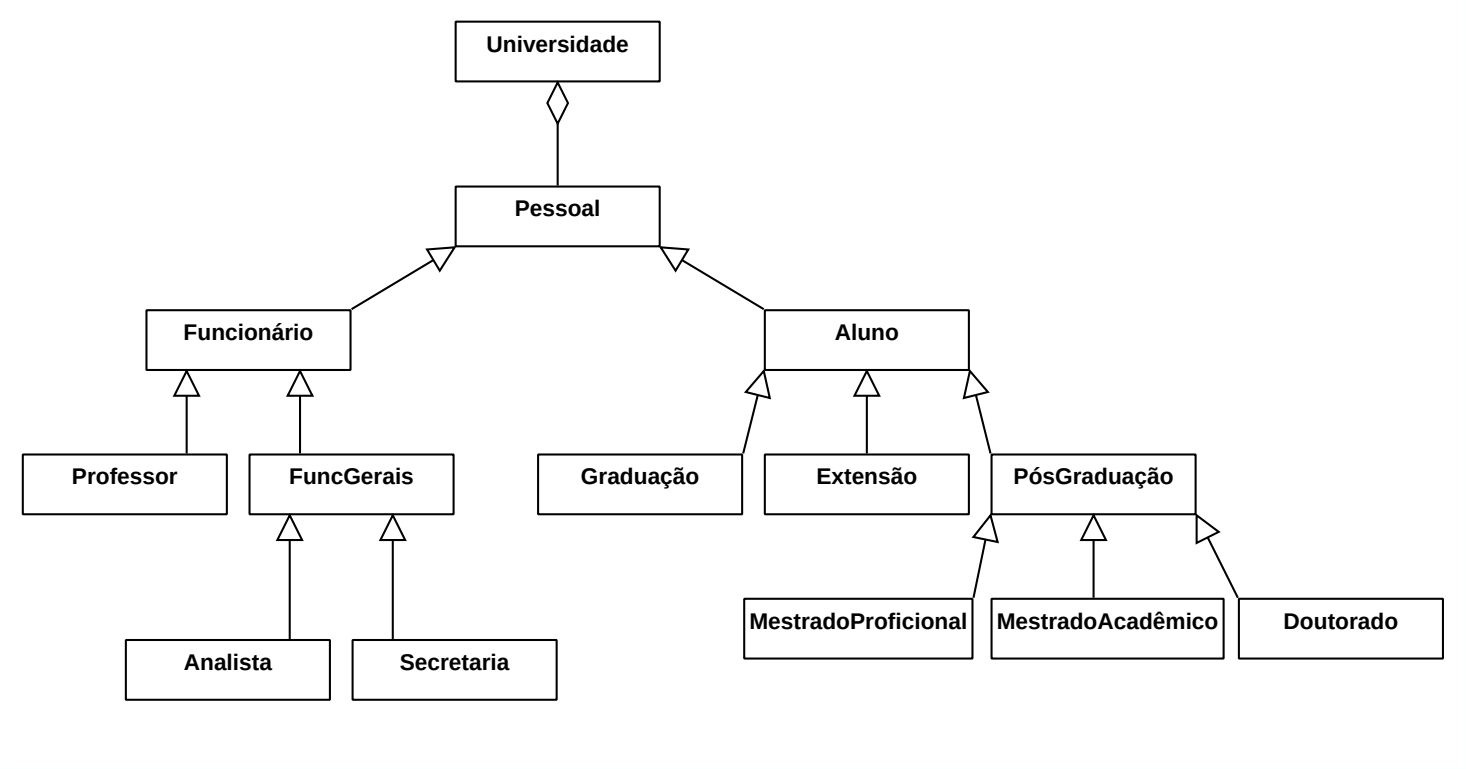
Enunciado 1: Domínio Universidade

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 funcionários, que podem ser professores, 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 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

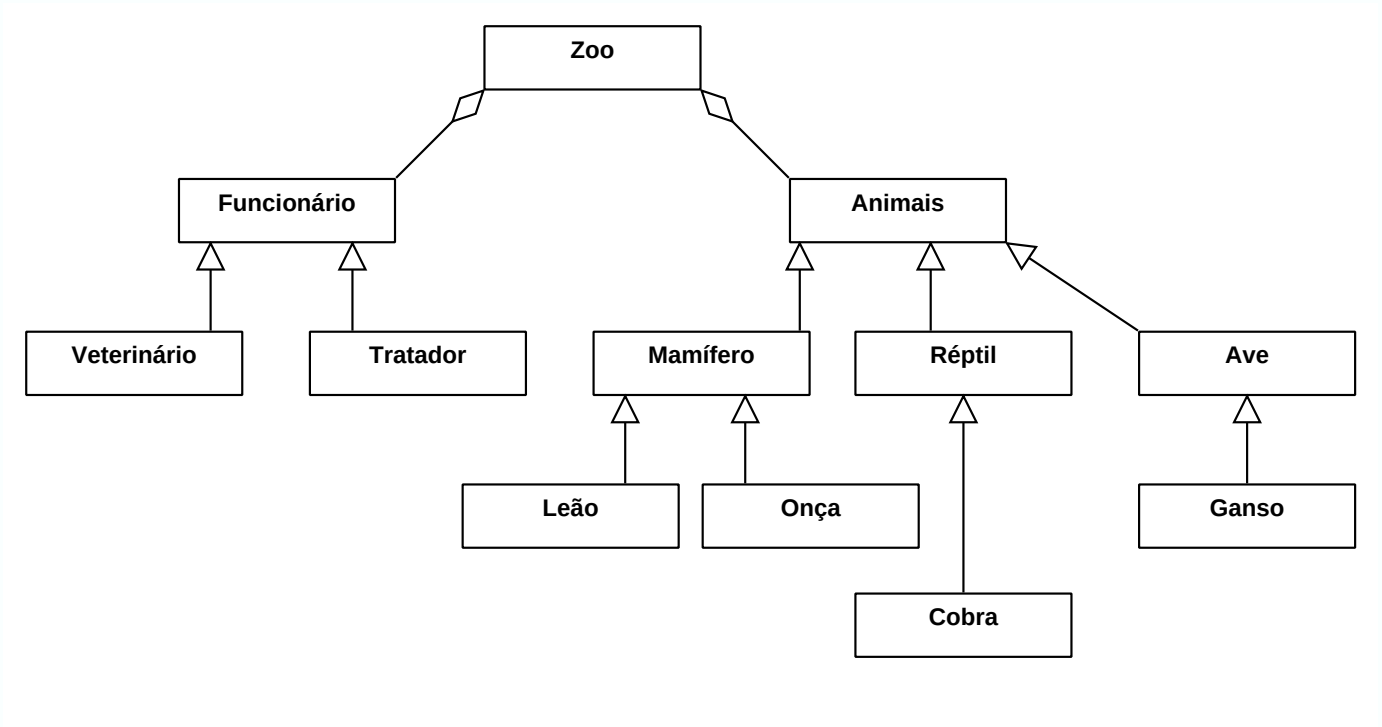


Enunciado 2: Domínio Zoológico

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 bicho.

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

Solução: Domínio Zoológico



Objetos e Classes

- **Tópicos:** hierarquia de abstrações, objetos, classes, tipos abstratos de dados.
- **Objetivos:** exercitar a criação de classes e objetos, o acesso aos elementos de um objeto (atributos e operações) e a elaboração de tipos abstratos de dados.
- **Pré-requisitos:** conceitos fundamentais de programação estruturada.

Abstração de Dados ou Tipos Abstratos de Dados (TAD)

Abstração de dados proporciona uma abstração sobre uma **estrutura de dados** em termos de uma **interface bem definida**

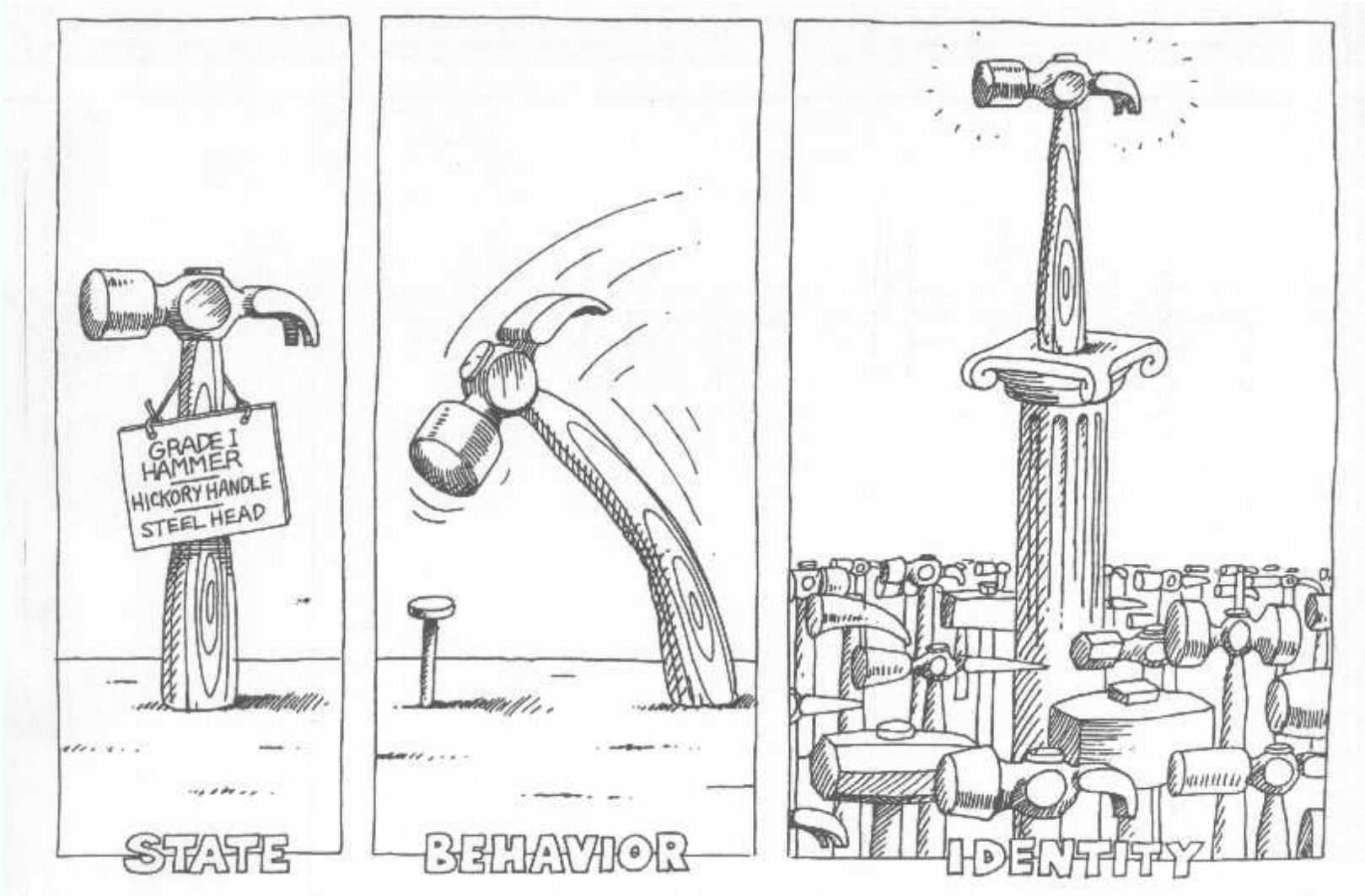
Vantagens:

- O fato do código e estrutura de dados de uma abstração estarem armazenados num mesmo lugar, cria um programa bem estruturado e legível que pode ser facilmente modificável
- O aspecto do ocultamento da informação e encapsulamento proporciona um nível de proteção contra acessos inesperados à estrutura de dados, o que mantém a integridade do objeto

Objetos (I)

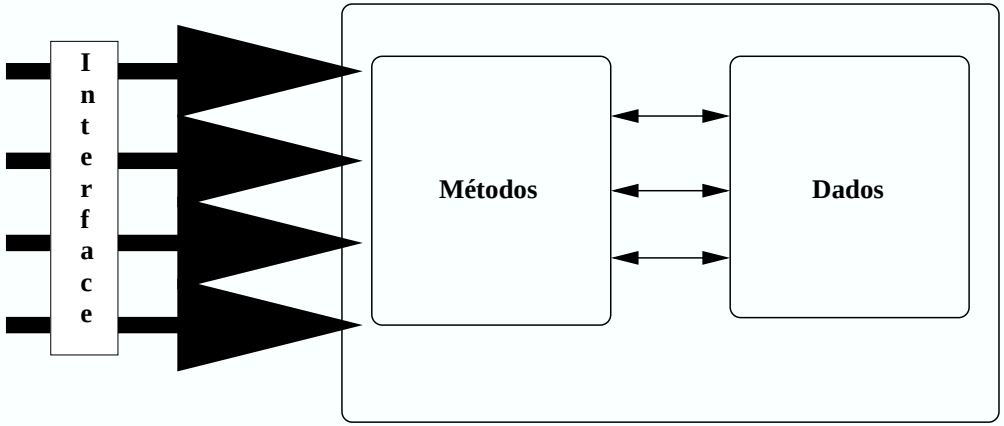
- Objetos são entidades que encapsulam **informação de estado** ou **dados**, e um conjunto de operações associadas que manipulam esses dados
- O estado de um objeto é completamente escondido e protegido de outros objetos, e a única maneira de examiná-lo é através da invocação de uma operação declarada na **interface pública** do objeto
- Objetos têm uma identidade única. **Identidade** é a propriedade de um objeto que o distingüe de outros objetos

Objetos (II)


GRADE I
HAMMER
HICKORY HANDLE
STEEL HEAD
STATE
BEHAVIOR
IDENTITY

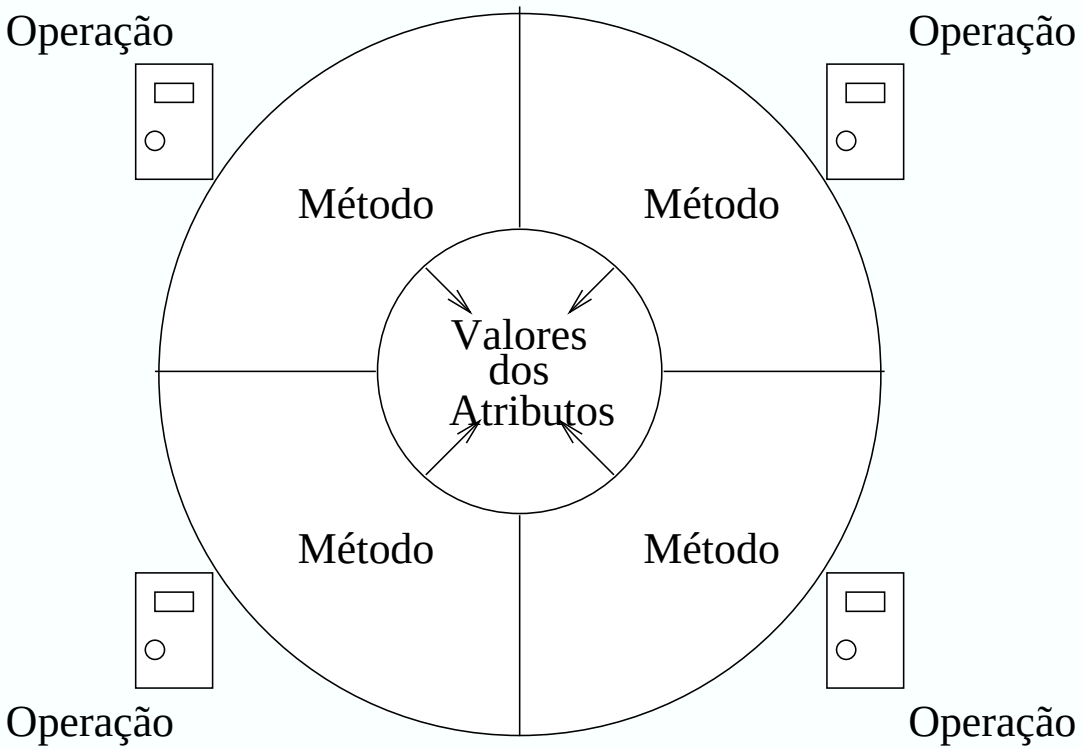
Identidade Única: Marty

Objetos (III)



- Em resumo, um objeto tem um estado, um comportamento ("behaviour") bem definido, e uma identidade que é única

Operações vs. Métodos



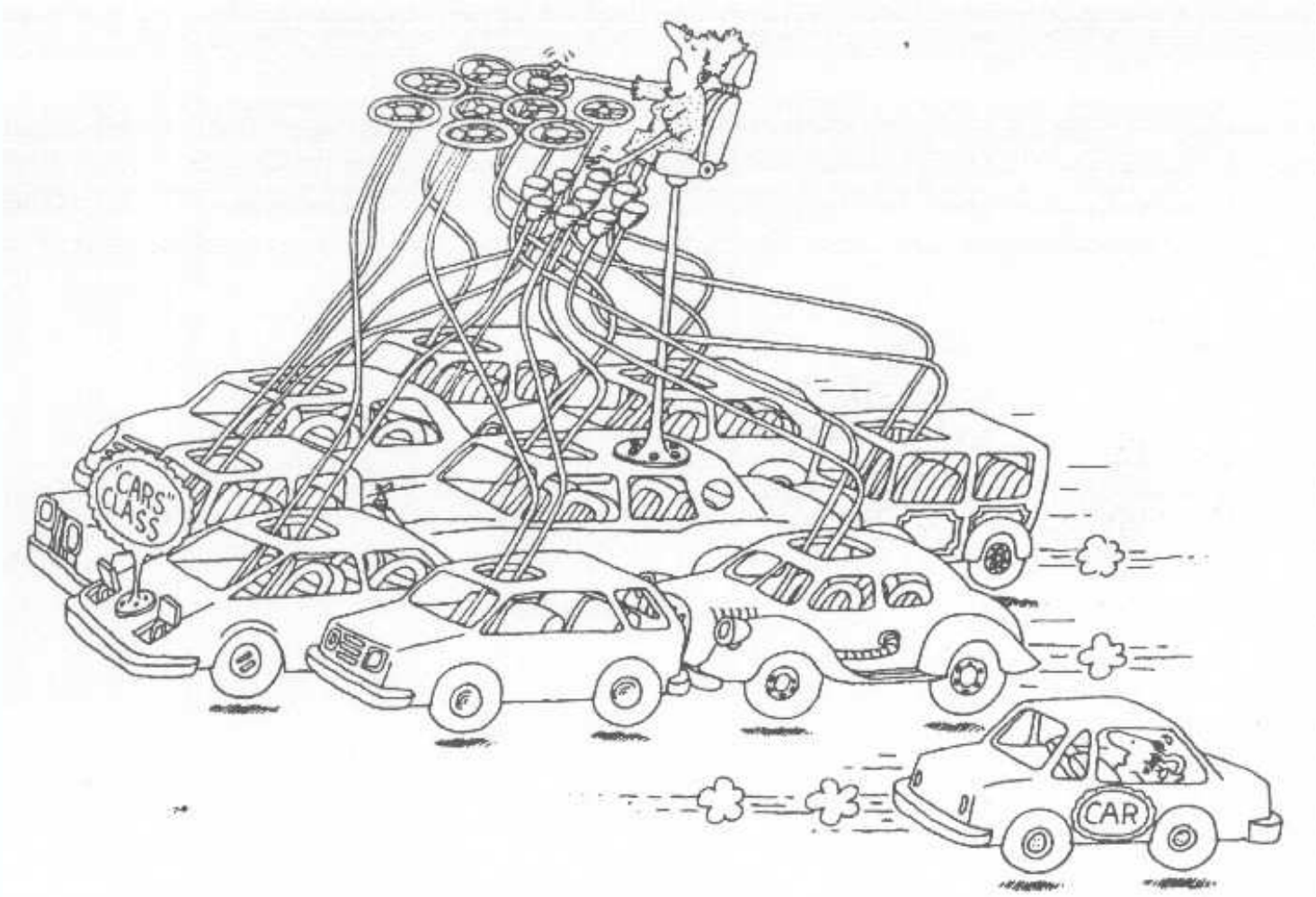
Objetos (IV)

- **Comportamento:** é como um objeto age e reage em termos das suas mudanças de estado e passagem de mensagens
- **Comportamento (na teoria TAD):** é usado para denotar a **interface abstrata** de um objeto, definida pelo conjunto de operações visíveis que podem ser aplicadas ao mesmo
- **Operação:** é alguma ação que um objeto realiza sobre um outro para explicitar uma reação

Classes (I)

- Uma classe é a descrição de um molde ("skeleton") que especifica os atributos (ou propriedades) e o comportamento para um conjunto de objetos similares
- Todo objeto é **instância** de uma classe
- Atributos e operações são partes da definição de uma classe
- **Atributos** são propriedades nomeadas de um objeto que armazenam o estado abstrato de cada objeto
- **Operações** caracterizam o comportamento de um objeto, e são o único meio para acessar, manipular e modificar os atributos de um objeto

Classes (II)



Classes (III)

- Um objeto comunica-se com outro através de mensagens que identificam operações a serem realizadas no segundo objeto
- Um objeto responde a uma mensagem mudando possivelmente os valores de seus atributos e/ou retornando um resultado
- A interface de uma classe compreende o conjunto de operações que podem ser chamadas por outros objetos
- A visão externa de um objeto não é nada mais do que a sua **interface pública**

Classes (IV)

- Métodos são implementações de operações que um cliente pode chamar num determinado objeto
- Operação é diferente de método. Operação é abstrata. Método é concreto.

Dicas e Recomendações (I)

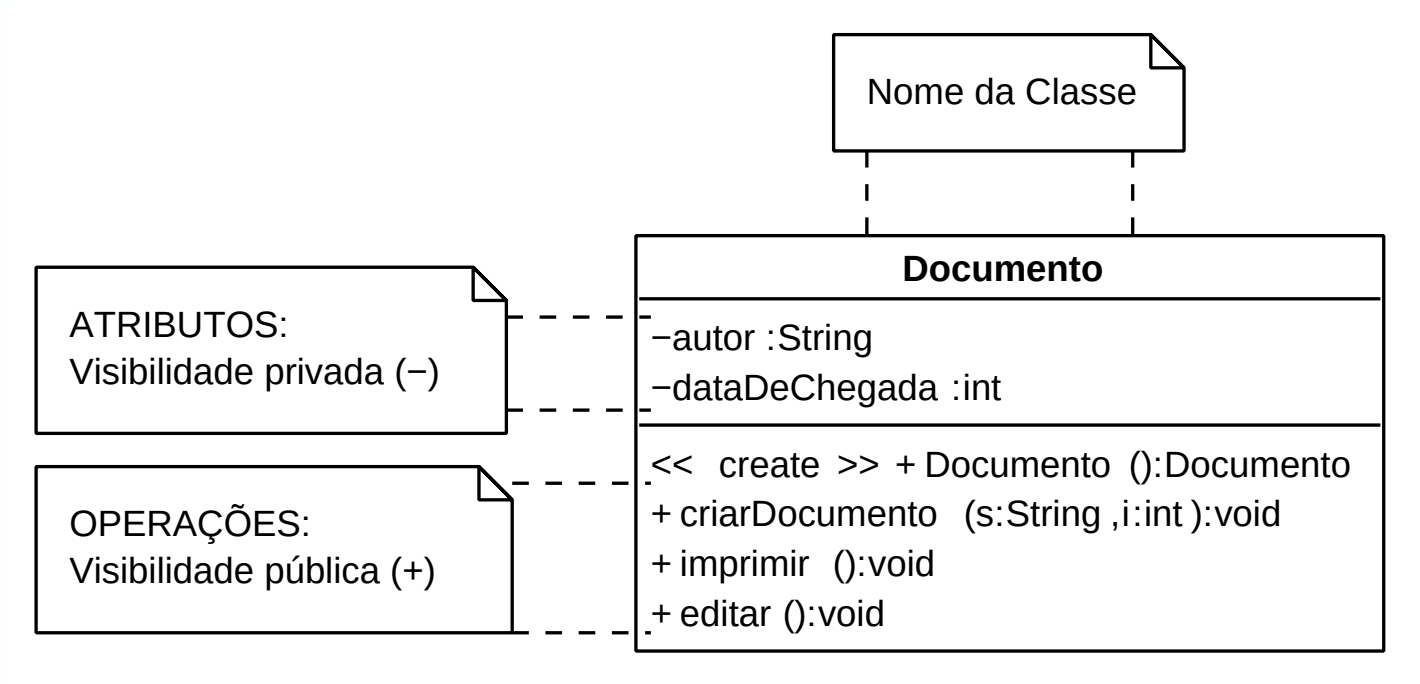
- De acordo com o conceito de TAD, uma classe deve definir um conjunto de atributos com visibilidade privada (nunca pública)
- A interface pública da classe é formada pelo conjunto de operações com visibilidade pública. A classe pode conter operações com visibilidade privada.

Dicas e Recomendações (II)

Uma classe é considerada bem estruturada quando:

1. Proporciona uma abstração bem definida de um elemento oriundo do vocabulário do domínio do problema ou da solução.
2. É formada por um conjunto pequeno e bem definido de responsabilidades que devem ser cumpridas pela classe.
3. Proporciona uma separação clara entre a especificação da abstração e a sua implementação.
4. É simples e fácil de entender, e ao mesmo tempo tem capacidade de extensão e adaptação.

Exemplo de Classe em UML



Declaração de uma Classe em Java

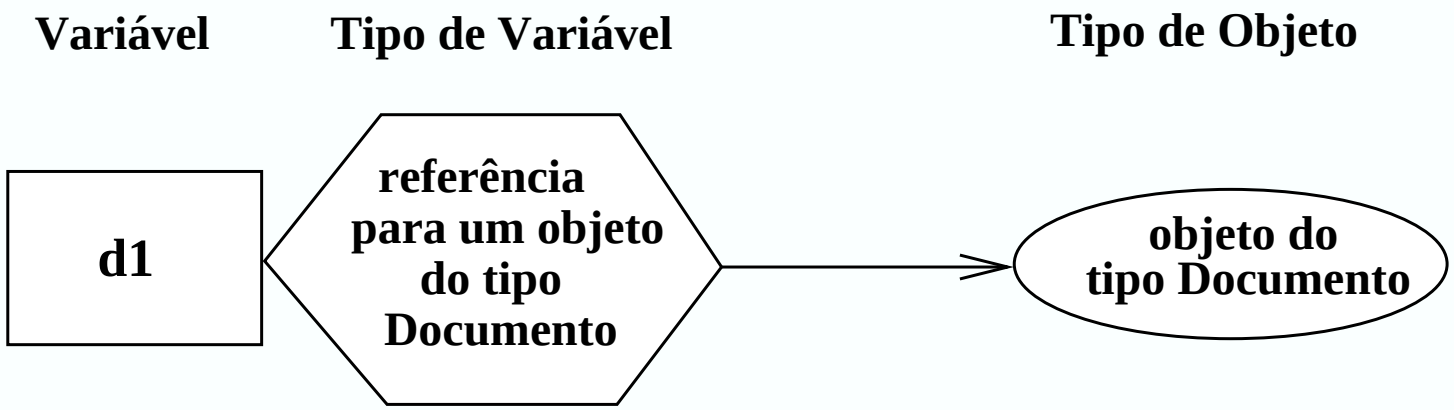
```
class Documento{
   private String autor;
   private int dataDeChegada;

   public void criarDocumento(String s, int i){
      autor = s; dataDeChegada = i;
   }
   public void imprimir(){
      System.out.println(``Imprime o Documento'');
   }
   public void editar(){
      System.out.println(``Edita o Documento'');
   }
} // fim da classe Documento
```

Criação de Objetos do tipo Documento(I)

```
Documento d1;  // declaração de uma referência para um
               // objeto do tipo Documento
d1 = new Documento(); // alocação dinâmica de memória
               // para a criação do objeto
d1.criarDocumento(‘‘Cecilia’’, 18032009);
d1.imprimir(); // envio de mensagem para o objeto
```

Criação de Objetos do tipo Documento(II)



Métodos Construtores (I)

- Um **construtor** é um método especial, ativado automaticamente no momento da criação de um objeto da classe
- Um método construtor possui nome idêntico ao da sua classe
- Ele não pode retornar nenhum resultado
  (nem mesmo void)

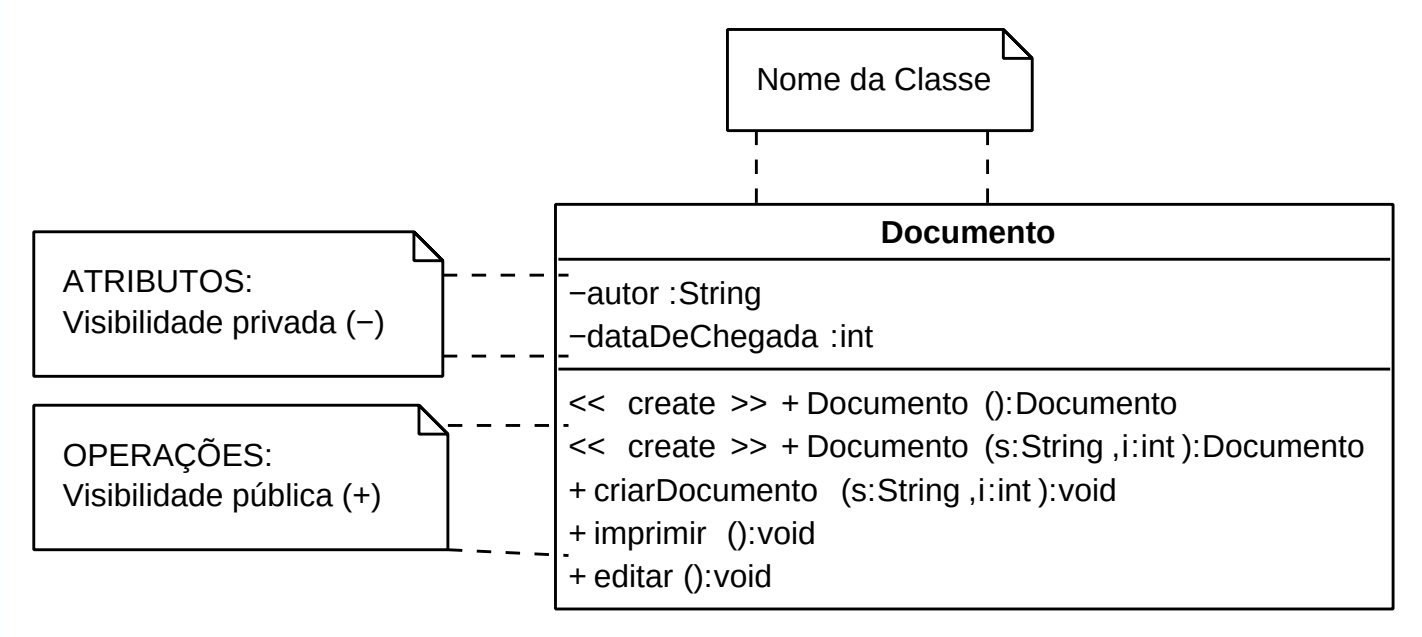
Métodos Construtores (II)

- Um método construtor pode atribuir valores iniciais para os atributos
- Java fornece um construtor padrão para cada classe que irá atribuir valores aos atributos; conforme suas definições de tipo
- Exemplo: o construtor padrão Documento irá preencher as variáveis numéricas com zero e as referências de objetos com "null" (se existirem)

Métodos Construtores (III)

- Os métodos construtores podem receber valores através de parâmetros
- Normalmente, os valores iniciais dos atributos são parametrizados
- Um método construtor com parâmetros é conhecido como **construtor não-padrão**

Classe Documento com Construtor Não-Padrão



Declaração de uma Classe em Java

```
// arquivo Documento.java
public class Documento{
   private String autor;
   private int dataDeChegada;

   public Documento(){/*valores ‘‘default’’*/}

   public Documento(String s, int i){
      autor = s;
      dataDeChegada = i;
   }

   public void criarDocumento(String s, int i){
      autor = s; dataDeChegada = i;
   }
```

```
    public void imprimir(){
        System.out.println(``Imprime o Documento'');
    }
    public void editar(){
        System.out.println(``Edita o Documento'');
    }
}
```

Criação de Objetos com Construtor

```
//arquivo Principal.java
class Principal {
        static public void main (String args[ ]){
           Documento d1;
           d1 = new Documento(‘‘Mary’’, 19032009);
           d1.imprimir();

           Documento d2;
           d2 = new Documento();
           d2.criarDocumento(‘‘Rubira’’, 20032009);
           d2.imprimir();
        }
} // fim da classe Principal
```

Compilando o seu Primeiro Programa Java

(0)- Certifique-se de que os *paths* do compilador e do interpretador Java estejam definidos; javac e java, respectivamente;

(1)- Crie um arquivo de nome Hello.java :

```
//arquivo Hello.java
class Hello {
static public void main (String args[ ]){
            System.out.println(‘‘Hello World’’);
        }
} // fim da classe Hello
```

(2)- Chame o compilador Java fornecendo o arquivo Hello.java como entrada (no caso, Java 6 SDK):
javac Hello.java

- O Java 6 (versão 1.6.0) está disponível em:
http://java.sun.com/javase/downloads/

Compilando o seu Primeiro Programa Java

(3)- Como o resultado da compilação, o arquivo Hello.class é criado no seu diretório; contendo o código executável Java ("bytecodes").

(4)- Execute o interpretador Java fornecendo o arquivo Hello.class como entrada (extensão .class deve ser omitida):
java Hello

(5)- Como resultado da execução; o programa imprime a seguinte mensagem na tela:
Hello World

Estágios envolvidos na compilação e execução de um Programa Java

```
(1) criar um programa Java --> Hello.java
(2) compilar o programa --> javac Hello.java
(3) saída do compilador em bytecode --> Hello.class
(4) executar o programa:  --> java Hello on machine X
                          --> java Hello on machine Y
                          --> etc
```

- O compilador Java é chamado **javac** e um programa Java é armazenando num arquivo **.java**.
- A compilação de um arquivo **.java** produz um arquivo **.class** para cada classe do arquivo.
- o arquivo **.class** contém os bytecodes que representam a versão traduzida do programa fonte Java.

- Uma vez que o programa tenha sido compilado com sucesso, ele pode ser dado para um interpretador de bytecodes para ser rodado.
- O interpretador de bytecodes da plataforma Java chama-se **java**.
- Um programa é rodado passando o nome da classe que contém o método **main** para o interpretador.
- Os bytecodes podem ser rodados em qualquer máquina para a qual esteja disponível um interpretador de bytecodes.

Utilizando a Classe Documento

(1)- Crie um arquivo de nome Principal.java :

```
//arquivo Principal.java
class Principal {
     static public void main (String args[ ]){
           Documento d1;
           d1 = new Documento(‘‘Mary’’, 02062005);
           d1.imprimir();
     }
} // fim da classe Principal
```

(2)- Chame o compilador Java fornecendo os arquivos Documento.java e Principal.java como entradas:

> **javac Documento.java**

> **javac Principal.java**

Utilizando a Classe Documento

- Como o resultado da compilação, os arquivos Documento.class e Principal.class são criados no seu diretório; contendo o código executável Java.
- Execute o interpretador Java fornecendo o arquivo Principal.class como entrada (extensão .class deve ser omitida): **java Principal**
- Como resultado da execução; o programa imprime a seguinte mensagem na tela:
  Imprime o Documento

O Conceito de Tipo (I)

- De forma geral, **tipo** é uma especificação de um conjunto de valores que podem ser associados com uma variável, junto com as operações que podem ser legalmente usadas para criar, acessar e modificar tais valores. Esse conjunto de operações é chamado **interface pública** do tipo.

Exemplo:

```
Type Boolean
valores: TRUE, FALSE
interface pública: AND, OR, NOT
```

O Conceito de Tipo (II)

No modelo de objetos,

- **Tipo** é definido como a coleção de todos os objetos do sistema que respondem do mesmo modo ao mesmo conjunto de mensagens
- **Tipo** é uma coleção de objetos com a mesma interface pública

O Conceito de Tipo (III)

A classe Documento oferece uma implementação concreta para a seguinte especificação do tipo Documento:

```
tipoDocumento = {criarDocumento(),
                 imprimir(), editar()}
```

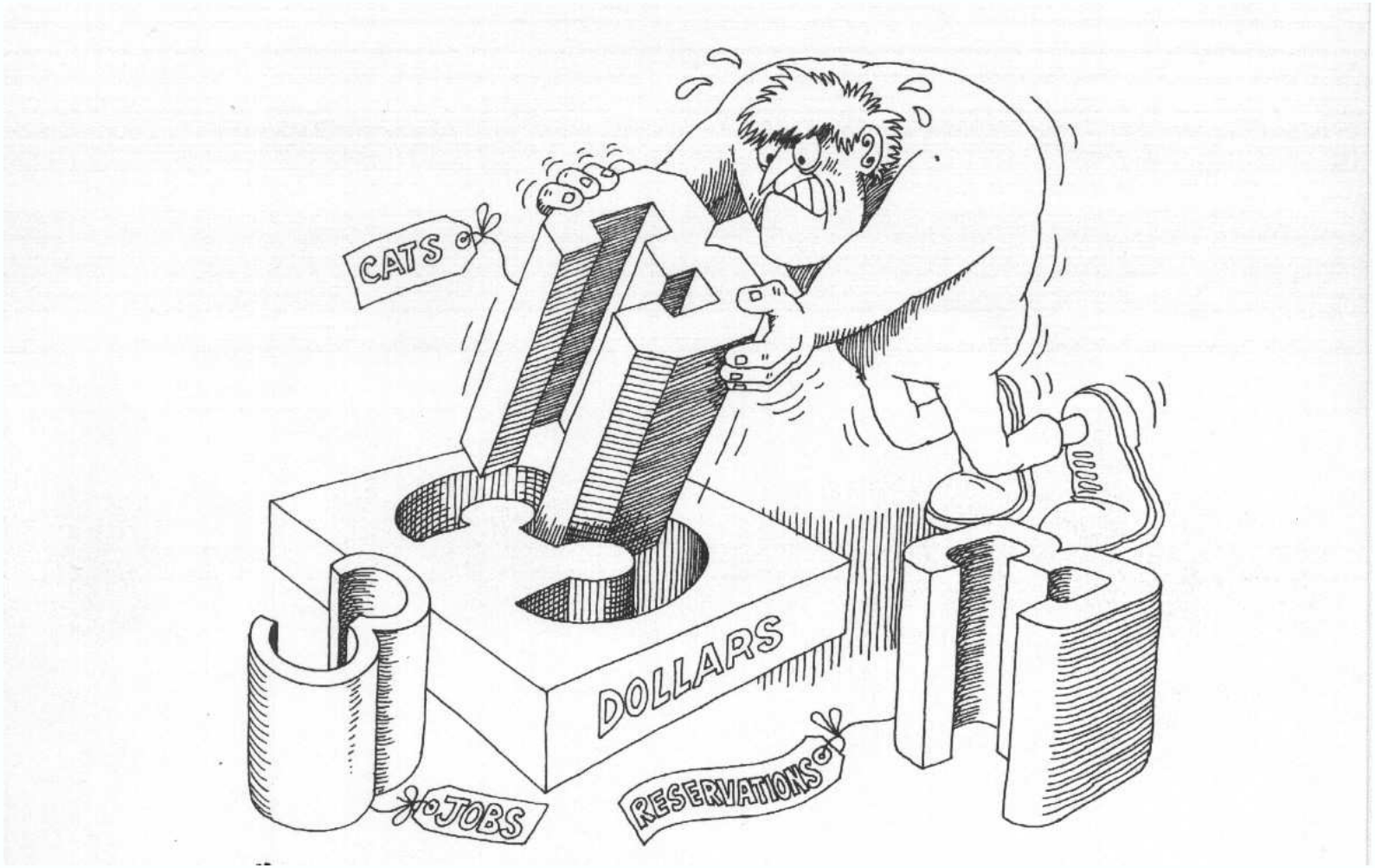
O Conceito de Tipo (IV)

- **Tipo** não é classe. Tipo é a descrição da interface de um objeto
- **Classe** especifica uma implementação particular de um tipo
- Java separa explicitamente a definição de tipo (palavra reservada interface) da sua implementação (palavra reservada class)
- C++, Eiffel e Modula-3 não oferecem essa separação explícita
- UML tem o conceito de "interface"

Deteção de Erros de Tipos (I)

- Existem duas possibilidades:
  - Em tempo de compilação
  - Em tempo de execução
- **Linguagens tipadas estaticamente:** são aquelas onde o tipo de uma variável referente a um objeto é definido em tempo de compilação. Todos os "erros de tipos" são pegos em tempo de compilação. Ex.: Java, C++, Eifell e Modula-3
  - Conhecidas como **"linguagens fortemente tipadas"**;

Deteção de Erros de Tipos (II)



Deteção de Erros de Tipos (III)

- **Linguagens tipadas dinamicamente:** são aquelas onde o tipo de uma variável referente a um objeto é conhecida em tempo de compilação mas pode ser mudado dinamicamente em tempo de execução
- Todos os erros de tipos são pegos em tempo de execução. Como conseqüência, a validade das operações podem ser verificadas em tempo de execução. Ex.: Smalltalk, Objective-C e Python.
  - Conhecidas como **"linguagens fracamente tipadas"**;

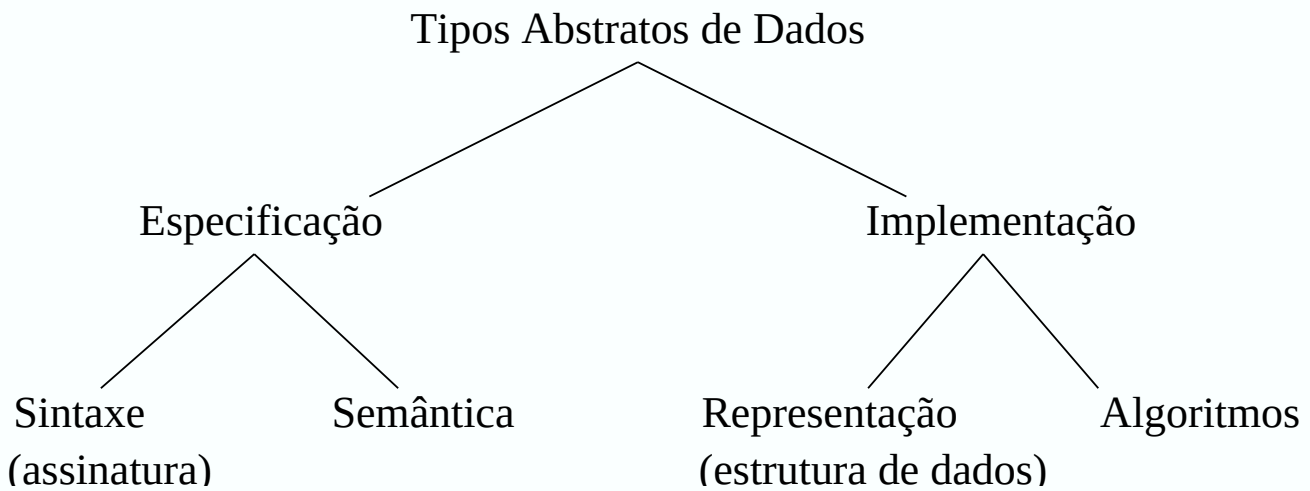
Verificação de Tipos (I)

- A verificação de tipos decide se uma ação é válida ou não e se ela levará a uma inconsistência de tipos. Ela não define como uma operação será executada, ou mais especificamente, qual é o código da operação que será executado. Esta é tarefa do **acoplamento** ("binding")
- Ela preocupa-se com a prevenção de inconsistências de tipos na linguagem através da eliminação de "erros de tipos"

Verificação de Tipos (II)

- Um "erro de tipo" é uma ação na linguagem de programação que resulta ou poderá resultar numa aplicação de uma operação inválida
- Existem duas fontes de erros de tipo:
  - Atribuição
  - Passagem de parâmetros

O Conceito de Tipo Abstrato de Dados (TAD) (I)



O Conceito de Tipo Abstrato de Dados (II)

- Separação de especificação e implementação: permite o uso do TAD sem conhecer nada sobre a sua implementação. Portanto, um TAD pode ter mais de uma implementação
- A **sintaxe** de um TAD é conhecida como o conjunto de **assinaturas** de operações que especifica a sua interface
- A especificação sintática de um TAD não é suficiente para descrever o comportamento; é necessário também algo para especificar a sua **semântica** de maneira independente da implementação

Assinatura de Operações

- O termo assinatura se refere aos requisitos de entrada e saída para procedimentos/funções.
- Assinatura é a lista dos tipos dos argumentos, onde a ordem e o número de vezes em que os tipos dos argumentos aparecem é relevante.
- O tipo do retorno não é parte da assinatura (C++).
- Ex: void print(int i, double x);
  void print(double x, int x);
  void print(int i);
  void print(int i, int j);

Exemplo de técnicas para a representação semântica de um TAD (I)

- **Especificação Algébrica:** consiste na definição de um conjunto de equações que interrelacionam as operações do TAD
- **Predicados:** são pré-condições, pós-condições e invariantes aplicadas às operações do TAD que são representadas utilizando expressões lógicas

Exemplo de técnicas para a representação semântica de um TAD (II)

- A especificação semântica determina o comportamento de uma operação. Incluindo o significado dos seus parâmetros, o significado do seu resultado de retorno, e o seu efeito no estado observável do objeto

Exemplo de técnicas para a representação semântica de um TAD (III)

- Os **predicados** são expressões lógicas que devolvem *"true"* ou *"false"*. Eles podem ser usados para expressar pré-condições, pós-condições e invariantes de uma operação
- As **invariantes** definem uma condição, relativa do estado observável de um objeto, que nunca pode ser violada pela operação. Uma invariante de classe é aplicada a todas as operações definidas pela classe

Exemplo de técnicas para a representação semântica de um TAD (IV)

- As **pré-condições** definem uma condição pré-existente que deve ser verdadeira para que uma operação possa ser executada
- As **pós-condições** definem uma condição final que deve ser assegurada depois que a operação é executada

Exemplo: TAD Pilha (I)

Especificação sintática

Consiste na especificação das seguintes operações:

- **Pilha():** o construtor não recebe parâmetros de entrada, não devolve nenhum resultado e cria uma pilha vazia
- **void empilha(int elem):** recebe um elemento para ser empilhado e não retorna nenhum resultado
- **int desempilha():** não recebe parâmetros de entrada e devolve um elemento do tipo integer

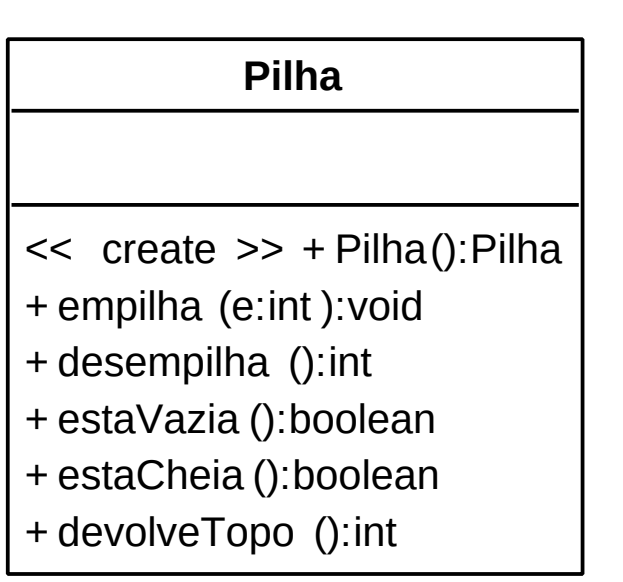
Exemplo: TAD Pilha (II)

Especificação sintática

Consiste na especificação das seguintes operações:

- **boolean estaVazia():** não recebe parâmetros de entrada e devolve *"true"* se a pilha está vazia e *"false"*, caso contrário
- **boolean estaCheia():** não recebe parâmetros de entrada e devolve *"true"* se a pilha está cheia *"false"*, caso contrário
- **int devolveTopo():** não recebe parâmetros de entrada; inspeciona o topo da pilha e devolve uma cópia do elemento como resultado de retorno

Exemplo: TAD Pilha (III)



Exemplo: TAD Pilha (IV)

```
class Pilha{
   //...
   // métodos:
   public Pilha() {...}
   public void empilha(int e) {...}
   public int desempilha() {...}
   public boolean estaVazia() {...}
   public boolean estaCheia() {...}
   public int devolveTopo() {...}
} // fim da classe Pilha
```

Exemplo: TAD Pilha (V)

Especificação semântica utilizando pré- e pós-condições:

Operação	Pré-condição	Pós-condição
empilha()	!estaCheia()	!estaVazia()
desempilha()	!estaVazia()	!estaCheia()
estaVazia()	TRUE*	Nenhuma mudança no estado da pilha
estaCheia()	TRUE	Nenhuma mudança no estado da pilha
devolveTopo()	!estaVazia()	Nenhuma mudança no estado da pilha
construtorPilha()	TRUE	estaVazia()

*TRUE significa que a operação pode ser chamada incondicionalmente.

Exemplo: TAD Pilha (VI)

Especificação algébrica usando equações:

Tem-se que para toda Pilha p e todo elemento e:

(1) - [Pilha()].estaVazia é verdadeiro.

(2) - [p.empilha(e)].estaVazia é falso.

(3) - [p.empilha(e)].desempilha() $==$ e AND [[p.empilha(e)].desempilha()] $\Rightarrow$ [p fica inalterado]]

(4) - [p.empilha(p.desempilha())] $\Rightarrow$ [p fica inalterado]

Estudo de Caso : Automação Bancária - (I)

Suponha a existência de um banco hipotético onde as contas correntes se comportam do seguinte modo:

(1) As contas são sempre individuais e a única informação necessária a respeito do cliente é o seu nome.

(2) As contas são identificadas através de números.

(3) As contas devem receber um depósito inicial no momento de sua abertura.

(4) Uma vez criada uma conta, ela pode receber lançamentos de crédito e débito.

Estudo de Caso : Automação Bancária - (II)

(5) O saldo de uma conta nunca pode ficar negativo. Qualquer pedido de saque superior ao saldo deve ser rejeitado.

(6) Se, durante a movimentação da conta, o saldo se igualar a zero, a conta torna-se inativa; não podendo mais ser reaberta.

(7) Deve ser prevista uma operação de consulta de saldo.

(8) As operações de débito e consultas só podem ser efetuadas se fornecida uma senha numérica validada pelo cliente. A senha da conta é determinada no momento de sua criação.

Definição do TAD ContaCor

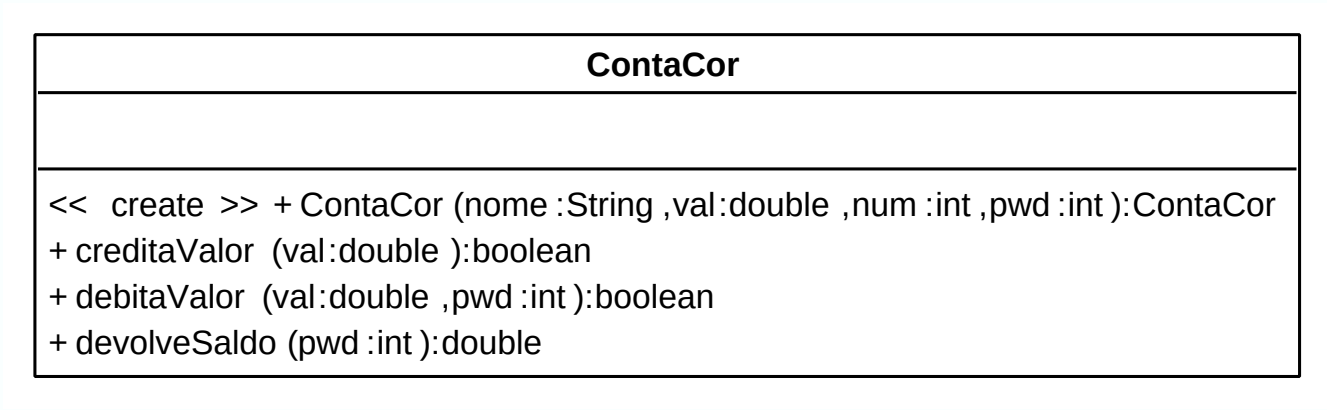
- Parte 1 - Especificação Sintática e Semântica
- Parte 2 - Implementação do TAD

Parte 1 - Especificação Sintática (I)

O TAD ContaCor fornece 4 operações cujas assinaturas são:

- **public boolean creditaValor(double val)** que recebe um valor a ser creditado. Resultado de retorno: status da execução: *"true"* para sucesso, *"false"* para insucesso.
- **public boolean debitaValor(double val, int pwd)** que recebe um valor a ser debitado e uma senha a ser validada. Resultado de retorno: status da execução: *"true"* para sucesso, *"false"* para insucesso.
- **public double devolveSaldo(int pwd)** que recebe uma senha a ser validada. Resultado de retorno: o saldo atual da conta.
- **public ContaCor(String nome, double val, int num, int pwd)** que recebe os valores iniciais de quatro atributos. Resultado de retorno: um objeto do tipo "ContaCor".

Parte 1 - Especificação Sintática (II)



Operação creditaValor(...)

- **Pré-Condições:** conta deve estar ativa e o valor a creditar deve ser maior que zero.
- **Pós-Condições:** valor a creditar é somado ao saldo da conta.
- **Invariante:** o estado da conta permanece inalterado durante a execução da operação, exceto pela alteração do saldo ao final.

Operação debitaValor(...)

- **Pré-Condições:** conta deve estar ativa, o valor a ser debitado deve ser maior que zero e deve ser menor ou igual ao saldo da conta, e a senha deve ser válida.
- **Pós-Condições:** valor a debitar é subtraído do saldo da conta, e se o saldo se tornar igual a zero, a conta deve se tornar inativa.
- **Invariante:** o estado da conta permanece inalterado durante a execução da operação, exceto pela alteração do saldo ao final.

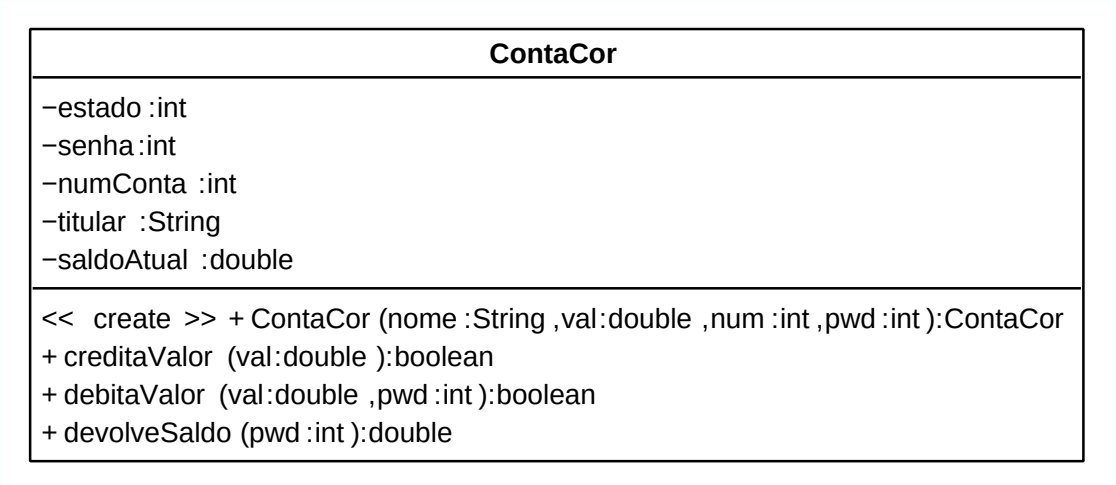
Operação devolveSaldo(...)

- **Pré-Condições:** conta deve estar ativa, e a senha fornecida deve ser válida.
- **Pós-Condições:** o valor do saldo é devolvido, e o estado da conta permanece inalterado.
- **Invariante:** o estado da conta permanece inalterado.

Operação ContaCor(...)

- **Pré-Condições:** o valor do atributo saldo deve ser maior que zero ($saldo > 0$).
- **Pós-Condições:** um objeto do tipo `ContaCor` representando uma conta ativa, com número `num`, saldo inicial `val`, titular igual a `nome` e senha igual a `pwd`.
- **Invariante:** o número de contas inativas permanece inalterado.

Parte 2 - Implementação do TAD



Implementação dos Atributos

```
// arquivo ContaCor.java
class ContaCor {
   private int estado; // 1 = ativo ; 2 = inativo
   private int senha;
   private int numConta;
   private String titular;
   private double saldoAtual;
   public ContaCor(String nome, double val,
                   int num ,int pwd) { ...}  }
   public double devolveSaldo(int pwd) { ...}
   public boolean creditaValor(double val) { ...}
   public boolean debitaValor(double val, int pwd) { ...}
} // fim da classe ContaCor
```

Implementação do Comportamento (I)

```
public ContaCor(String nome, double val,
                int num, int pwd) {
    titular = nome;
    numConta = num;
    senha = pwd;
    saldoAtual = val;
    estado=1; //conta é ativada quando criada
} // fim do construtor

public double devolveSaldo (int pwd) {
    if (estado!=1) return(-1); // conta deve estar ativa
    if (pwd!=senha) return(-1); //senha deve ser válida
    return(saldoAtual);
} // fim de devolveSaldo( )
```

Implementação do Comportamento (II)

```
public boolean creditaValor (double val) {
   if (estado!=1)
      return(false); //conta deve estar ativa
   if (val<=0)
      return (false); // val>0;
   saldoAtual+= val; // credita valor;
   return(true); // operação terminada com sucesso
} // fim de creditaValor( )
```

Implementação do Comportamento (III)

```
public boolean debitaValor (double val, int pwd) {
   if (estado!=1)
      return(false); // conta deve estar ativa
   if (val<=0)
      return (false); // val>0;
   if (pwd!=senha)
      return (false);//senha deve ser válida
   if (val>saldoAtual)
      return (false);// val<= saldoAtual
   saldoAtual -= val; //debita valor
   if(saldoAtual ==0)
      estado=2;// se saldo=0, torna conta inativa
   return(true);
} // fim de debitaValor( )
```

Usando o TAD ContaCor

```
//arquivo Exemplo.java
class Exemplo{
   public static void main(String args[]){
      ContaCor c1 = new ContaCor(‘‘Ursula’’, 500, 1, 1);
      c1.creditaValor(100);
      c1.debitaValor(50 ,1);
      System.out.println(‘‘Saldo final = ’’ +
         c1.devolveSaldo(1));
   } } // fim da classe Exemplo
```

Rodando o programa:

```
> javac ContaCor.java
> javac Exemplo.java
> java Exemplo
```

Saída impressa na tela : $Saldo\ final = 550$

Exemplo 1 - Contador - (I)

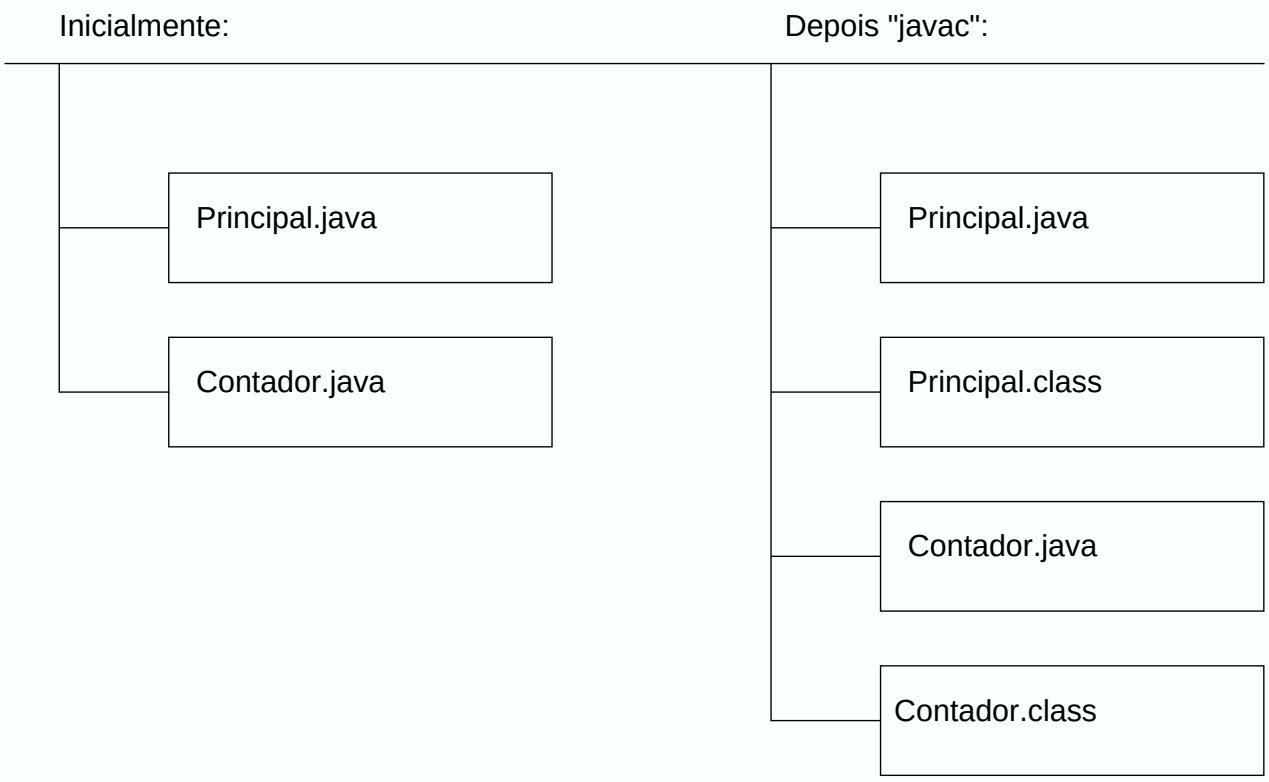
```
// arquivo Contador.java
public class Contador{
   private int num; // valor do contador
   public void incrementa ( ) {
      num = num +1; }
   public void decrementa ( ) {
      num = num -1; }
   public void comeca(int num) {
      this.num = num; }
   public int retornaNum( ) {
      return this.num; }
} // fim da classe Contador
```

OBS.: `this` significa "o objeto para qual este trecho de código está sendo executado"

Exemplo 1 - Contador - (II)

```
// arquivo Principal.java
class Principal{
   static public void  main (String args[ ]) {
      Contador umCont;
      umCont = new Contador( );
               //alocação dinâmica de memória
      umCont.comeca(0);
      System.out.println(umCont.retornaNum());
      umCont.incrementa( );
      System.out.println(umCont.retornaNum());
   }
} // fim da classe Principal
```

Exemplo 1 - Contador - (III)



Exemplo 2 - Círculo - (I)

```
// arquivo Circulo.java
public class Circulo{
   private float raio;
   private float x; // posição do centro do círculo
                    // em coordenadas cartesianas
   private float y;
   public Circulo (float r, float ax, float ay ) {
      raio = r;
      x = ax;
      y = ay;
   }
   ...
```

Exemplo 2 - Círculo - (II)

```
// arquivo Circulo.java
   ...
   public void move(float dx, float dy) {
     x = x + dx;   y = dy + y;}

   public void mostra ( ){ // imprime o estado do objeto
     System.out.println(''('' + x + '', '' + y + '', ''
                                + raio + '')'');}

   public void alteraRaio(float a) {
     raio = a; }
} // fim da classe Circulo
```

Exemplo 2 - Círculo - (III)

```
// arquivo Principal.java
class Principal{
   static public void main (String args[ ]) {
      Circulo umCirc;
      umCirc = new Circulo((float)1.0,
                           (float)0.0,
                           (float)0.0);
      umCirc.mostra( );
      umCirc.move((float)10.0, (float)10.0 );
      umCirc.mostra ( );
   }
} // fim da classe Principal
```

OBS.: Na tela é impresso:

```
(0, 0, 1)
(10, 10, 1)
```