. *C++ How to Program*. Pearson International. 10<sup>o</sup> ed. 2017.
- PRATA, S. *C++ Primer Plus*. 6<sup>th</sup> edition. Addison-Wesley Professional, 2011.
- STROUSTRUP, B. *A tour of C++*, 2<sup>nd</sup> edition. Addison-Wesley Professional, 2018.
- STROUSTRUP, B. *Programming: Principles and Practice Using C++*, 2<sup>nd</sup> edition. Addison-Wesley Professional, 2014.
- STROUSTRUP, B. *The C++ Programming Language*, 4<sup>th</sup> edition. Addison-Wesley Professional, 2013.
- WEISFELD, M. *The Object-Oriented Thought Process*. 5<sup>th</sup> ed. Addison-Wesley, 2019.