Conceitos Iniciais

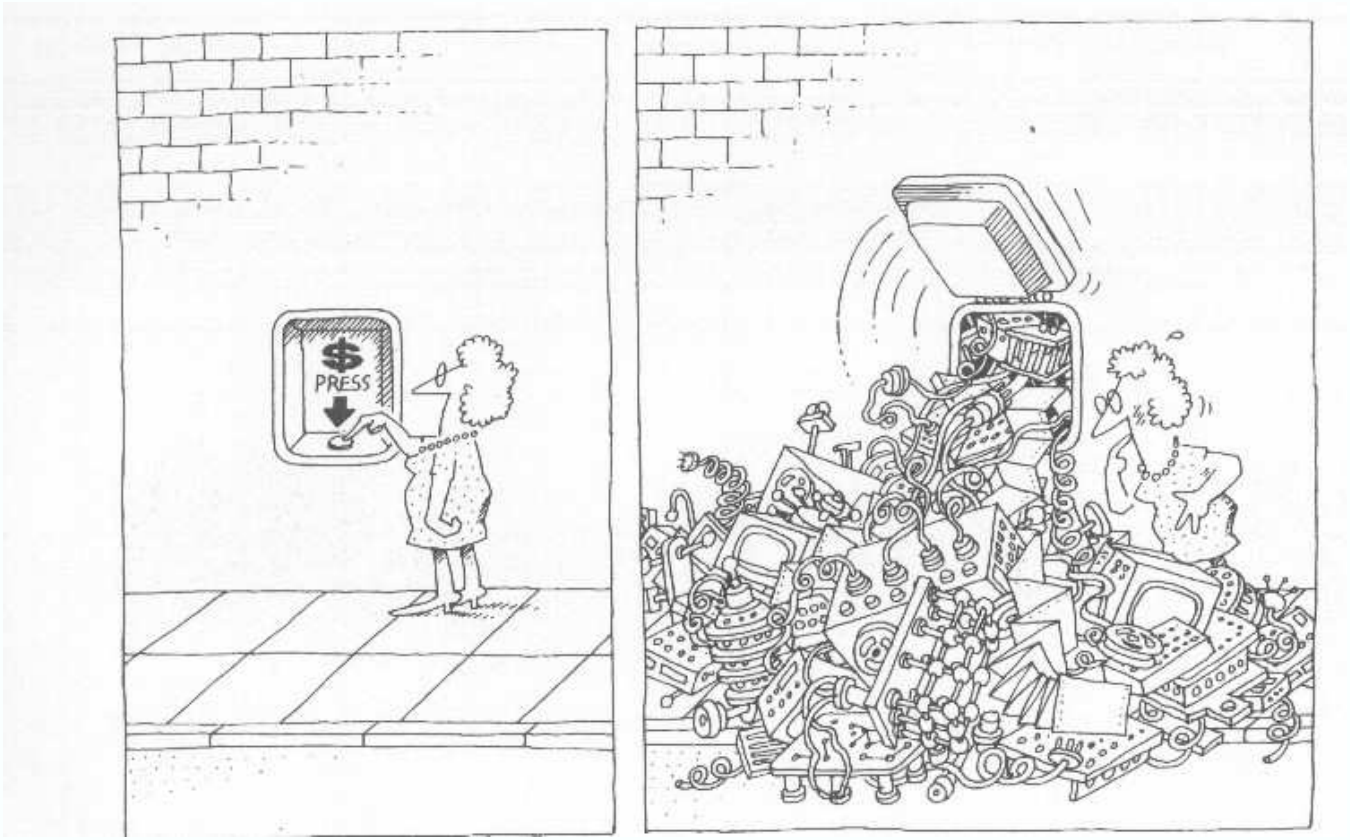
Introdução

Gerenciamento da Complexidade do Software (I)

- Sistemas de software são intrinsicamente complicados
- Os requisitos modernos tendem a complicá-lo cada vez mais:
  - Alta confiabilidade; alto desempenho; desenvolvimento rápido e barato
- São necessários mecanismos de gerenciamento da complexidade:
  - Abstração; generalização; agregação

Gerenciamento da Complexidade do Software (II)

- O sistema deve passar uma "ilusão" de simplicidade:



Crise de software X Orientação a Objetos

Crise de software X Orientação a Objetos

- Crise de Software -1968, Conferência OTAN na Europa

Problemas com o Desenvolvimento de Software	Benefícios do Modelo de Objetos
• pouco predizível, • qualidade baixa dos programas, • custo alto de manutenção, • duplicação de esforços.	• abstração de dados, • flexibilidade, • reutilização, • manutenção.

Modelo de Objetos

Idéias Básicas

- Representa o sistema por meio de elementos do mundo real
  - Entidades físicas (aviões, robôs, etc.)
  - Entidades abstratas (listas, pilhas, filas, etc.)
- Unificação dos conceitos de dados e funções (encapsulamento)

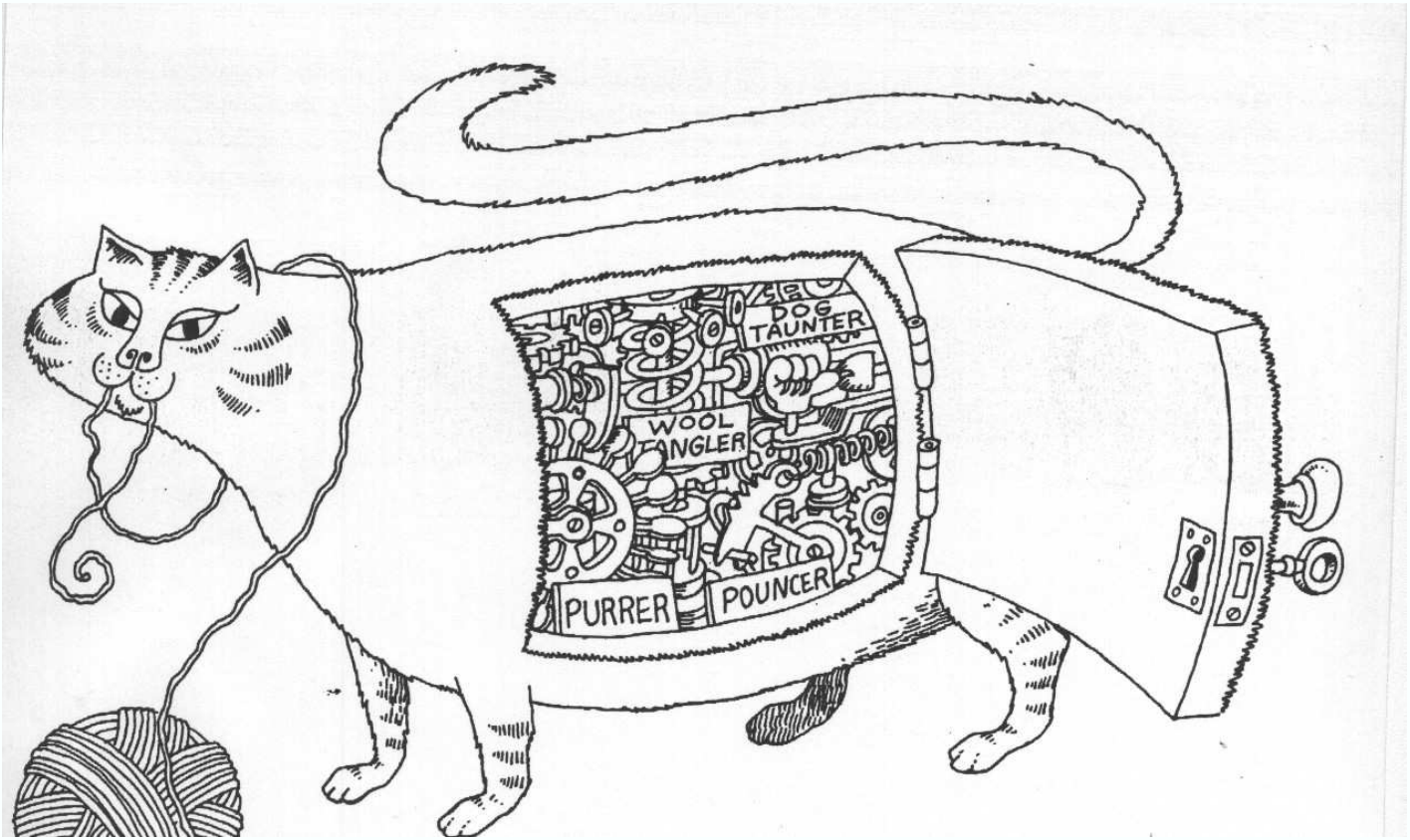
Fundamentos do Modelo de Objetos

- Encapsulamento
- Modularidade
- Abstração de dados
- Hierarquia de Abstração

Encapsulamento (I)

Encapsulamento é o processo de esconder todos os detalhes de um objeto que não contribuem para suas características essenciais.

Encapsulamento (II)

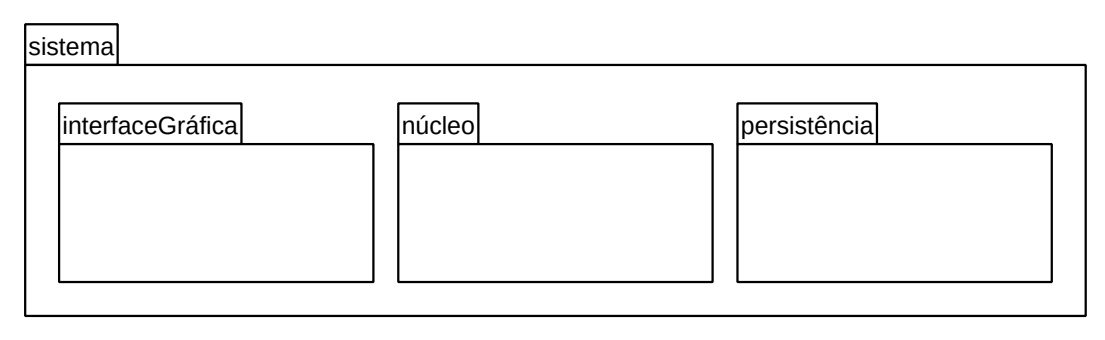


Modularidade (I)

Modularidade é a propriedade de um sistema que foi decomposto num conjunto de módulos altamente coesos e fracamente acoplados

Modularidade (II)

Pacotes

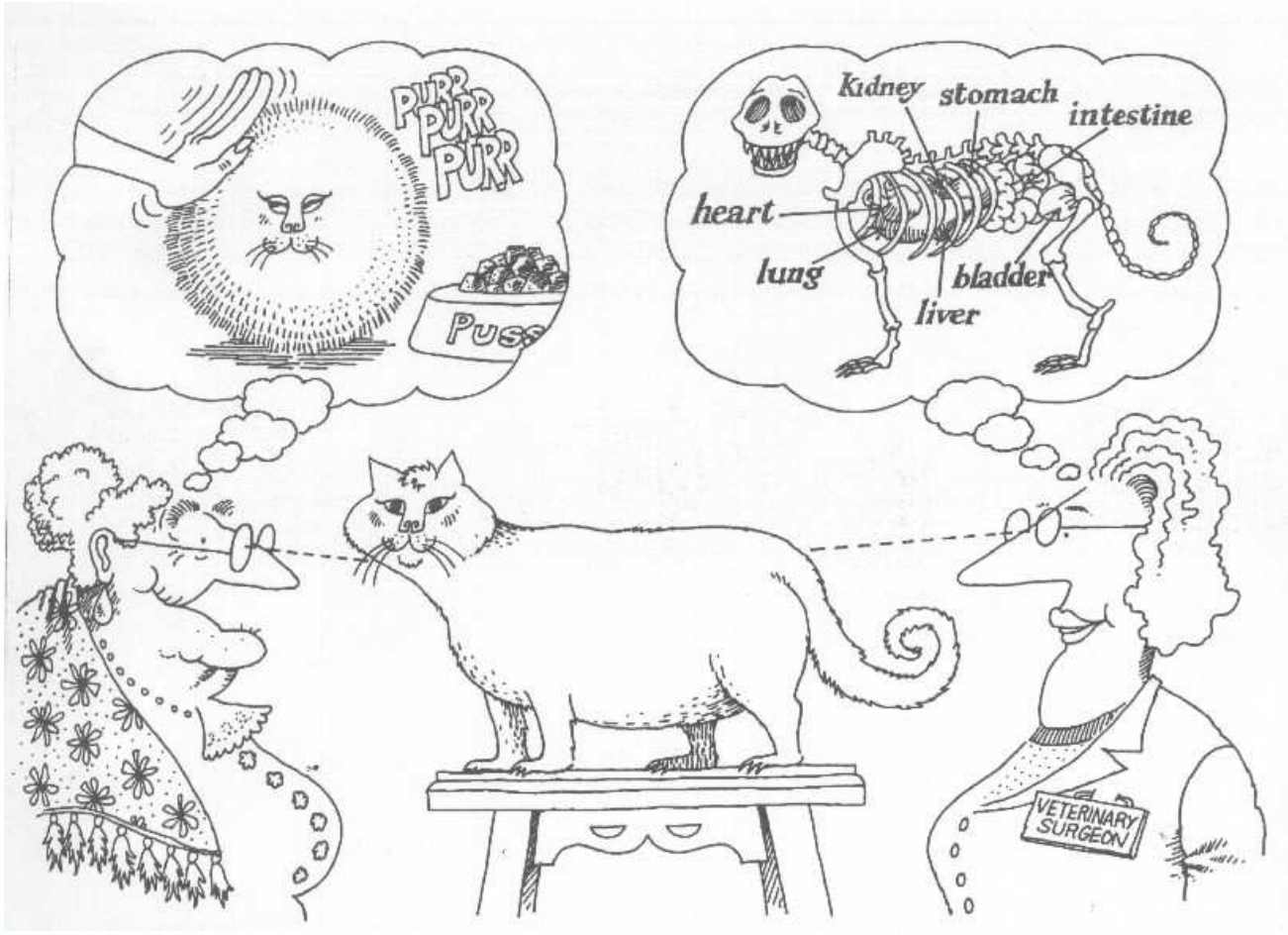


- É um mecanismo para organizar elementos hierarquicamente em grupos
- Um pacote pode agrupar classes, casos de uso, representações dinâmicas, ou outros pacotes

Abstração de Dados (I)

Uma abstração descreve as características essenciais de um objeto que o distingüe de todos os outros tipos de objetos, e portanto proporciona limites conceituais bem definidos, relativo à perspectiva do observador.

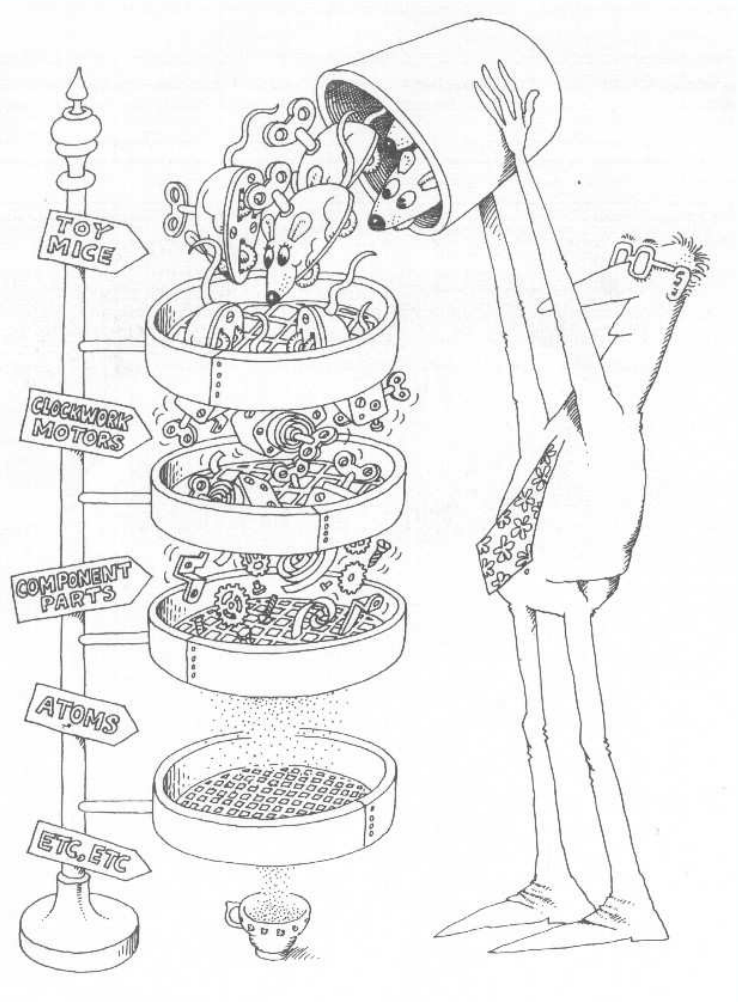
Abstração de Dados (II)



Hierarquia de Abstração (I)

- Quando um sistema tem muitos detalhes para serem descritos por meio de uma única abstração, ele pode ser decomposto numa hierarquia de abstrações
- Uma hierarquia permite que detalhes relevantes sejam introduzidos de uma maneira controlada
- Hierarquia é um "ranking" ou uma ordenação de abstrações
- Duas categorias de hierarquias de abstrações importantes:
  1. Hierarquia de agregação/decomposição
  2. Hierarquia de generalização/especialização

Hierarquia de Abstração (II)



Orientação a Objetos

- **Programação Orientada a Objetos** (POO) é um modelo de programação baseado em conceitos, tais como, objetos, classes, encapsulamento, ocultamento da informação ("information hiding"), herança, polimorfismo, etc.
- **Projeto Orientado a Objetos** ("Object-Oriented Design") é um modo de usar esses conceitos para estruturar e construir sistemas.

Pontos Importantes da Orientação a Objetos

- Obtenção de componentes de software reutilizáveis e flexivéis,
- O paradigma de Objetos é **ortogonal** a outros paradigmas de programação.
- Os conceitos de herança e acoplamento dinâmico ("dynamic binding") são específicos do modelo de objetos.

Limitações do Passado do Modelo de Objetos

- Foco na programação OO
- Trabalho em grupo pouco desenvolvido
- Métodos tradicionais eram inapropriados
- Necessidade de maneiras novas para gerência
- Baixa granularidade de reutilização (objeto) => Componentes de Software

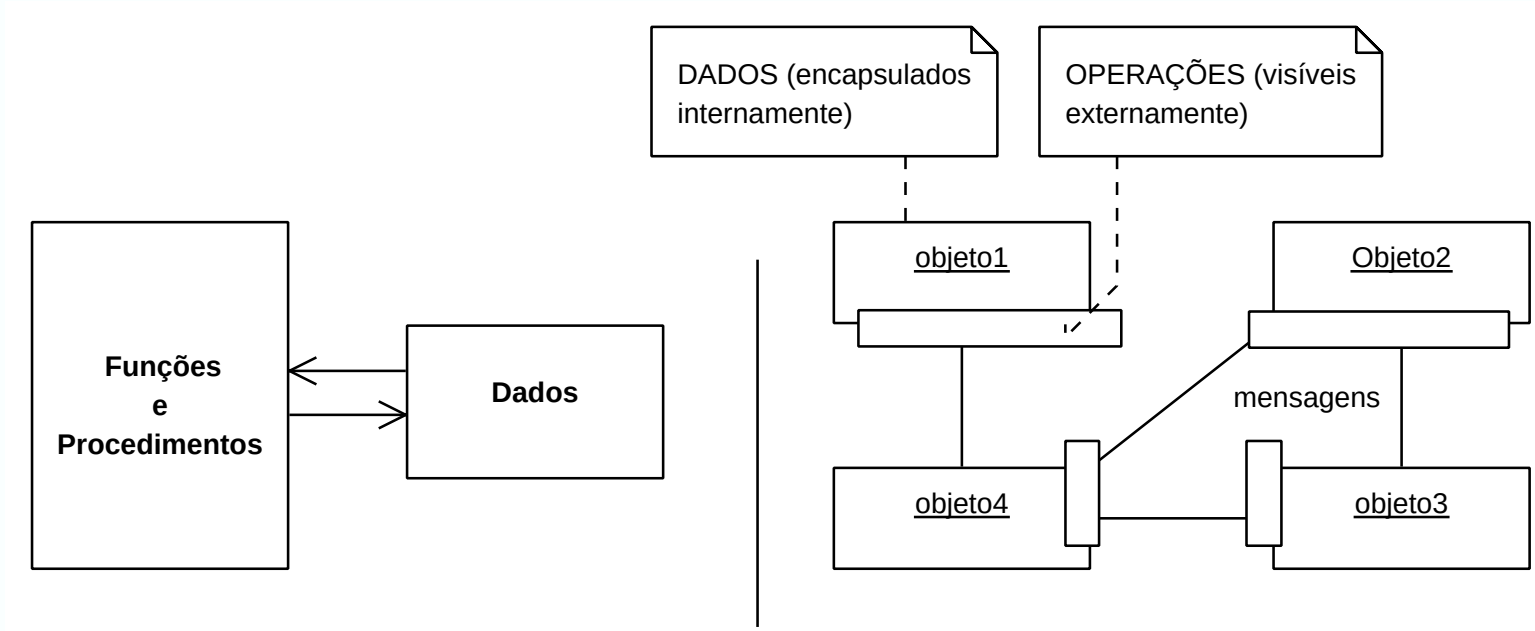
Breve História de Orientação a Objetos

1967	Simula – 67 (Kristen Nygaard e Ole-Johan Dahl)
1970	Pascal (Niklaus Wirth)
1972	Artigo de Dahl sobre ocultamento da informação
1976	Primeira versão de Smalltalk
	(Alan Kay, Dan Ingalls e Adele Goldberg)
1983	Primeira versão de C++ (Bjarne Stroustrup)
1988	Primeira versão de Eiffel (Bertrand Meyer)
1995	Primeira versão de Java
	(Patrick Naughton, Mike Sheridan, e James Gosling)

Paradigma Procedural vs. POO

Procedural	POO
• tipos de dados	• classes/TAD
• variável	• objeto/instância
• função/procedimento	• operação/serviço
• chamada de função	• passagem de mensagem

Paradigma Estruturado vs. POO



Uma Aplicação Estruturada

Uma Aplicação Orientada a Objetos

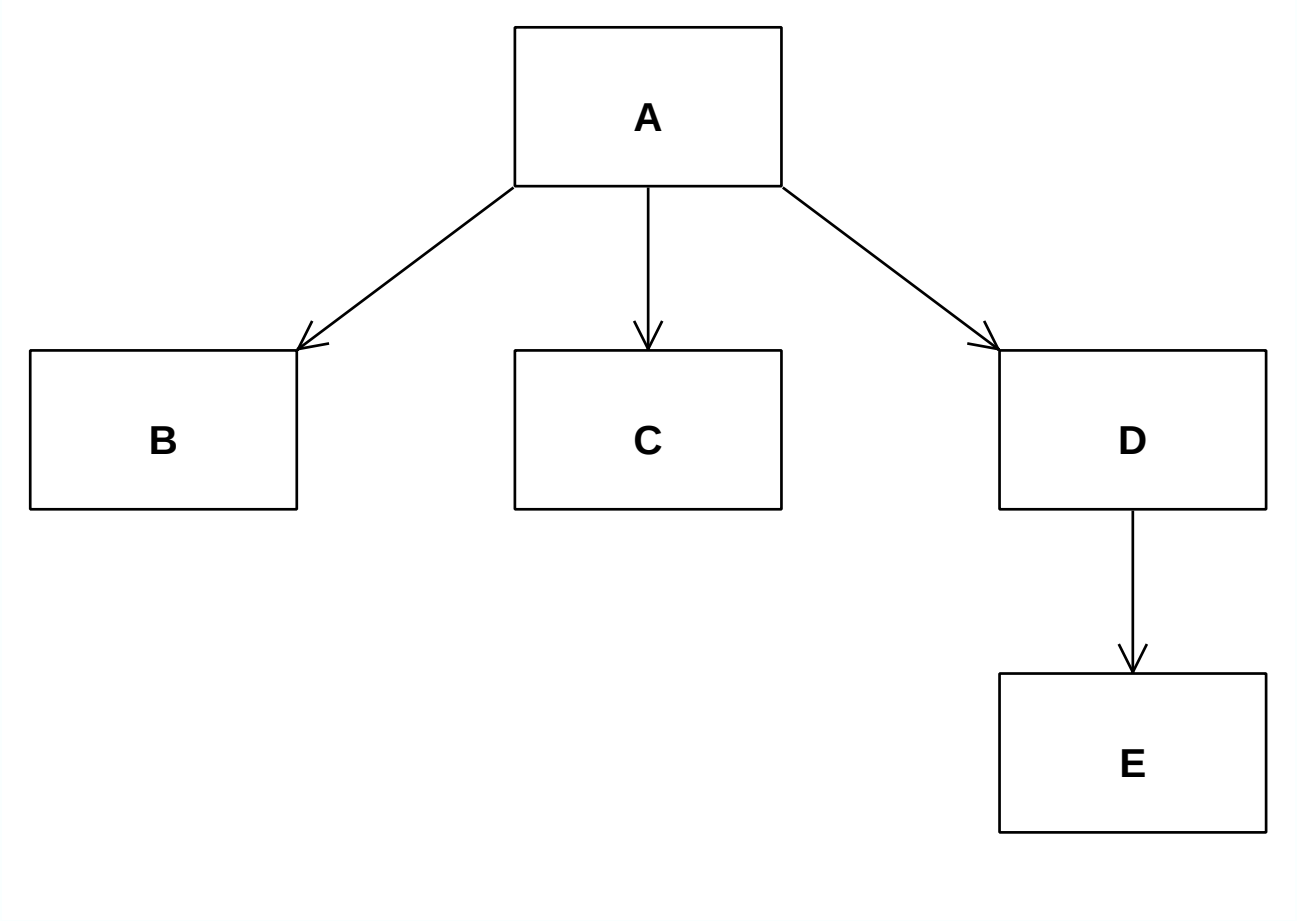
Programa Pascal (I)

```
program P;
const MAX = 100;
type {declarações de tipos}
 TipoT1 = ...;
 TipoT2 = ...;
procedure E(...);
 {variaveis locais}
 begin
...
 end;
procedure C(...);
 begin
  ...
 end;
```

Programa Pascal (II)

```
procedure B(...);
 begin
...
 end;
procedure D(...);
 begin
   ...
 end;
procedure A(...); begin ... end;
var i, j: integer; {variaveis globais}
begin {corpo principal}
{chamadas para procedimentos}
A();
end.
```

Programa Pascal (III)



Programa C (I)

```
/* arquivo1.c */
#include <stdio.h>
int x; /* variavel global */
void setGlobal( int _x);
int main(){
  x = 2;
  setGlobal(3);
  printf("O valor de global é: %d\n", x);
  return 0;}

/* arquivo2.c */
#include <stdio.h>
extern int x;
void setGlobal( int _x){
   x = _x;}
```

Programa C (II)

```
/* arquivo pilha.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct lista{
   int valor;
   struct lista *prox;
}Lista;

typedef struct pilha{
   Lista *prim;
}Pilha;

Pilha* pilha_cria(void){
   Pilha* p = (Pilha*)malloc(sizeof(Pilha));
```

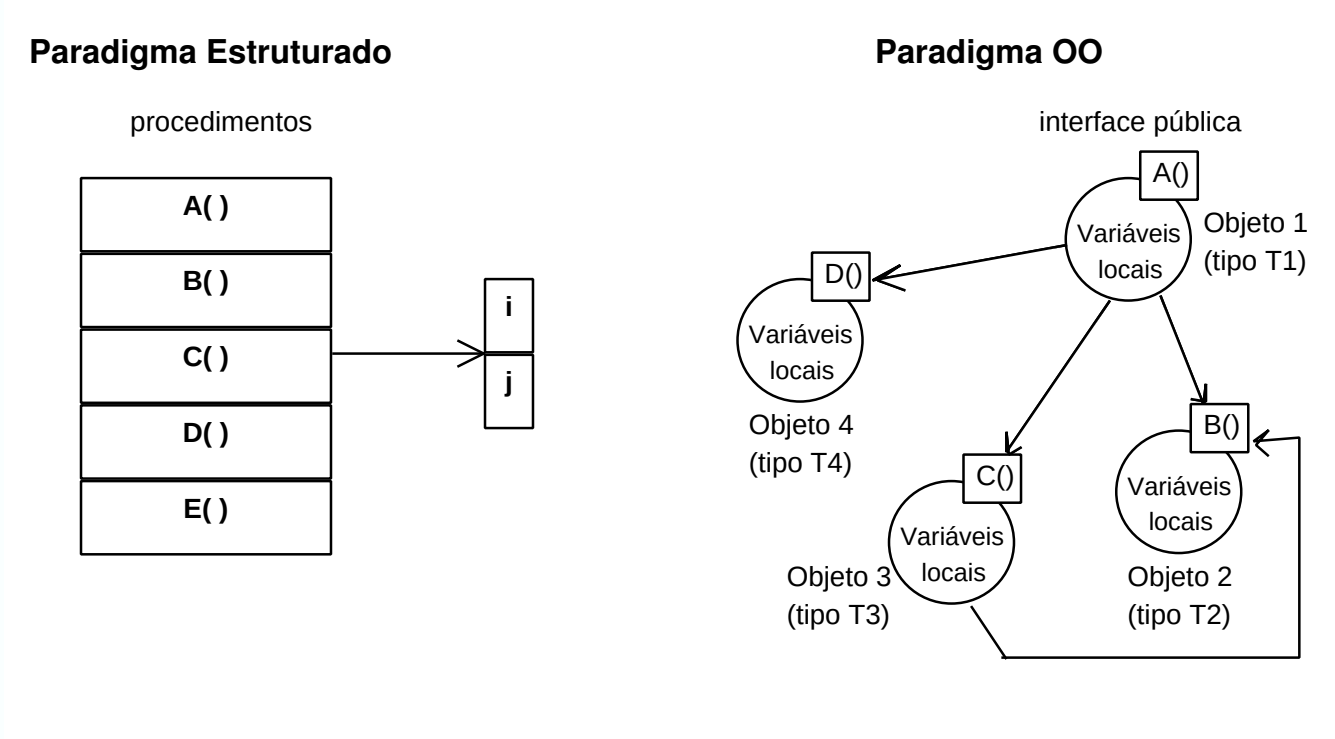
```
   p->prim = NULL;
   return p; }

void pilha_push(Pilha *p, int v){
 Lista *n = (Lista*)malloc(sizeof(Pilha));
 n->valor = v;
 n->prox = p->prim;
 p->prim = n; }

int pilha_vazia(Pilha *p){
  return (p->prim==NULL); }

int  pilha_pop(Pilha *p){
   Lista *t;
   int v;
   if(pilha_vazia(p)){
```

```
      printf("Pilha Vazia.\n");
      exit(1);
   }
   t = p->prim;
   v = t->valor;
   p->prim = t->prox;
   free(t);
   return v; }

void pilha_imprime(Pilha *p){
     Lista *q;
     for(q = p->prim ; q!=NULL; q = q->prox)
        printf("%d\n", q->valor); }

void pilha_libera(Pilha *p){
     Lista * q = p->prim;
```

```
      while(q!=NULL){
         Lista *t = q->prox;
         free(q);
         q=t;
      }
     free(p);
}
int main(){
int valor;
char opcao;
Pilha *pilha;
pilha = pilha_cria();
 do {
   scanf("%c", &opcao);
   switch(opcao) {
     case 'E':
```

```
        printf("Digite um valor a ser empilhado:\n");
        scanf("%d", &valor);
        pilha_push(pilha, valor);
        printf("Foi Empilhado o valor %d no topo\n", valor);
        break;
      case 'R':
printf("Removido o valor %d do topo\n",pilha_pop(pilha));
            break;
      case 'I':
             pilha_imprime(pilha);
            break;
   }
} while (opcao != 'S');
 return 0; }
```

Programa Estruturado vs. Paradigma OO

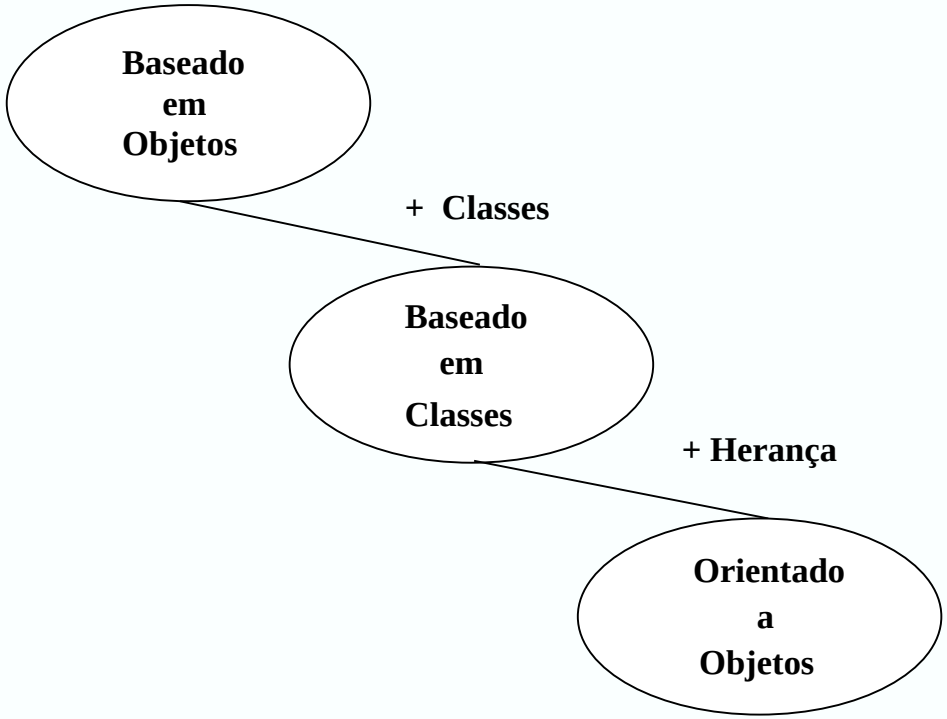


Ling. OO vs. Ling. Baseadas em Objetos

- **Linguagem Baseada em Objetos** ("Object-based Language"): é aquela linguagem baseada somente no conceito de objetos. Ex.: Ada 85 e CLU.
- **Linguagem Orientada a Objetos**: é aquela que proporciona suporte lingüistico para objetos, classes, e um mecanismo de herança. Ex.: C++, Smalltalk, Modula-3, Self, Actor, Ada93, Java.

$$\textbf{POO} = \textbf{Objetos} + \textbf{Classes} + \textbf{Herança}$$

Classificação de Wegner



Desenvolvimento de Software (I)

Etapas do processo de desenvolvimento de software:

1. especificação do problema,
2. entendimento dos requisitos,
3. planejamento da solução, e
4. implementação de um programa numa dada linguagem de programação

Desenvolvimento de Software (II)

- Uma metodologia de projeto consiste em construir um **modelo** do domínio de um problema e então adicionar detalhes de implementação ao modelo durante o projeto do sistema.
- A abordagem orientada a objetos para construção de sistemas permite que um mesmo conjunto de conceitos e uma mesma notação sejam usados durante todo o ciclo de vida do software: **análise, projeto** e **implementação**.

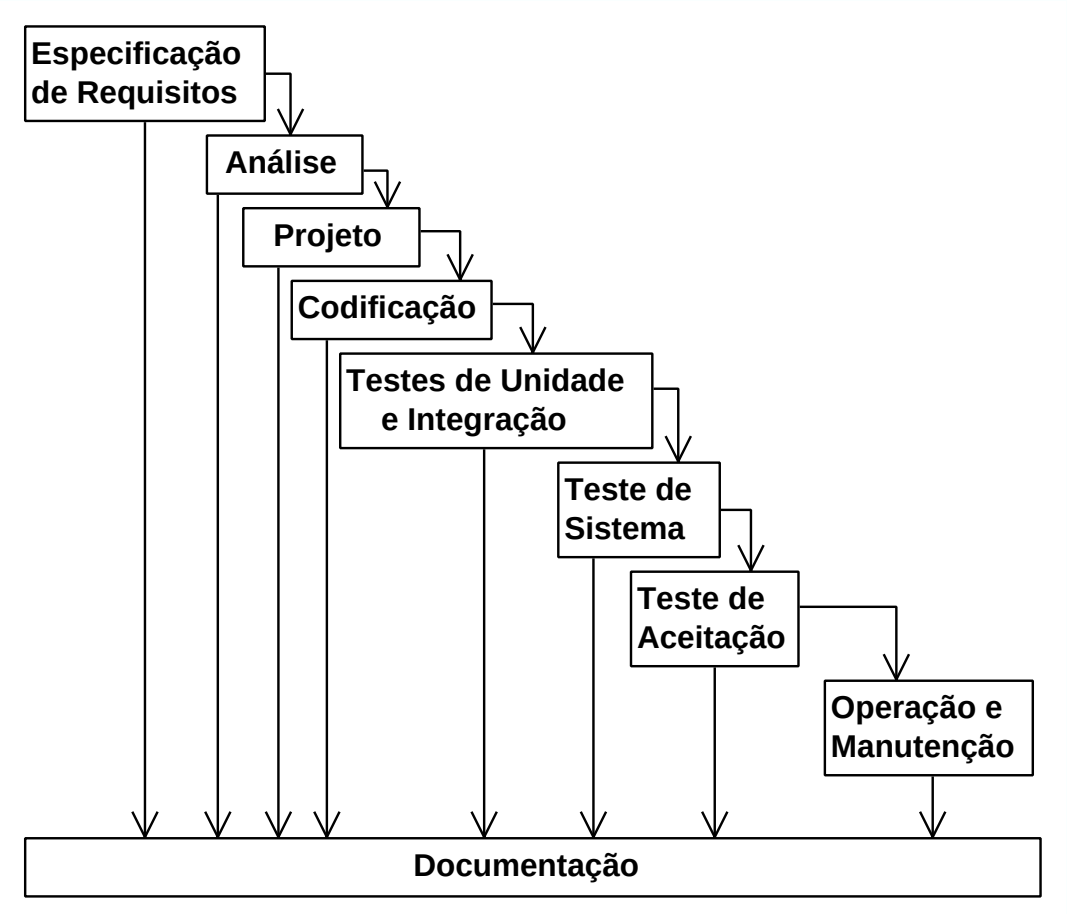
Modelos Tradicionais

- No modelo cascata, a fase de projeto usa decomposição funcional "top-down".
- Inicialmente, as funções (procedimentos) a serem realizadas pelo sistema são identificadas.
- Essas funções são divididas em subtarefas, que por sua vez são refinadas.
- Resultado: uma descrição hierárquica da estrutura do sistema.

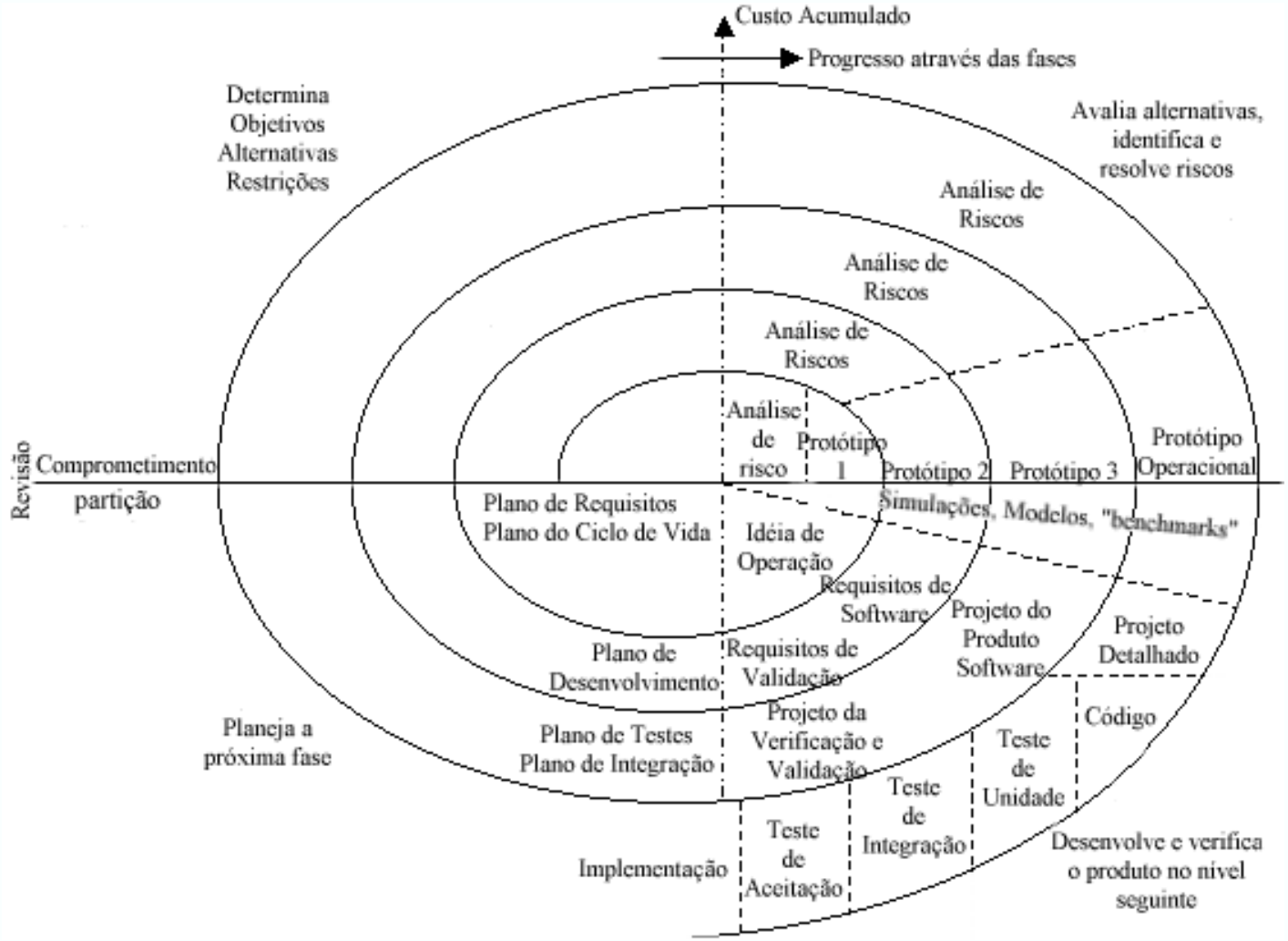
Limitações dos Modelos Tradicionais

- o sistema é inflexível, não pode ser adaptado facilmente a mudanças,
- o modelo não considera o uso de componentes de softwre já disponíveis. Abordagem "top-down" dificulta a reutilização de software. Por exemplo, projetistas de hardware iniciam um projeto examinando um catálogo de componentes já projetados ("bottom-up"),
- as diferentes fases não se integram de forma elegante e simples. Modelos e notações diferentes são usados para especificação, analise, projeto e implementação, dificultando as transições entre fases.

Modelo Cascata

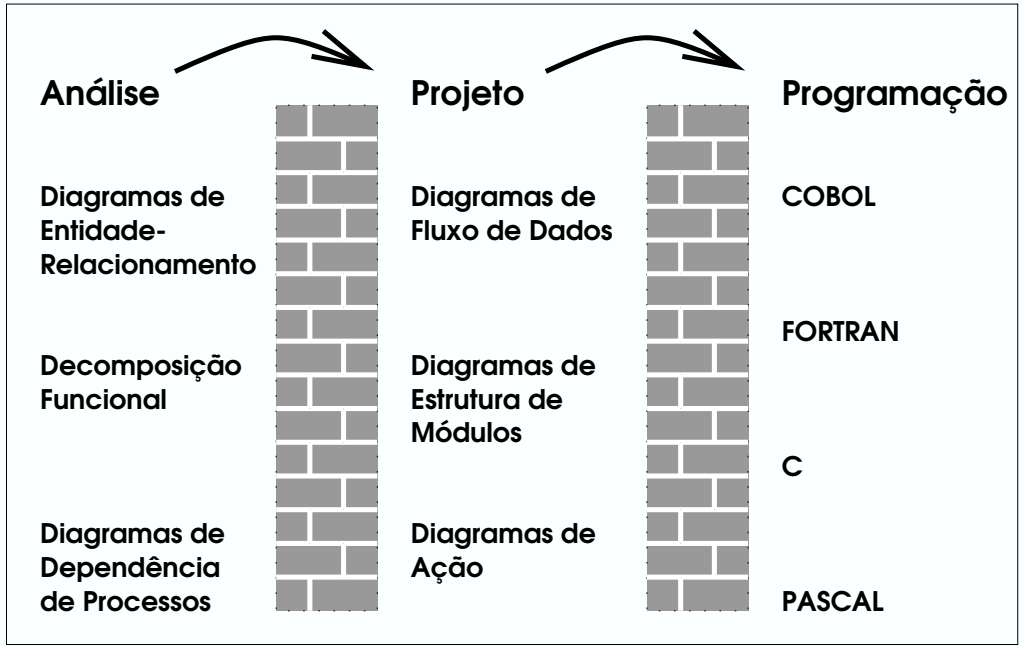


Modelo Espiral

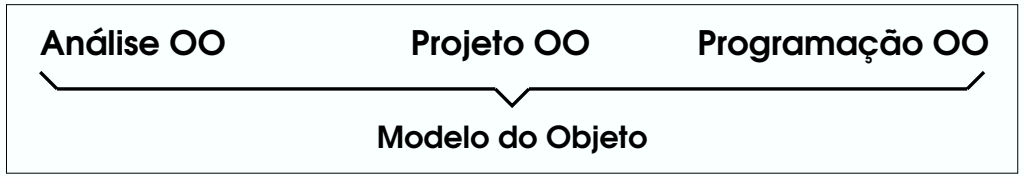


Comparação entre Processos Tradicionais vs. Orientados a Objetos

Nas metodologias tradicionais, analistas, projetistas e programadores têm diferentes modelos conceituais



A tecnologia de objetos usa um modelo unificado



Processo Orientado a Objetos (I)

Idéias Chaves:

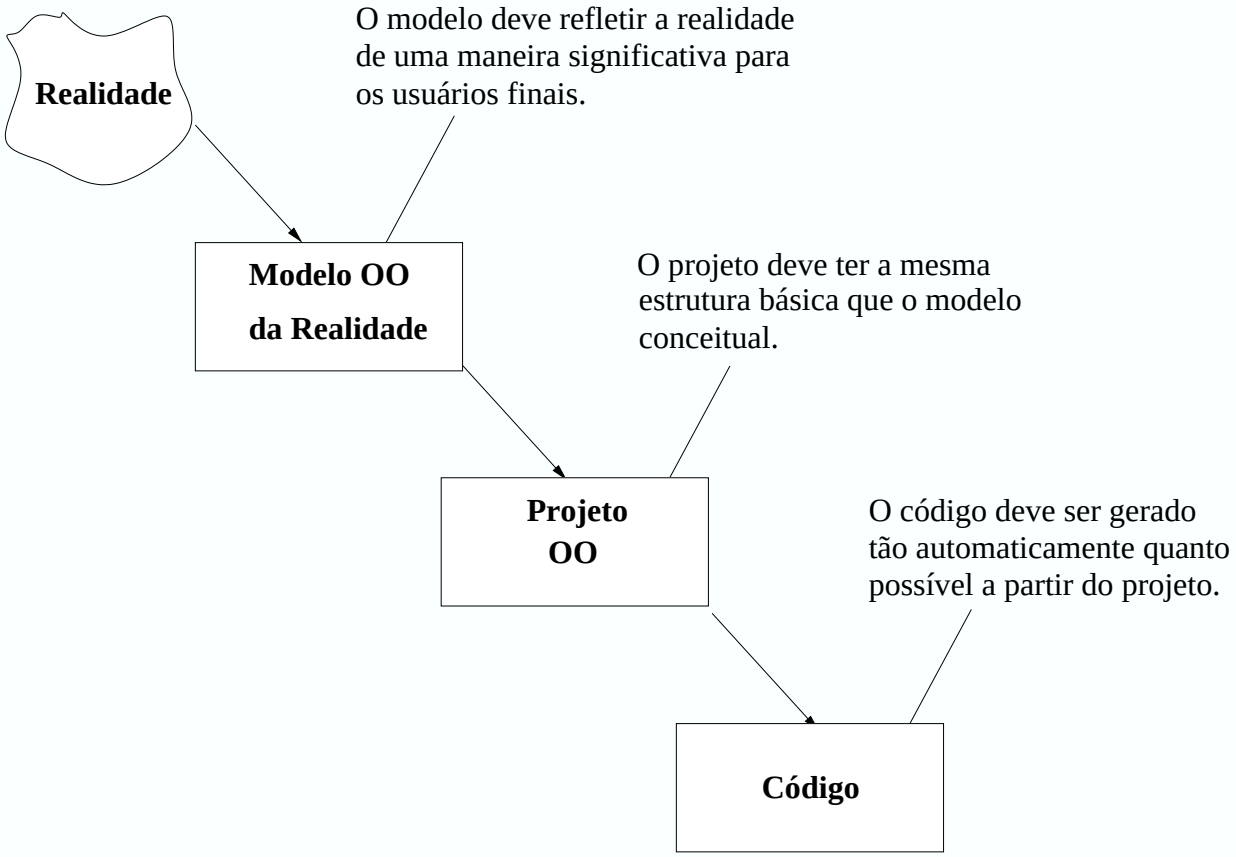
- Baseia-se em objetos e não em procedimentos.
- O projetista não inicia o processo pela identificação das funções do sistema, mas sim pela identificação dos objetos.

Processo Orientado a Objetos (II)

Motivação:

- Mudanças nos requisitos da especificação tendem a afetar menos os objetos do que as suas funções, portanto, o software se torna menos vulnerável.
- Os componentes de um sistema orientado a objetos são mais facilmente reutilizados pois eles escondem a representação de dados e sua implementação.
- O projeto orientado a objetos leva a uma melhor integração das diferentes fases, porque cada uma das fases lida com o mesmo tipo de entidades: os objetos.

Modelagem da Realidade



Unified Modeling Language (UML)

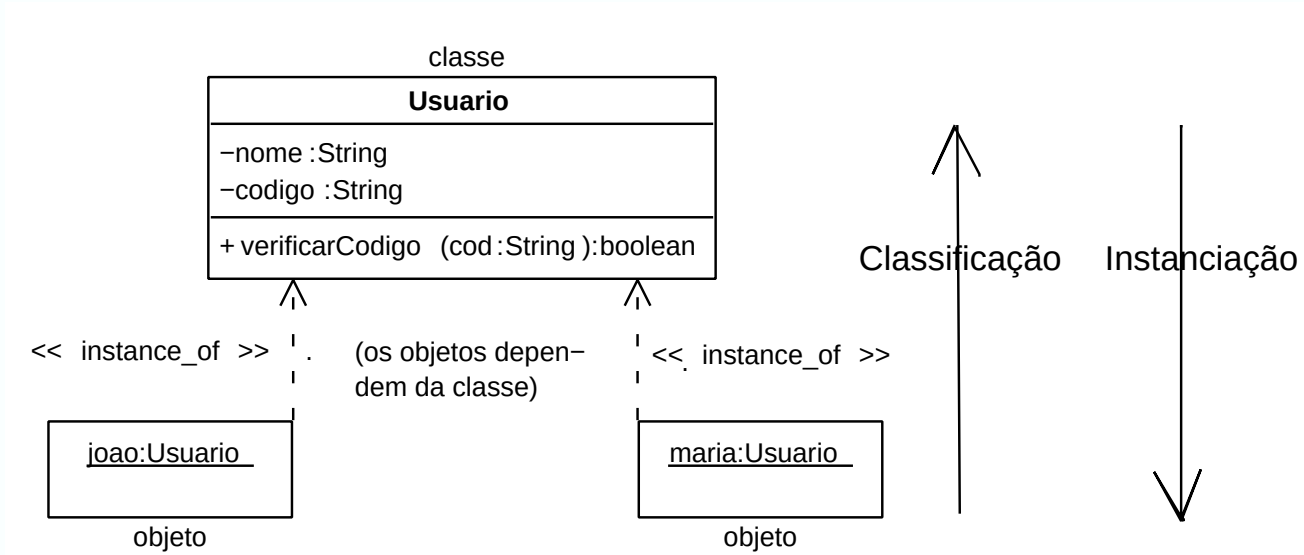
- É uma linguagem de notação que pode ser utilizada para representar modelos OO.
- Sua especificação iniciou-se em 1994 pela recém criada *Rational Software Corporation.*
- Tornou-se padrão do *Object Management Group* (OMG) em 1997.
- Se popularizou após a criação do *Rational Unified Process* (RUP), em 1998.

Exemplos de Hierarquias de Abstrações no Modelo OO

- Abstrações podem formar uma hierarquia.
- Três operações mentais são essenciais:

1. **Classificação/instanciação** para construir uma abstração.
2. **Agregação/decomposição** para construir uma hierarquia de agregação.
3. **Generalização/especialização** para construir uma hierarquia de generalização.

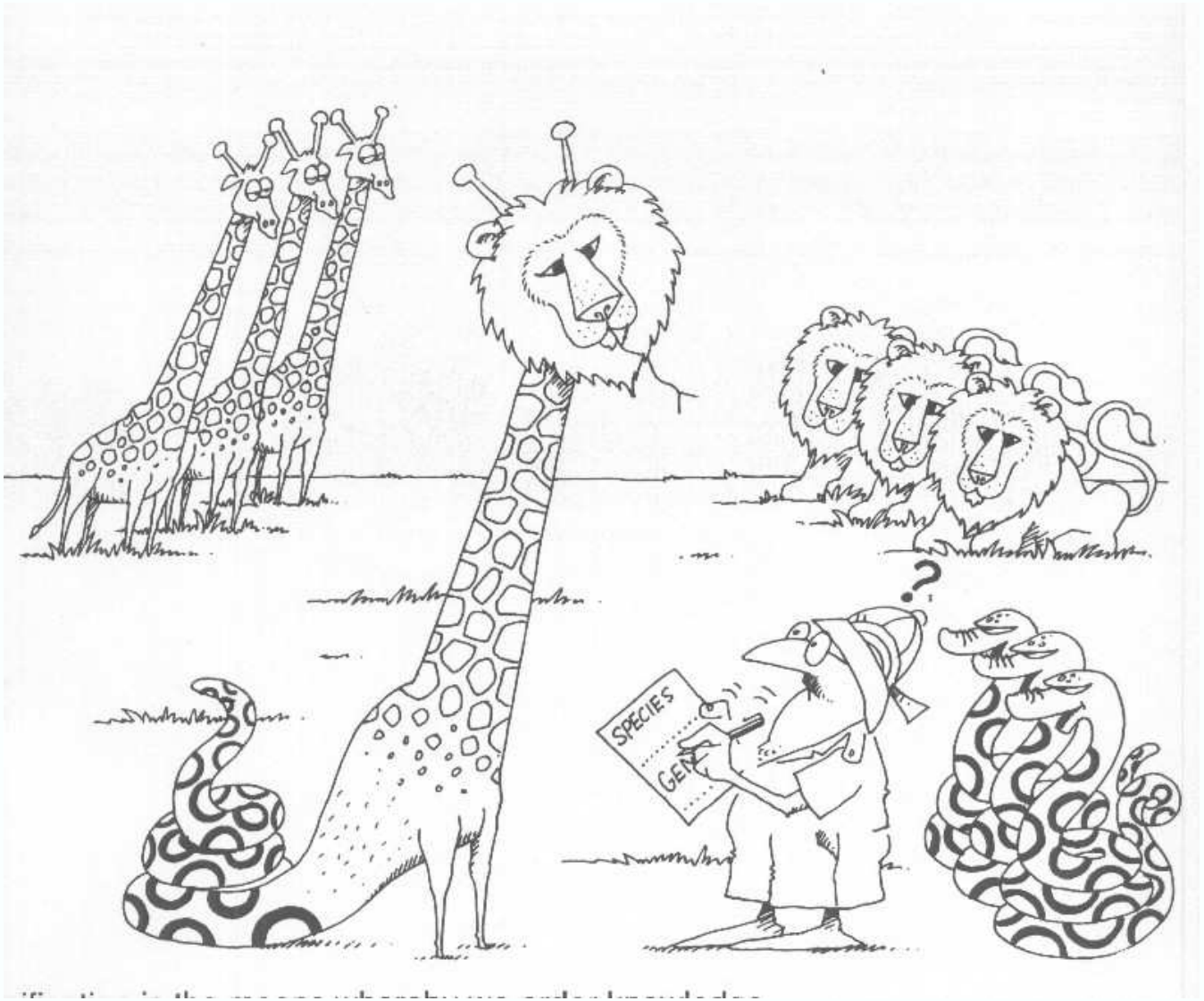
Classificação/Instanciação (I)



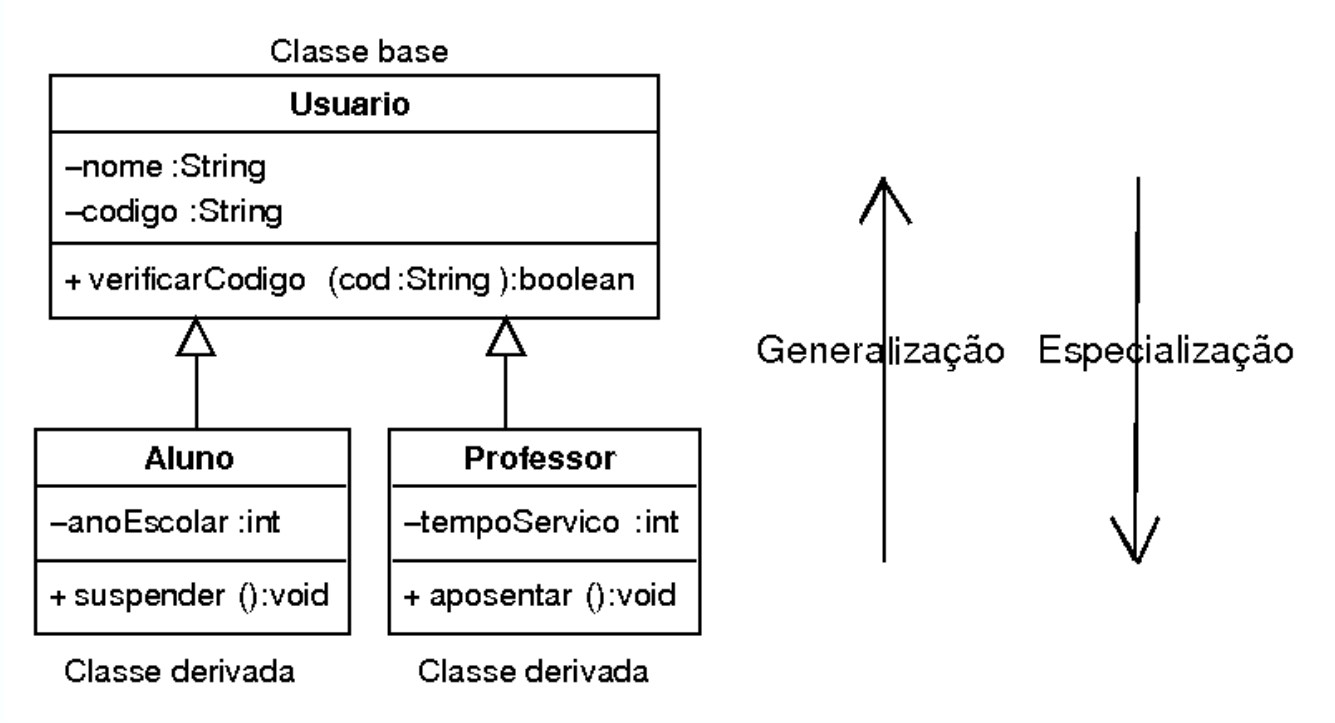
OBS.: No contexto de um sistema de controle de bibliotecas

- Um objeto é sempre instância de uma única classe, nunca de 2 ao mesmo tempo.
- Uma classe pode possuir várias instâncias (objetos).

Classificação/Instanciação (II)



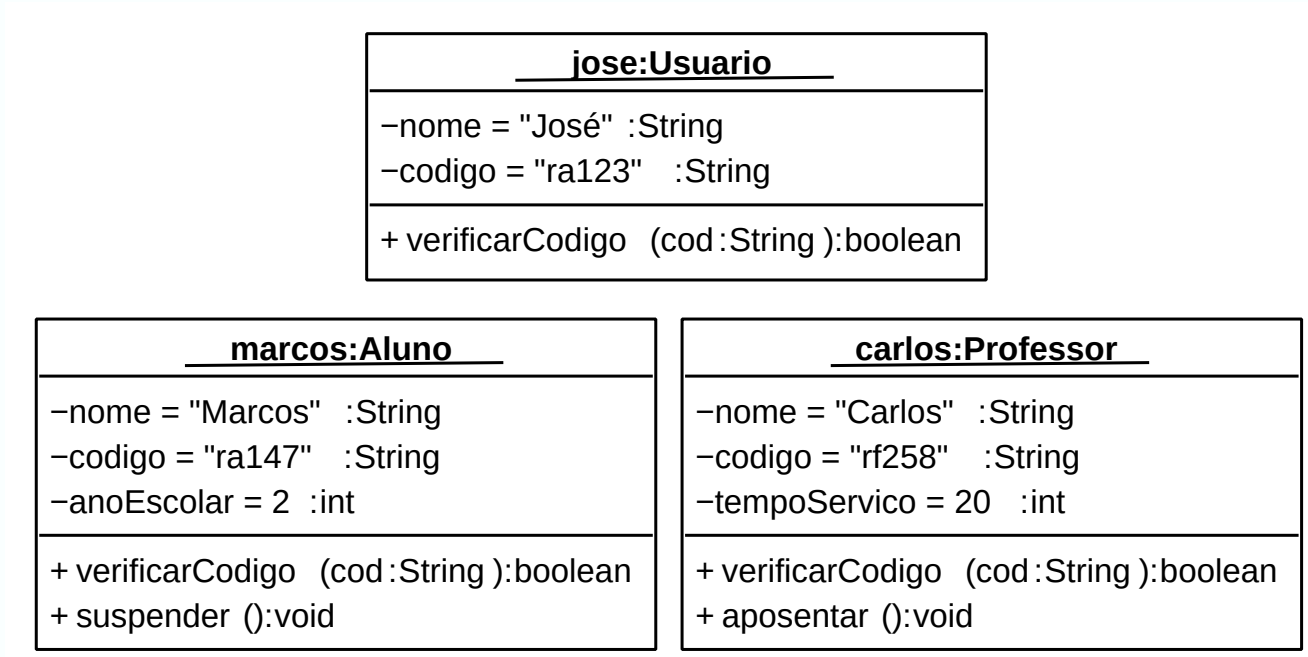
Generalização/Especialização (I)



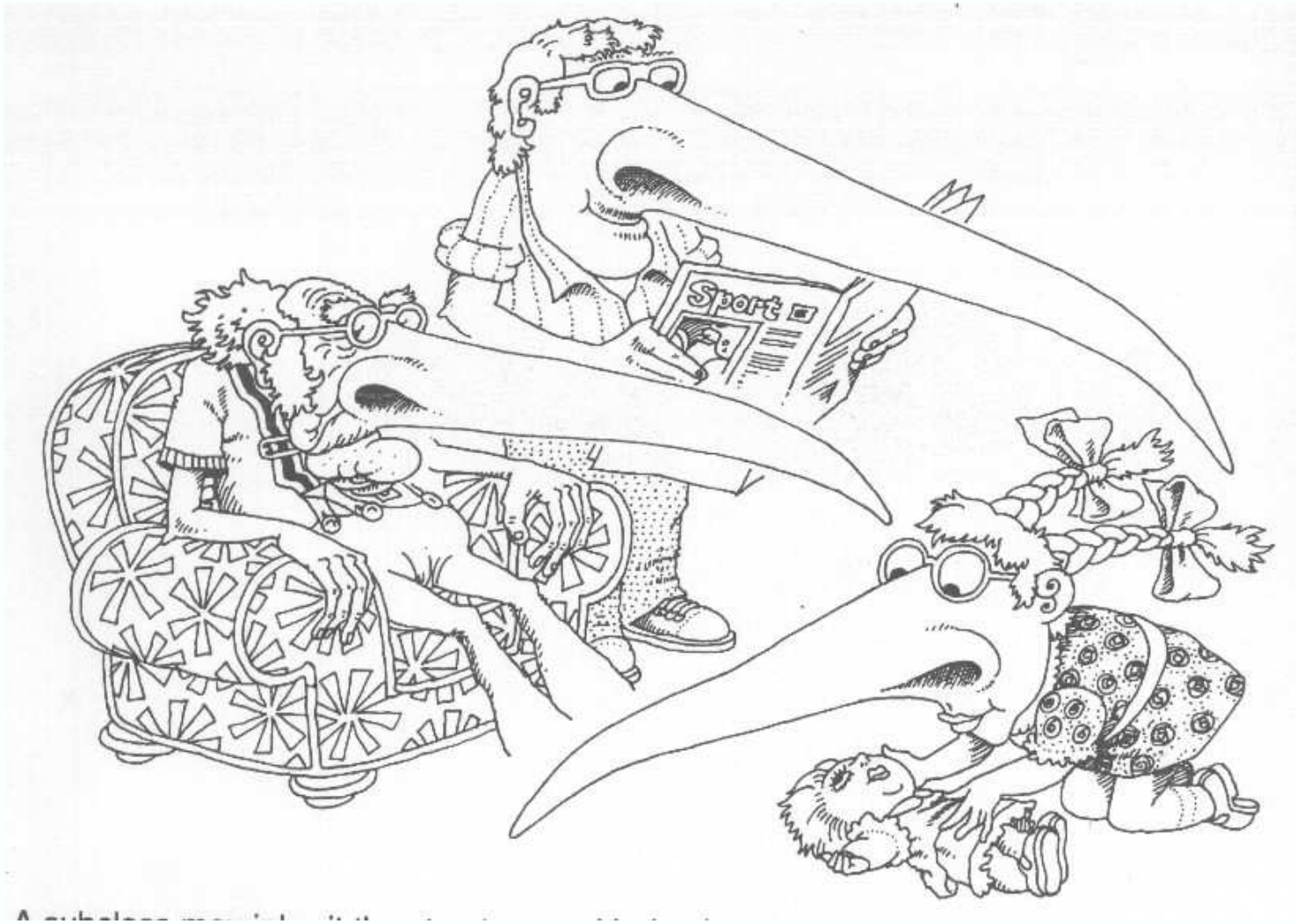
- Implementa o conceito de herança de tipos: é-um-tipo-de
- Permite que todas as instâncias de uma categoria específica também pertençam a instâncias de uma categoria mais abrangente

Generalização/Especialização (II)

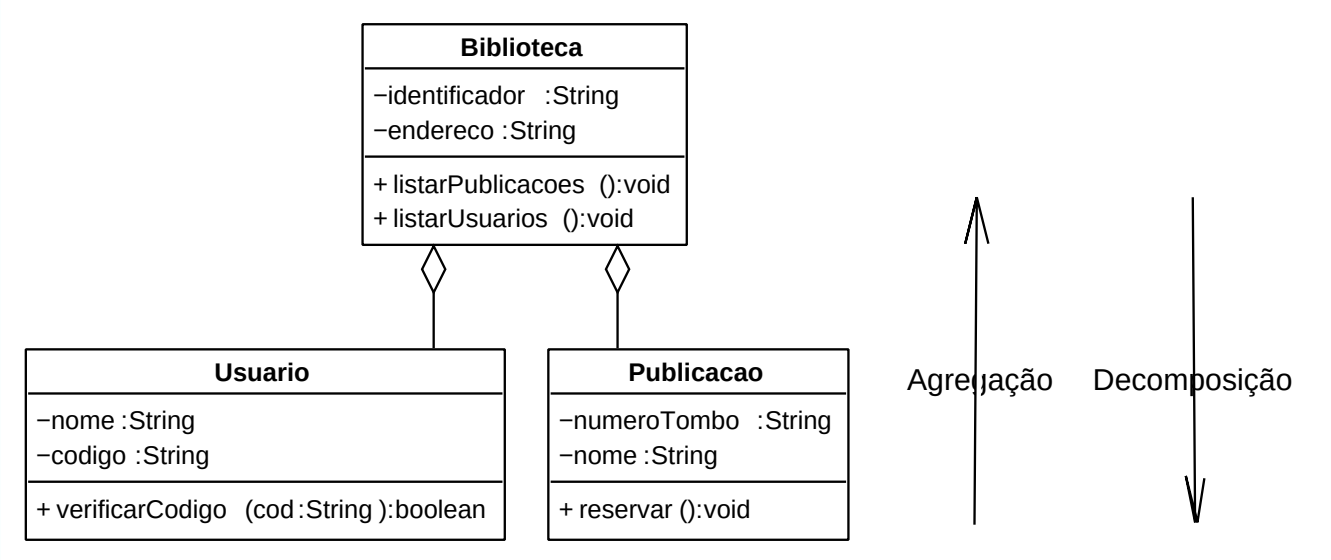
- Diagrama de Objetos:



Generalização/Especialização (III)



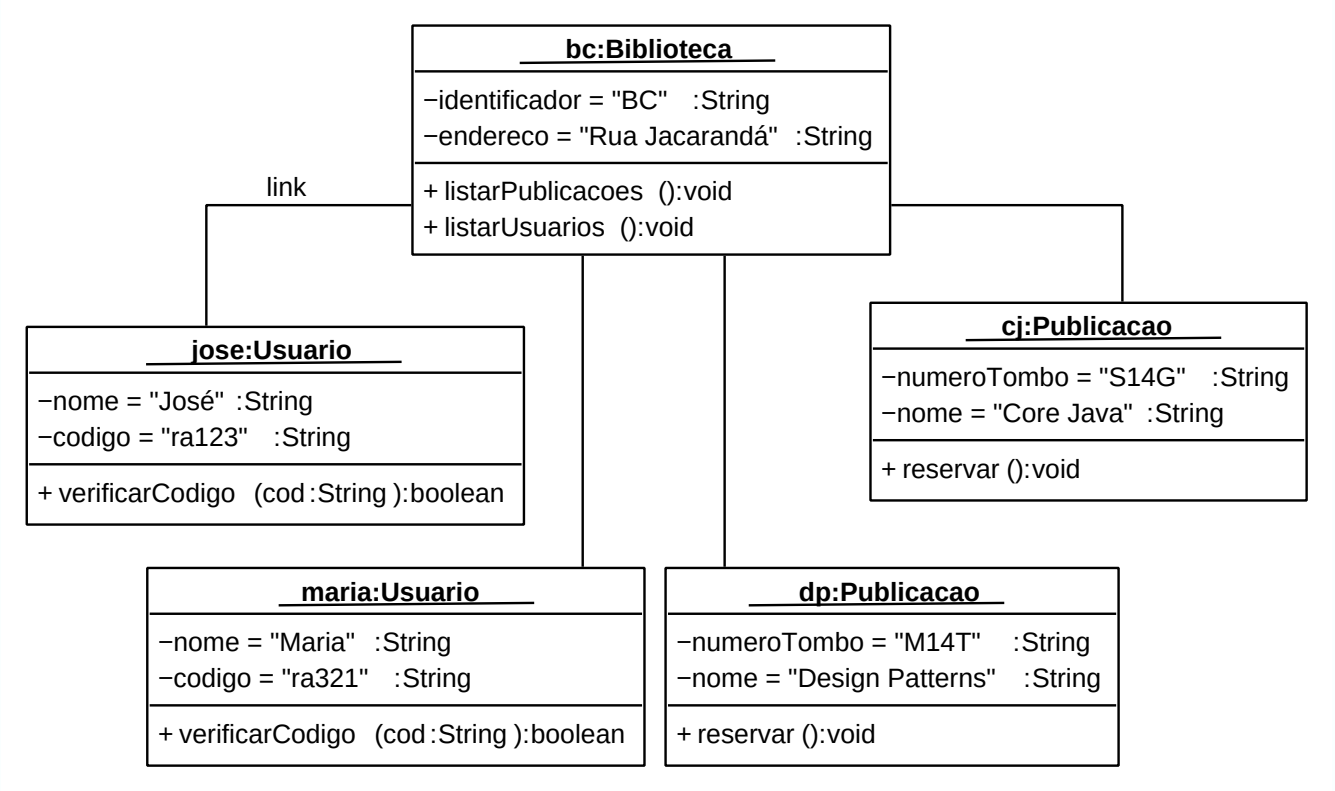
Agregação/Decomposição (I)



- Agregação é um relacionamento estrutural entre o todo e suas partes.
- Ela implementa o conceito de decomposição hierárquica: é-parte-de
- É um mecanismo que permite a montagem do todo a partir de suas partes

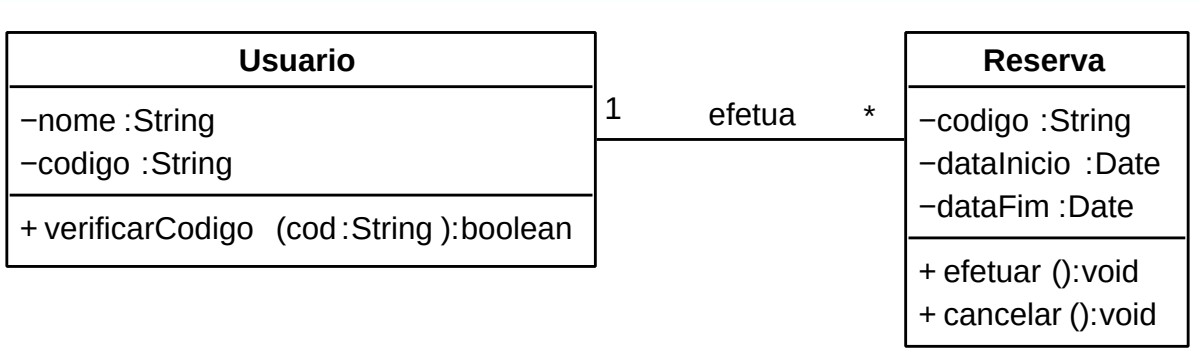
Agregação/Decomposição (II)

- Diagrama de Objetos:



- A agregação é representada como um conjunto de "links";
- Um "link" é uma conexão entre dois objetos.

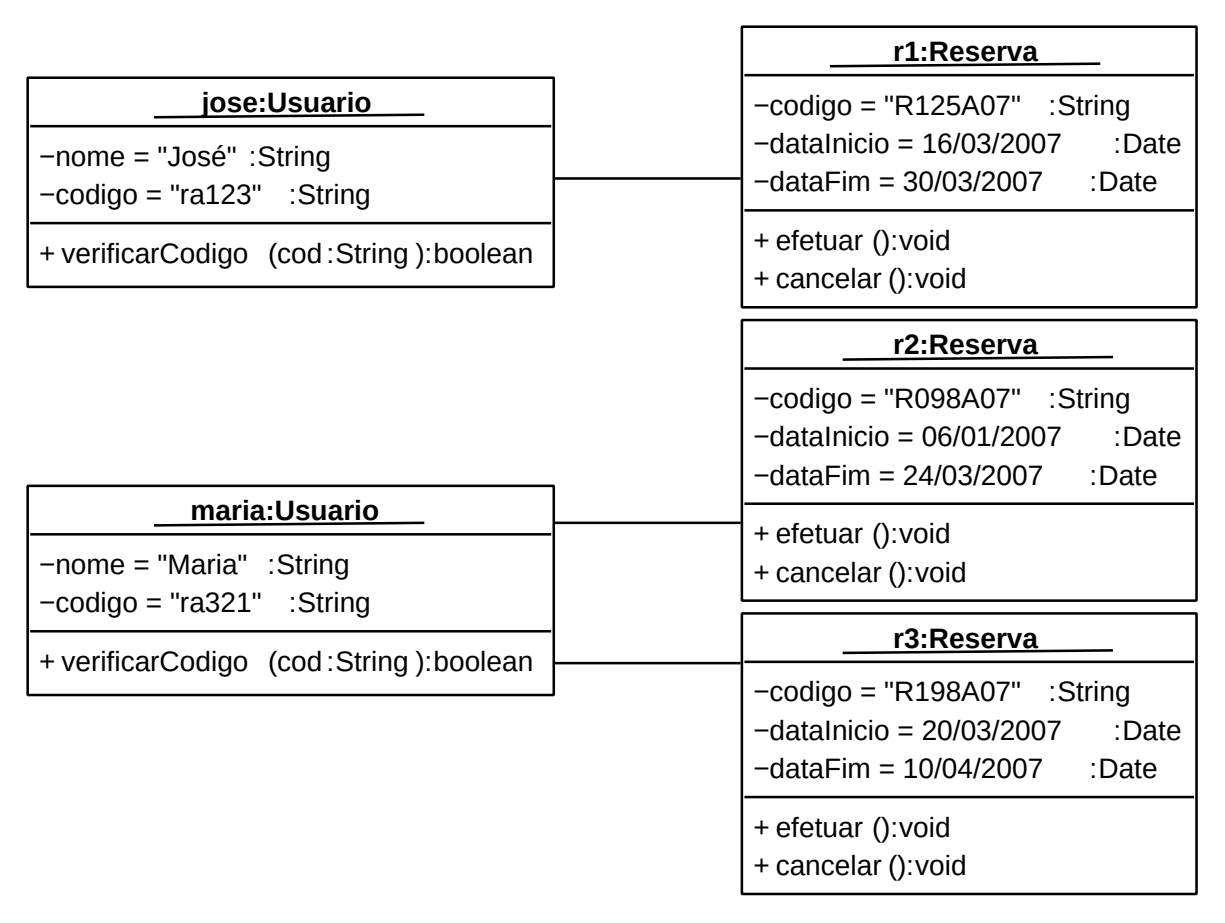
Associação (I)



- Uma associação é um relacionamento estrutural que descreve um conjunto de "links";
- Agregação é um tipo especial de associação;
- Um usuário pode efetuar várias reservas e uma reserva é feita por apenas um usuário.

Associação (II)

- Diagrama de Objetos:



Exemplos de Modelagem Orientada a Objetos

Enunciado 1: Sistema de Controle Acadêmico

Considere como domínio do problema uma universidade como a UNICAMP, onde existem diversas unidades, que podem ser institutos, faculdades e núcleos. Unidades são subdivididas em departamentos. Uma universidade tem um quadro de pessoal permanente composto por funcinários, que podem ser docentes, secretários, analistas, copeiras, etc.

Nos cursos oferecidos pela universidade ingressam alunos de graduação, pós-graduação e de extensão universitária. Cursos são compostos por disciplinas, de acordo com o catálogo de cursos da universidade. Alunos podem se matricular em turmas de uma disciplina específica. A pós-graduação pode ter programas de mestrado acadêmico,

mestrado profissional e de doutorado.

Solução: Domínio Universidade

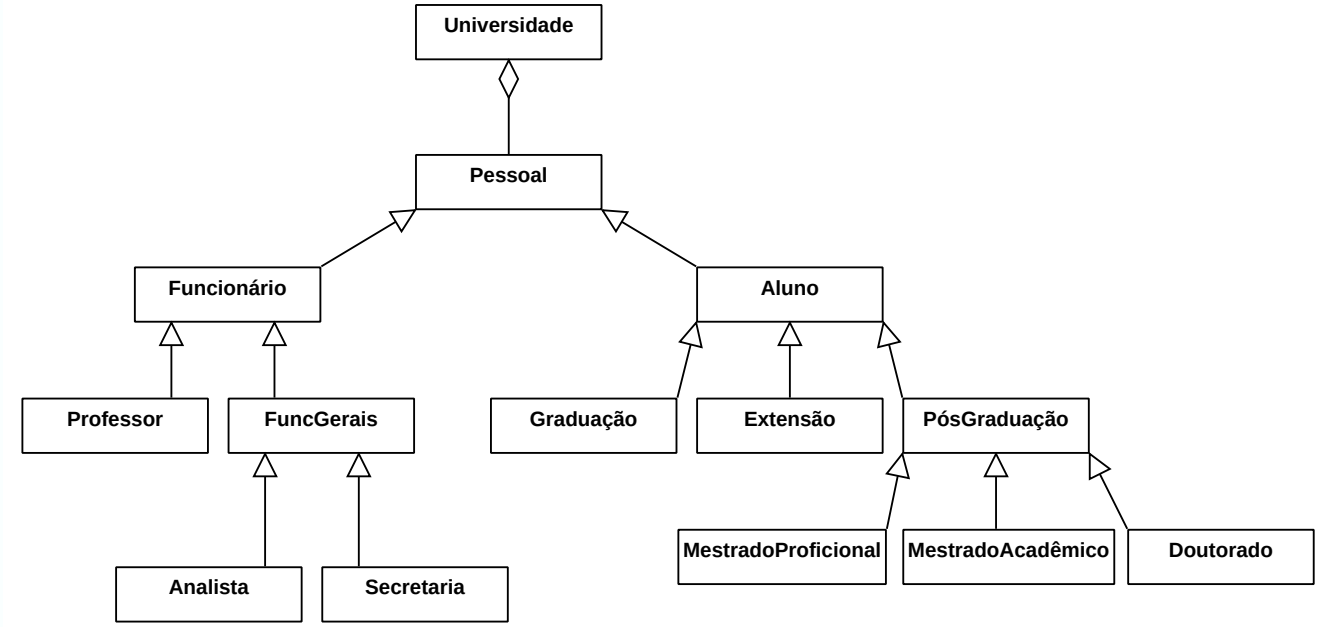


Figura 1: Exemplo de Generalização/Especialização

Enunciado 2: Domínio Zoológico

Considere como domínio do problema, um zoológico onde existem diversos tipos de animais com uma estrutura administrativa, como por exemplo, diretor, tratadores, veterinários, etc. A administração do zoo envolve o preparo de cardápios específicos para cada tipo de animal.

Além disso, o sistema deve possibilitar o cadastro dos animais existentes no zoo, classificados de acordo com sua respectiva classe: mamífero, réptil, ave, etc.

Solução: Domínio Zoológico

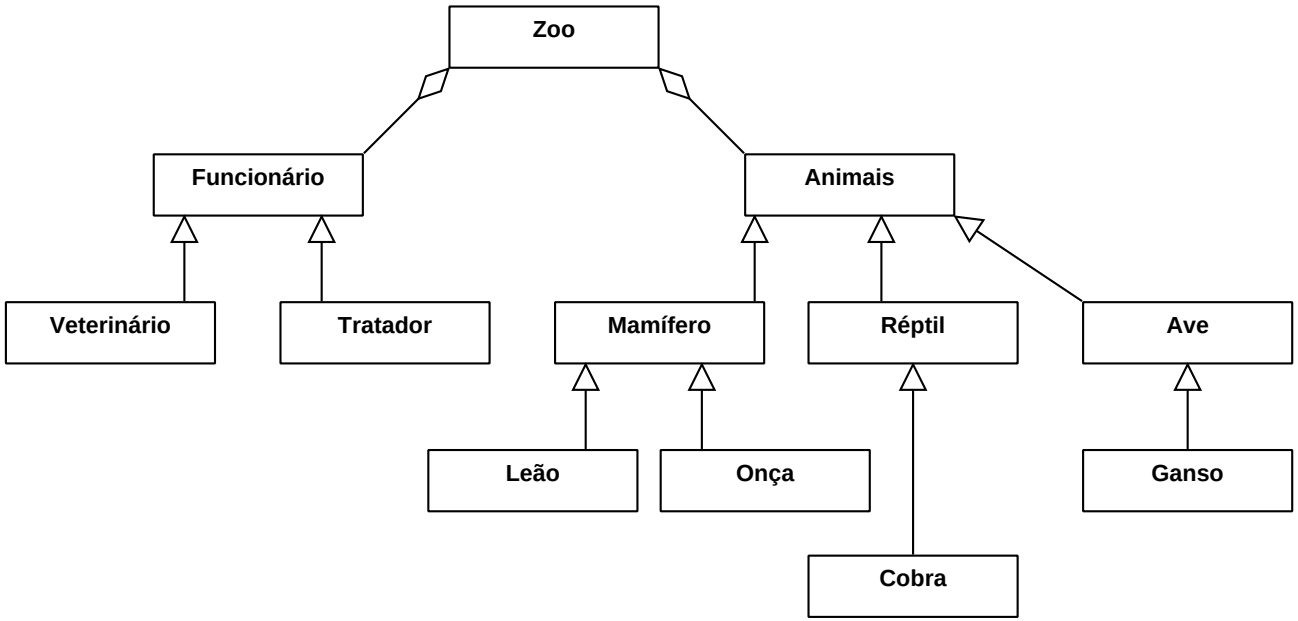


Figura 2: Exemplo de Generalização/Especialização

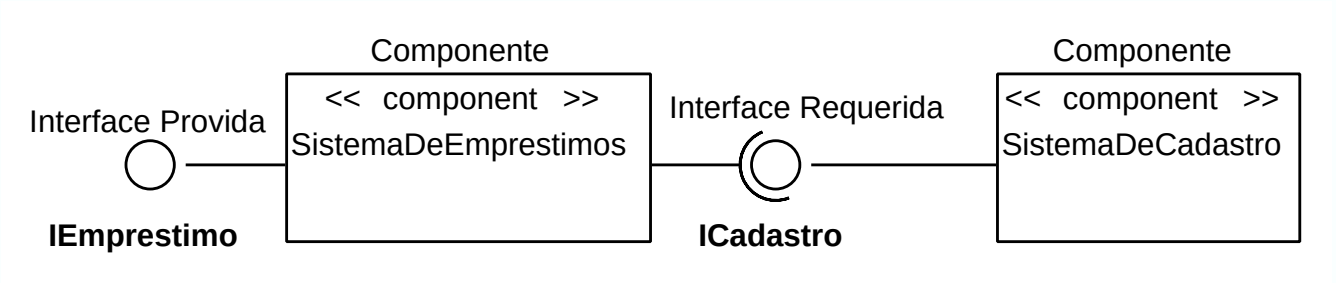
Desenvolvimento Baseado em Componentes (DBC)

- No DBC, a aplicação é construída a partir da composição de componentes de software.
- Ganho de produtividade:
  - Decorrente da reutilização de componentes existentes na construção de novos sistemas.
  - Possibilidade de usar componentes de prateleira (do inglês COTS *Common-Of-The-Shelf*);
- Ganho de qualidade:
  - Uma conseqüência do fato dos componentes reutilizados já terem sido empegrados e testados em outros contextos;

Componente de Software

- Unidade de composição com interfaces especificadas por meio de contratos e dependências de contexto explícitas:
  - **Interfaces Providas.** Definição dos serviços oferecidos pelo componente;
  - **Interfaces Requeridas.** Definição dos serviços necessários para o componente funcionar adequadamente;

Representação de Componentes em UML



Especificação de Requisitos de Software

- A primeira atividade do desenvolvimento de software;
- Abrange quatro etapas:
  1. Identificação do domínio do problema;
  2. Identificação dos **requisitos funcionais** esperados pelo sistema (objetivos);
  3. Definição de restrições existentes para o desenvolvimento;
  4. Identificação dos **requisitos não-funcionais** desejados.

Classificação dos Requisitos

- Requisitos Funcionais. Representam o comportamento esperado pelo sistema. Esse comportamento pode ser representado como serviços, tarefas ou funções que o sistema deve oferecer.
- Requisitos Não-Funcionais. Descrevem os atributos de qualidade do sistema. Apesar de não representarem funcionalidades diretamente, esses atributos podem interferir na maneira como o sistema deve ser estruturado para executá-las.

Padrões de Nomenclatura para Requisitos Não-Funcionais ABNT/ISO 9126

Os Requisitos Não-Funcionais são classificados em 6 Grupos:

Grupo 1. Funcionalidade. (satisfaz às necessidades?)

- Adequação. Existência de um conjunto de funções e sua apropriação para as tarefas especificadas.
- Acurácia. Gera resultados ou efeitos corretos (conforme acordado).
- Interoperabilidade. Interage com outros sistemas.
- Conformidade. Está de acordo com as normas, convenções ou regulamentações previstas.
- Segurança de acesso. Evita o acesso não autorizado, acidental ou deliberado, a programas e dados.

Grupo 2. Confiabilidade (é imune às possíveis falhas?)

- Maturidade. Freqüência de falhas por defeitos do software.
- Tolerância a Falhas. Mantém o sistema funcionando, mesmo na ocorrência das falhas especificadas (falhas toleráveis).
- Recuperabilidade. É a característica do sistema poder recuperar seus dados, isto é, a garantia da existência de cópias de segurança dos dados.

Grupo 3. Usabilidade (é fácil de usar?)

- Inteligibilidade. Esforço do usuário para conhecer o conceito lógico e sua aplicabilidade.
- Apreensibilidade. Esforço do usuário para compreender/assimilar o funcionamento do sistema.
- Operacionalidade. Esforço do usuário para controlar a operação do sistema.

Grupo 4. Eficiência (é rápido e 'enxuto'?)

- Comportamento em relação ao tempo. Tempo de resposta, tempo de processamento e velocidade na execução das funcionalidades.
- Comportamento em relação aos recursos. Quantidade de recursos usados e a duração de seu uso na execução das funcionalidades.

Grupo 5. Manutenibilidade (é fácil de modificar?)

- Analisabilidade. Esforço necessário para diagnosticar deficiências ou causas de falhas, ou para identificar as partes a serem modificadas.
- Modificabilidade. Esforço necessário para modificar o sistema: modificações e atualizações.
- Estabilidade. Risco de defeitos inesperados ocasionados por modificações.
- Testabilidade. Esforço necessário para validar o software modificado.

Grupo 6. Portabilidade (é fácil de usar em outro ambiente?)

- Adaptabilidade. Adaptável a ambientes diferentes, sem a necessidade de se aplicar outras ações ou meios, além das definidas no próprio sistema.
- Capacidade para ser instalado. Esforço necessário para a sua instalação num ambiente especificado.
- Conformidade. Está consoante com padrões ou convenções relacionados à portabilidade.
- Capacidade para substituir. Capacidade e esforços necessários para substituir por um outro sistema, respeitando o ambiente existente.