Tópicos em Linguagem de Programação

Prof. Tiago Eugenio de Melo, MSc

tiago@comunidadesol.org

www.tiagodemelo.info

Sumário

- Introdução
- Objetos e Classes
- Herança
- Polimorfismo
- Acoplamento Dinâmico
- Pacotes
- Construtores
- Uso de Herança
- Uso de Polimorfismo

Introdução

- Estruturação de software
  - Sistemas de software modernos requerem um alto grau de confiabilidade, disponibilidade e tolerância a falhas.
  - Sistemas de software são intrisicamente complicados e têm se tornado ainda mais complicados com os novos requisitos impostos pelas aplicações modernas:
    - Alta confiabilidade.
    - Alto desempenho.
    - Desenvolvimento de software rápido e barato.
    - Complexidade e tamanhos grandes.

Introdução

- A chave do sucesso para um promissor desenvolvimento de software está no controle de sua complexidade.
- Necessidade de melhores ferramentas e metodologias para gerenciar a complexidade.
- Técnicas para redução da complexidade:
  - Particionamento do sistema em partes que sejam muito bem limitadas (modularidade).
  - Representação do sistema como uma hierarquia.
  - Maximização da independência entre as partes do sistema (baixo acoplamento e alta coesão).

Introdução

- Na Crise de Software (1968) foi possível identificar vários problemas na produção de software:
  - Pouco predizível.
  - Baixa qualidade.
  - Alto custo de manutenção.
  - Muita duplicação de esforços.
- Criação de diversas metodologias de desenvolvimento.
- Surgimento do Paradigma Orientado a Objetos.

Introdução

- Objetos em software: idéias básicas
  - A característica mais importante (e diferente) da abordagem orientada a objetos é a unificação de dois elementos: **dados** e **funções**.

Objeto = dados (privados) + funções (públicas)

  - A unificação desses dois elementos deu origem ao que chamamos de **encapsulamento**.

Introdução

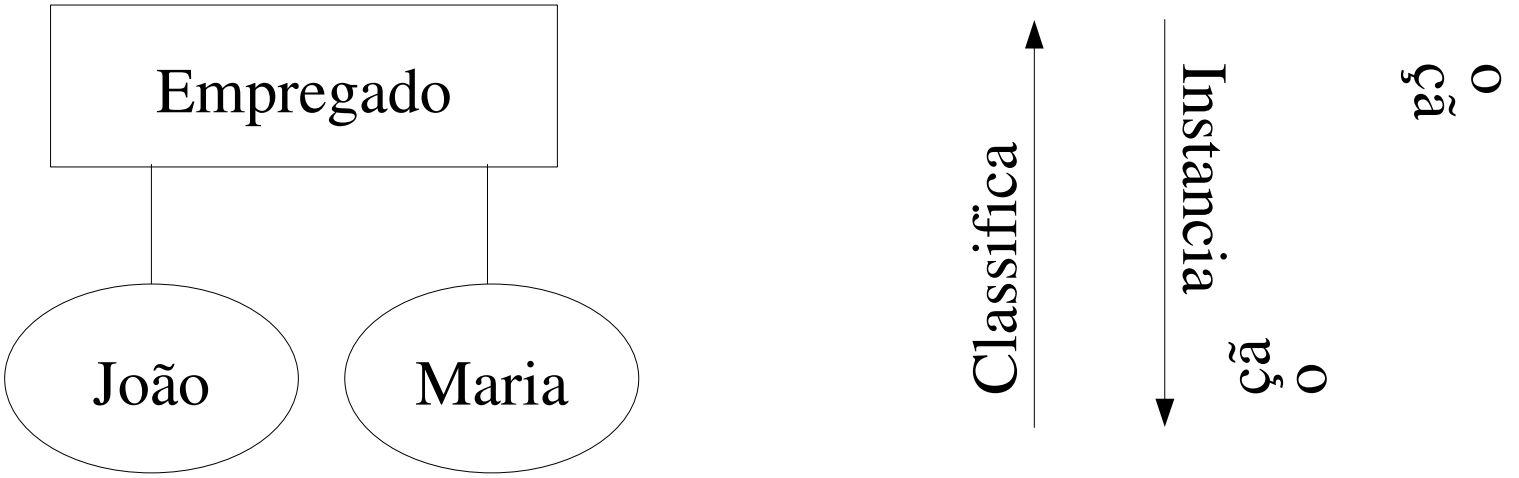
- Abstração de Dados
  - Uma das tarefas mais importantes em desenvolvimento de software é a análise do domínio do problema e a modelagem das entidades e fenômenos relevantes para a aplicação.
  - A modelagem conceitual envolve dois aspectos relevantes:
    - Abstração.
    - Representação.
  - Abstração é definida como um mecanismo através do qual nós observamos o domínio de um problema e focamos nos objetos e ações que são relevantes para uma aplicação específica, ignorando todos os outros pontos que são irrelevantes.

Introdução

- A abstração envolve três operações:
  - Classificação/Instanciação.
  - Generalização/Especialização.
  - Agregação/Decomposição.

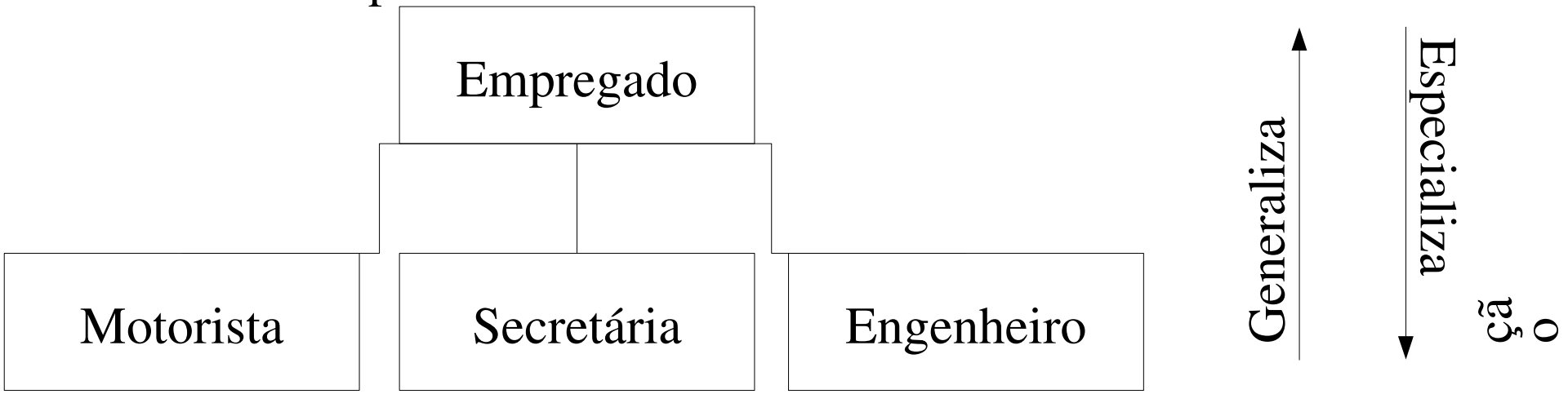
Introdução

- Classificação/Instanciação
  - Permite agrupar objetos similares em uma mesma categoria.
  - Exemplo



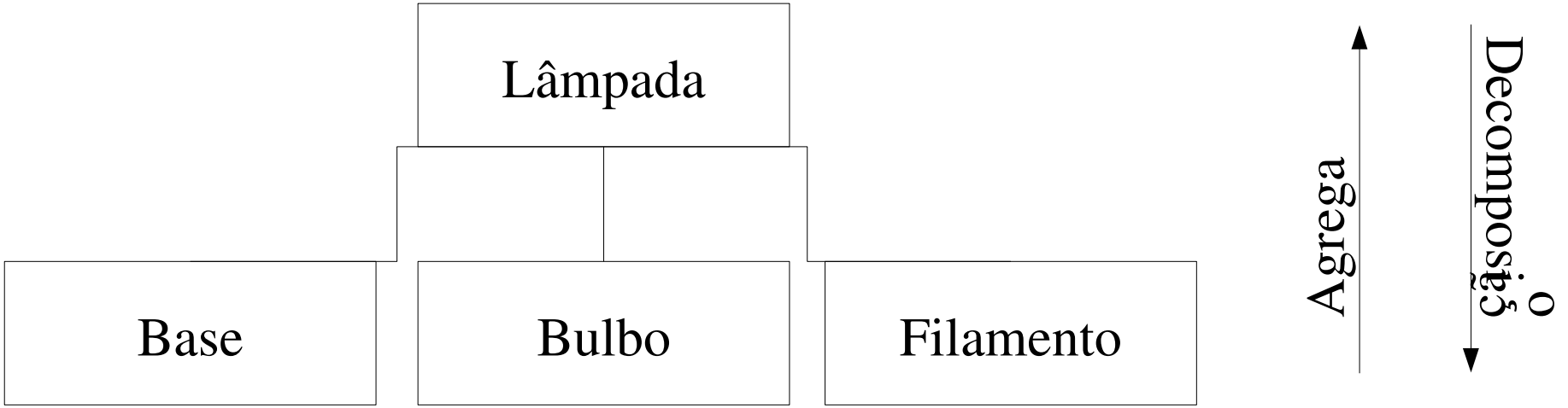
Introdução

- Generalização/Especialização
  - A generalização permite que todas as instâncias de uma categoria específica sejam também consideradas instâncias de uma categoria mais abrangente.
  - O inverso não é necessariamente válido.
  - Exemplo:



Introdução

- Agregação/Decomposição
  - Agregação é uma abstração que permite a identificação das categorias constituintes de outras categorias.
  - Ela é um mecanismo que forma o todo a partir das partes. Exemplo:



Introdução

- Agregação vs Generalização
  - Agregação e generalização são noções completamente diferentes e complementares.
  - Elas são conceitos ortogonais.
  - Nas hierarquias de generalização/especialização a noção de herança está implicitamente representada, enquanto que nas hierarquias de agregação/decomposição isso nem sempre é verdade.

Introdução

- O modelo de objetos proporciona modularidade, trazendo os seguintes benefícios:
  - Reusabilidade: software pode ser escrito a partir de componentes já existentes que podem ser usados em diversas aplicações.
  - Extensibilidade: novos componentes de software podem ser desenvolvidos a partir de componentes já existentes sem afetar os componentes originais.

Introdução

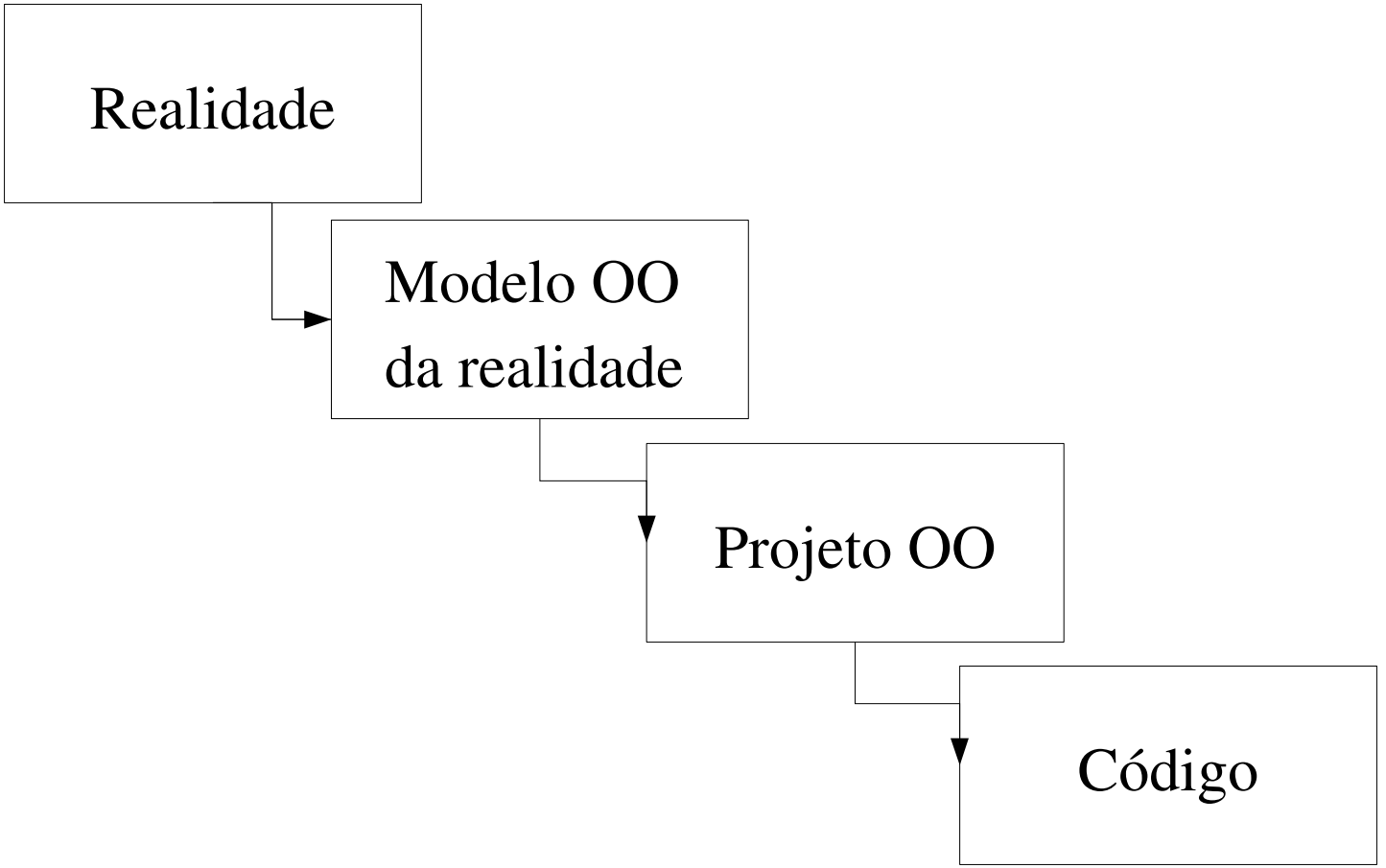
- De acordo Meyer, existem cinco critérios para a obtenção de modularidade:
  - Decomposibilidade: particionamento de um sistema unidades manejáveis.
  - Composibilidade: módulos podem ser livremente combinados em outros sistemas.
  - Entendimento: a compreensão de uma parte contribui para o entendimento do todo.
  - Continuidade: pequenas mudanças no sistema implicam em pequenas mudanças no comportamento.
  - Proteção: Condições excepcionais ou errôneas são confinadas aos subsistemas nas quais elas ocorrem ou afetam apenas as partes diretamente a elas relacionadas.

Introdução

- Linguagens de Programação Orientadas a Objetos
  - Para transformar as abstrações do mundo diretamente nas abstrações de uma linguagem de programação é necessário a passagem por passos intermediários.

Introdução

- Passos intermediários da modelagem da realidade



Introdução

- Evolução da abstração em linguagens de programação
  - No início do desenvolvimento das linguagens de programação, as linguagens de montagem permitiam aos projetistas escrever programas baseados em instruções de máquina (operadores) que manipulavam conteúdo de memória (operandos).
  - Fortran e Cobol introduziram a noção de subprogramas – funções e procedimentos.
  - Algol-60 introduziu o conceito de estrutura de bloco, procedimento etc.
  - Desde os anos 70, as linguagens de programação têm se preocupado em dar mais suporte para programação em larga escala. Surgindo o conceito de módulo.

Introdução

- Um módulo é uma unidade de programa que pode ser implementado de uma maneira mais ou menos independente dos outros módulos.
- Em 1972, surge o conceito de ocultamento da informação, também conhecido como encapsulamento.
- A idéia era encapsular variáveis globais em um módulo juntamente com o grupo de operações que tinham acesso direto a elas.

Introdução

- Somente **o que** o módulo faz é passado para o cliente do módulo; **o como** é implementado somente diz respeito ao implementador do módulo.
- A comunicação entre módulos é feita através de interfaces bem definidas que previnem o acesso direto a estruturas de dados de dentro de um módulo.

Introdução

- Existem dois mecanismos de linguagens de programação que dão apoio a essas idéias:
  - Módulo.
  - Tipo abstrato de dados.
- Um módulo consiste de duas partes:
  - Especificação do módulo (spec).
  - Implementação do módulo (body).
- Spec é um conjunto de declarações de estruturas de dados e assinaturas de procedimentos.
- Body contém as implementações da entidade declaradas na parte spec.

Introdução

- A noção de tipos abstratos de dados refere-se ao encapsulamento de uma estrutura de dados juntamente com as operações que manipulam essas estruturas dentro de uma região protegida.

Introdução

- Breve história de orientação a objetos

1967	Simula 67
1972	Artigo de Dahl sobre ocultamento da informação
1976	Primeira versão de Smalltalk
1983	Primeira versão de C++
1988	Primeira versão de Eifel
1995	Primeira versão de Java

Introdução

- Programação Orientada a Objetos
  - Programação orientada a objetos é um modelo de programação baseado em conceitos, tais como objetos, classes, tipos, ocultamento de informações, herança e polimorfismo.
  - A essência da programação orientada a objetos é a resolução de problemas baseada na identificação de objetos do mundo real pertencentes ao domínio da aplicação e o processamento requerido por esses objetos e, então, na criação de simulações deles.

Introdução

- Este estilo de programação tem as seguintes características positivas:
  - Modularidade.
  - Suporte explícito para generalização/especialização.
  - Visão unificada de dados e processos.
  - Atividade incremental e evolucionária.
  - Reusabilidade.
- Muito do interesse no modelo de objetos é devido à crescente percepção da indústria de que ele é uma maneira melhor de construir programas complexos, pois consegue promover a reutilização de software, melhorando a sua qualidade e reduzindo o seu custo de desenvolvimento.

Introdução

- Linguagens Orientadas a Objetos vs Linguagens Baseadas em Objetos
  - Uma linguagem é baseada em objetos quando ela dá apoio explícito somente ao conceito de objetos.
  - Uma linguagem é baseada em classes quando ela dá apoio para a criação de objetos e existe um mecanismo de agrupamento de objetos para a criação de novas classes, mas não dá suporte para um mecanismo de herança.
  - JavaScript é um exemplo de linguagem baseada em objetos.

Introdução

- Uma linguagem é dita orientada a objetos quando ela proporciona suporte lingüístico para objetos, requer que esses objetos sejam instâncias de classes e, além disso, oferece suporte para um mecanismo de herança.
- Resumindo:

Linguagem Orientada a Objetos = Objetos + Classes + Herança

Introdução

- Linguagens Orientadas a Objetos vs Linguagens Procedurais
  - C é um exemplo de linguagem que é orientada a ações, enquanto Java é orientada a objetos.
  - A unidade de programação em C é a função/procedimento, enquanto que em Java é a classe.
  - No paradigma procedural usam-se os verbos para identificação dos requisitos, enquanto que no paradigma orientado a objetos usam-se os substantivos.

Introdução

- A tabela abaixo traça um paralelo entre a abordagem procedural e orientada a objetos.

Paradigma Procedural	Paradigma de Objetos
tipos de dados	classes/tipos abstratos de dados
variável	objeto/instância
função/procedimento	operação/serviço
chamada de função	envio de mensagem

Introdução

- Erros comuns quando se conhece a Orientação a Objetos:
  - Pensar na POO simplesmente como uma linguagem
    - Supor que se está programando de maneira orientada a objetos simplesmente porque usa uma linguagem orientada a objetos.
    - A verdadeira POO é um estado da mente que exige que o programador veja seus problemas como um grupo de objetos e use encapsulamento, herança e polimorfismo.

Introdução

- Erros comuns quando se conhece a Orientação a Objetos:
  - Medo de reutilização
    - Os programadores gostam de criar e muitas vezes não consideram interessante reutilizar um componente. Com isso, perdem a oportunidade de criar algo maior e de maneira mais eficiente.
    - Muitos programadores não confiam no software que não escreveram. Um bom programa de teste pode resolver este medo.

Introdução

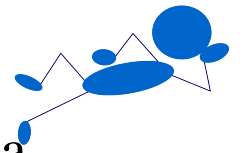
- Erros comuns quando se conhece a Orientação a Objetos:
  - Pensar na OO como uma solução para tudo
    - Apesar das vantagens da POO, existem situações em que não se deve usar OO.
    - Deve-se ter bom senso na escolha da ferramenta correta para o trabalho a ser feito.
    - O projeto não terá sucesso automaticamente, apenas porque foi usada uma linguagem OO.

Introdução

- Erros comuns quando se conhece a Orientação a Objetos:
  - Programação egoísta
    - Não seja egoísta, assim como o programador deve aprender a reutilizar, também deve aprender a compartilhar o código que cria.
    - Interfaces limpas e inteligíveis e documentação serão fatores positivos para reutilização.

Exercícios

- Explique duas técnicas utilizadas para redução da complexidade do processo de desenvolvimento de software.
- O que você entende por encapsulamento? Dê um exemplo.
- O que você entende por abstração? Como ela pode ser aplicada no desenvolvimento de software?
- Qual é a diferença entre generalização e especialização? Dê um exemplo de cada.
- O que é tipo abstrato de dados?
- Comente duas características essenciais das linguagens de programação orientadas a objeto.
- Qual é a diferença entre uma linguagem orientada a objetos e uma linguagem baseada em objetos? Dê um exemplo de cada.



Objetos e Classes

- Abstração
  - 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.
  - Pode-se dizer que abstração de dados (ou tipo abstrato de dados) proporciona uma abstração sobre uma estrutura de dados em termos de uma interface bem definida.

Objetos e Classes

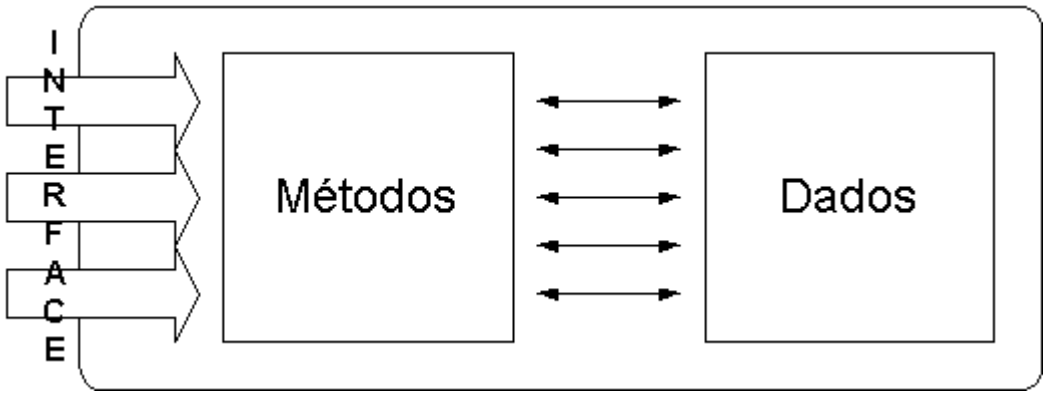
- Vantagens:
  - O fato do código e da estrutura de dados de uma abstração estarem armazenados num mesmo lugar cria um programa bem estruturado e legível, que pode ser facilmente modificável.
  - O aspecto do ocultamento da informação proporciona um nível de proteção contra acessos inesperados à estrutura de dados, o que mantém a integridade do objeto.
- Existe forte semelhança entre essas vantagens e as linguagens orientadas a objetos.

Objetos e Classes

- Objetos
  - Um objeto é uma entidade que possui um estado que pode ser alterado ao longo do seu tempo de vida.
  - O seu estado interno pode ser alterado através do envio de mensagens.
  - O envio de uma mensagem por um objeto implica na execução de uma operação correspondente da interface pública do objeto recipiente.
  - A única maneira de outras comunidades se comunicarem com outros objetos é através do envio de mensagens.

Objetos e Classes

- Portanto, um objeto encapsula ambos (i) estado (dados) e (ii) métodos (código) que podem acessar dados. Ilustração do conceito de objeto:



- Objetos também possuem um comportamento bem definido e uma identidade que é única.

Objetos e Classes

- O comportamento define como um objeto age e reage em termos de suas mudanças de estado e passagem de mensagens.
- Na teoria de tipos abstratos de dados, o termo comportamento é usado para denotar a interface abstrata (ou interface pública) de um objeto.
- Operação é definida como sendo alguma ação que um objeto realiza sobre um outro objeto para elicitar uma reação.
- Identidade é a propriedade de um objeto que o distingüe dos demais objetos.
- Em resumo, um objeto possui: (i) um estado, (ii) um

Objetos e Classes

- Classes
  - Uma classe é uma descrição de um molde que especifica as propriedades e o comportamento de um conjunto de objetos similares.
  - Todo objeto é instância de apenas uma classe.
  - Atributos são propriedades nomeadas de um objeto e que armazenam os estados abstratos de cada objeto.
  - Operações caracterizam o comportamento de um objeto e são o único meio de acessar, manipular e modificar os atributos de um objeto.
  - Métodos são as implementações das operações que compõem a interface pública de um objeto.

Objetos e Classes

- Exemplo de declaração de uma classe em C++:

```
class Documento {
private:
char *autor[100]; int dataDeChegada;
protected:
void iniciaAutor(char *nome);
void iniciaData(int num);
public:
Document( ); //construtor
~Documento( );//destrutor
char *devolverAutor( ); int devolverData( );
imprimir( ); editar( );
};
```

Objetos e Classes

- Exemplo de método em C++:

```
int Documento::devolveData( )
{
    return dataChegada;
}
```

- Criação e deleção de objetos:

```
Documento *pdoc; //declara um apontador p/ um objeto
                    documento
pdoc = new Documento;
```

  - Sempre que um objeto é destruído através do comando delete, o destrutor da classe é chamado, no exemplo, `~Documento()`.

Objetos e Classes

- Variáveis vs. Objetos
  - Em linguagens imperativas, dados são usualmente acessados através de variáveis.
  - Objetos contém dados, enquanto que variáveis contém referências para objetos; elas não são objetos.
  - Na maioria das linguagens orientadas a objetos, variáveis são apenas nomes que contêm referências para objetos; elas não armazenam dados diretamente nelas mesmas.
  - A linguagem C++ é um exemplo de exceção, pois permite que objetos sejam criados de duas formas diferentes: (i) criação estática e (ii) criação dinâmica.
  - A justificativa é que se tornam mais lentos.

Objetos e Classes

- Conceito de Tipo
  - 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.
  - Exemplo:
    - O tipo *Boolean* é associado aos valores *true* e *false* e às operações AND, OR e NOT.
  - Em orientação a objetos, tipo é uma coleção de objetos com mesma interface pública.
  - Tipo não é classe, embora muitos autores usem esses termos como sinônimos.

Objetos e Classes

- Tipo é essencialmente uma descrição de interface.
- Um classe especifica uma particular **implementação** de um tipo.
- A verificação de tipos decide se uma determinada operação é válida ou não sobre um objeto e se ela levará a uma inconsistência de tipos.

Objetos e Classes

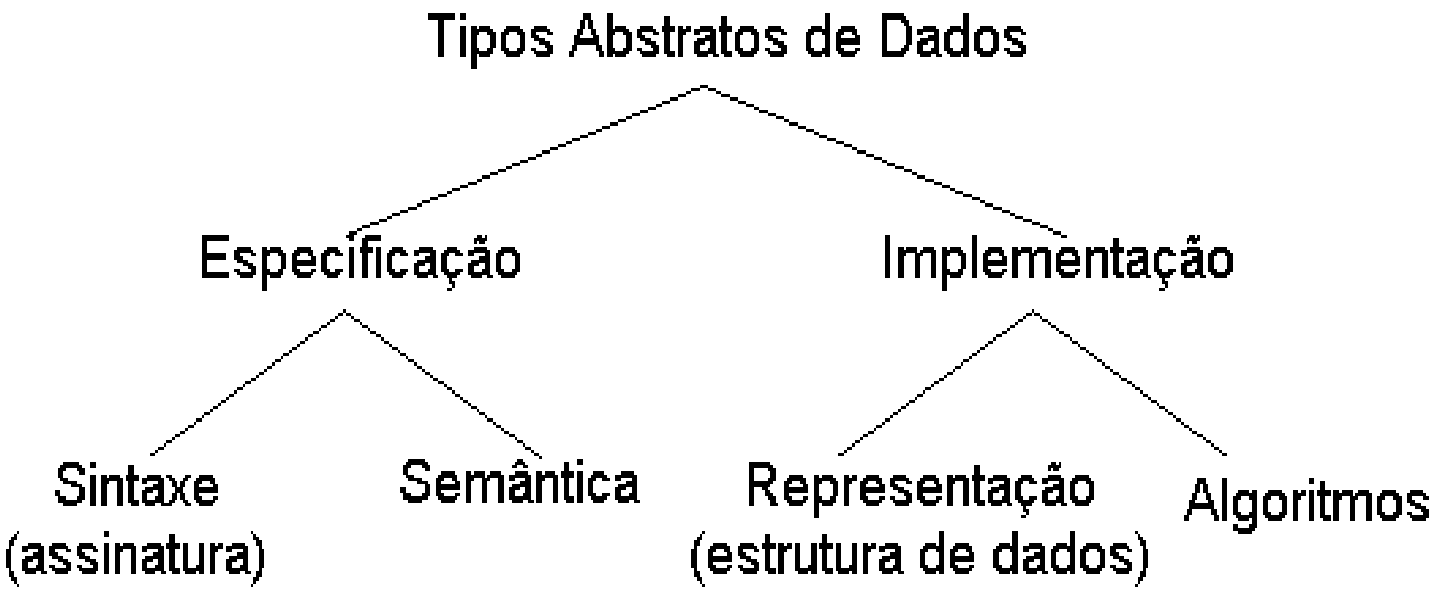
- Existem dois estilos de linguagens em relação a isso:
  - Linguagens tipadas estaticamente. São aquelas onde o tipo de uma variável que aponta para um objeto é definido em tempo de compilação.
  - Linguagens tipadas dinamicamente. São aquelas onde o tipo de uma variável referente a um objeto é conhecida em tempo de compilação, mas pode ser mudado dinamicamente em tempo de execução.

Objetos e Classes

- Tipos Abstratos de Dados
  - Um tipo abstrato de dados é um tipo acrescido com a noção de ocultamento da informação.
  - São consituídos de duas partes:
    - Parte da especificação.
    - Parte da implementação.

Objetos e Classes

- Estrutura de um tipo abstrato de dados



Objetos e Classes

- Qual é a vantagem dessa separação?
  - É que um cliente pode fazer uso do tipo abstrato de dados sem conhecer nada sobre a implementação.
- E qual conseqüência pode ser observada?
  - Nada impede que um tipo abstrato de dados tenha mais de uma implementação.
- A sintaxe de um tipo abstrato de dados é normalmente conhecida como sua assinatura que define a sua interface.
- A semântica é obtida através de especificação algébrica ou na provisão de pré e pós condições (uso de predicados).

Objetos e Classes

- Porém poucas são as linguagens que permitem a especificação de predicados associados a métodos. Eiffel é um exemplo.

Objetos e Classes

- Módulos/Pacotes
  - Módulos/Pacotes agrupam as classes relacionadas.
  - Eles definem restrições de acesso do seu conteúdo para clientes fora do módulo.
  - Módulos utilizam o conceito de tipo abstrato de dados, possuindo duas partes:
    - Especificação.
    - Implementação.

Objetos e Classes

- Encapsulamento
  - É definido como sendo uma técnica para minimizar as interdependências entre módulos programados independentemente, através de interfaces externas restritas.
  - Benefícios:
    - Manutenção.
    - Evolução do software.

Objetos e Classes

- As classes podem ser derivadas de três formas:
  - Clientes por instanciação: são os que criam instâncias da classe e manipulam essas instâncias através dos métodos.
  - Clientes por herança: são as subclasses que herdam os métodos e estrutura da classe.
  - Clientes por módulo: são as classes que estão dentro do mesmo módulo/pacote.

Objetos e Classes

- Existem quatro opções definidas:
  - Pública. Qualquer um dos três tipos de clientes pode fazer acesso, manipular e invocar diretamente atributos e métodos declarados como *public*.
  - Privada. Atributos e métodos declarados como private são acessíveis somente dentro da própria classe. Não estão disponíveis para clientes por instanciação, herança ou módulo.
  - Visível pela subclasse. Se um atributo ou método é declarado como protected, somente os clientes por herança têm acesso a eles.
  - Acesso ao Módulo/Pacote. Sem a especificação do controle de acesso, atributos e métodos são declarados como friendly.

Herança

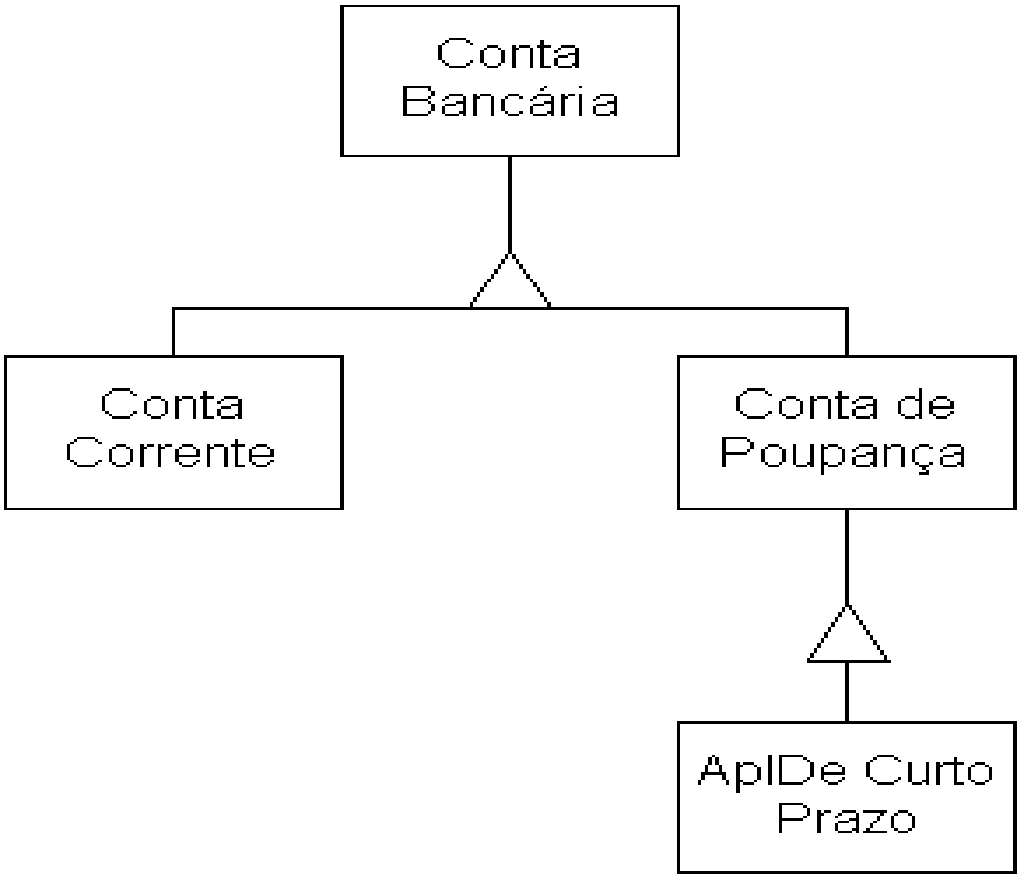
- O que é herança?
  - Mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento.
  - Uma classe derivada herda a representação de dados e operações de sua classe base.
  - Além disso, pode adicionar novas operações, estender a representação de dados ou redefinir a implementação de métodos já existentes.

Herança

- Terminologia:
  - Classe derivada ou subclasse ou classe filha: é uma classe que herda parte dos seus atributos e métodos de outra classe.
  - Classe base ou superclasse ou classe pai: é uma classe a partir da qual classes novas podem ser derivadas.

Herança

- Exemplo de hieraquia de classes de Conta Bancária

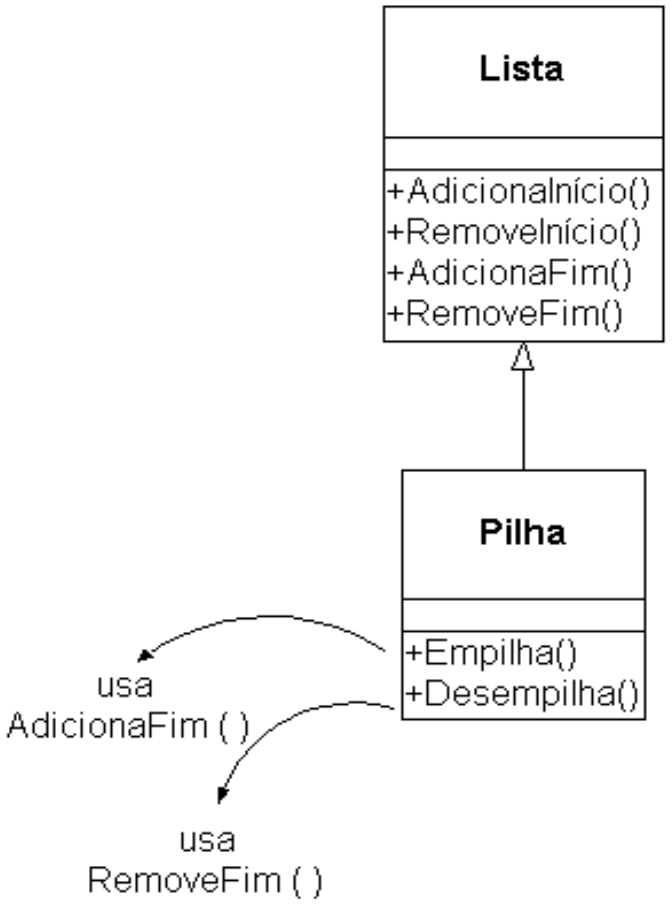


Herança

- Usos de herança
  - A herança é usada para a construção de:
    - Implementação.
    - Comportamento.
  - Herança de implementação
    - Esse tipo de herança é usado como uma técnica para implementar um tipo abstrato de dados que é similar a outros tipos já existentes.

Herança

- Exemplo de herança de implementação
  - Suponha que temos que construir uma classe Pilha, mas já temos implementado uma classe Lista.



Herança

- Poderíamos implementar Pilha como uma subclasse de Lista usando uma estrutura de lista ligada para implementarmos a pilha.
- A operação *empilha()* pode ser implementada adicionando um elemento no fim da lista (através do método *adicionaFim( )* herdado de Lista).
- A operação *desempilha( )* pode ser implementada pela remoção do elemento do fim da lista (através do método *removeFim( )*).

Herança

- Este tipo de uso de herança pode levar a sérios problemas se as operações que são herdadas proporcionam um comportamento não desejado.
- É possível que algumas operações indesejadas possam ser herdadas:
  - *adicionaInício( ).*
  - *RemoveFim( ).*
- Se elas forem usadas, por causa de algum erro na programação, a pilha será corrompida.
- C++, por exemplo, implementa herança de implementação através de um mecanismo chamado derivação de classe.

Herança

- Herança de comportamento
  - Esse tipo de herança é usado para a construção de hierarquias de tipos.
  - Definição de subtipo: um tipo S é um subtipo de T se, e somente se, S proporciona pelo menos o comportamento de T.
  - Em outras palavras, um objeto do tipo T pode ser substituído por um objeto do tipo S.
  - Essa noção é conhecida como conformidade, isto é, tipo S se conforma (ou é subtipo) do tipo T.

Herança

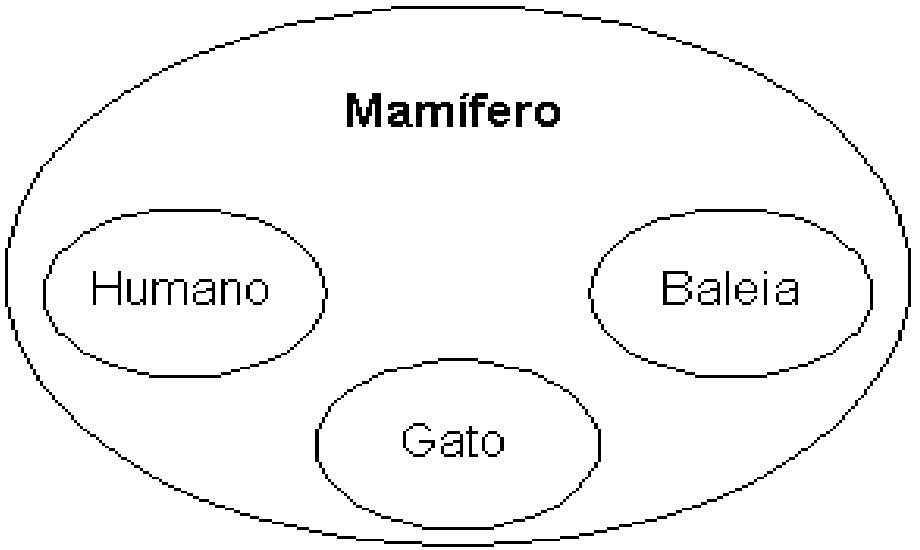
- Exemplo de subtipo na linguagem ADA:

```
type FLAVOUR is
(Chocolate, Mint, Peach, Vanilla, Garlic,
 Onion)
subtype ICE_CREAM is
   FLAVOUR range Chocolate .. Vanilla;
subtype SMALL_INT is INTEGER range
 -10..10;
```

- Uma variável do tipo ICE_CREAM_FLAVOUR herda todas as propriedades do tipo FLAVOUR.
- Da mesma forma que uma variável do tipo SMALL_INT herda todas as propriedades do tipo INTEGER.

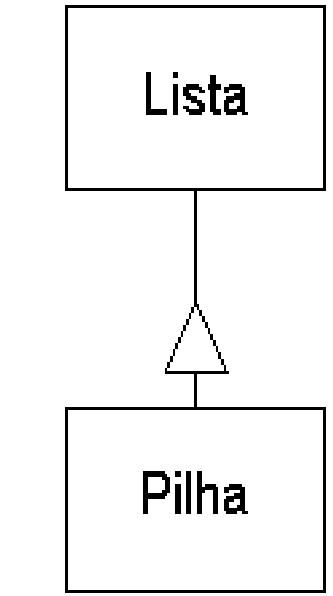
Herança

- Intuitivamente, a noção de tipo/subtipo assemelha-se à noção de conjunto/subconjunto.
- Exemplo:

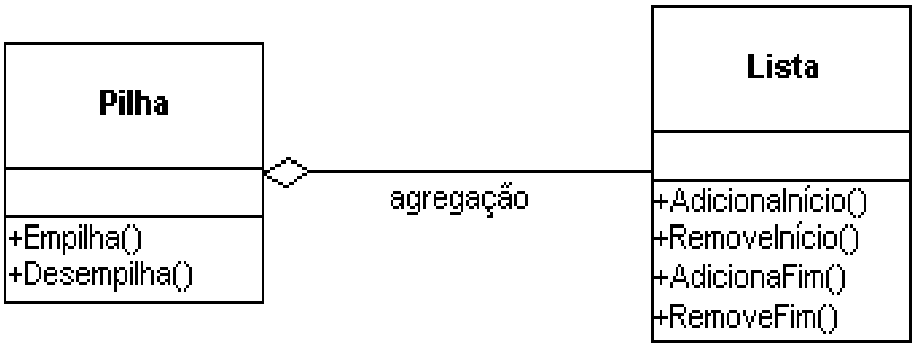


Herança

- Em orientação a objetos, compartilhamento de comportamento é somente justificável quando existe um relacionamento verdadeiro de generalização/especialização entre as duas classes.
- A pilha não é um subtipo de lista.



INCORRETO



RECOMENDADO

Herança

- Herança Múltipla
  - A herança múltipla permite combinar diversas classes para produzir uma nova classe.
  - Quando uma classe herda de mais de um pai, existe a possibilidade de conflitos: operações e atributos com mesmo nome, mas com semânticas diferentes.

Herança

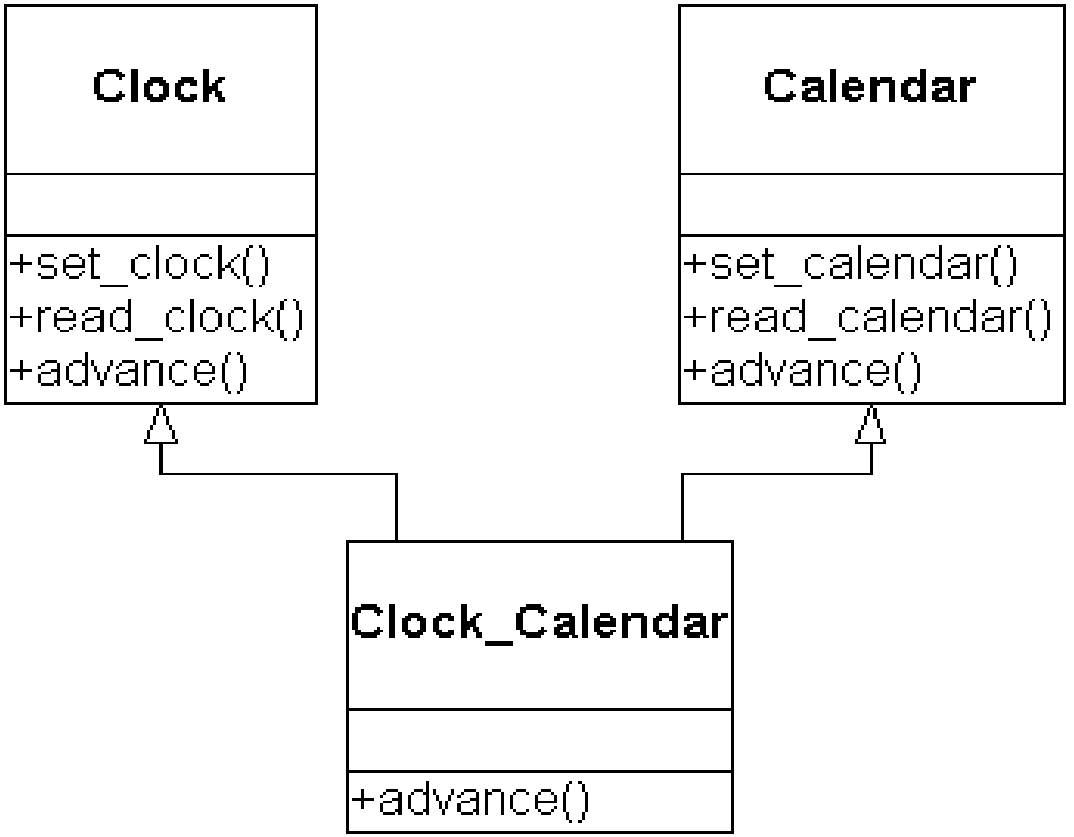
- Estratégias propostas:
  - Linearização. Esta estratégia especifica uma ordem linear e total das classes e determina que a busca do atributo ou método seja efetuada da ordem estabelecida.
  - Renomeação. Ela requer a alteração dos nomes de atributos e operações que sejam conflitantes. Esta abordagem é implementada por Eiffel.
  - Qualificação. Sempre que ocorrer ambigüidade deve-se usar o operador de escopo "::", como por exemplo em C++. Esse operador permite a qualificação (identificação única) do atributo ou método conflitante. Esta abordagem é implementada em C++.

Herança

- A linearização esconde o problema da resolução de conflitos do programador, mas introduz uma ordem obrigatória na herança das classes.
- A renomeação e a qualificação são estratégias que promovem uma maior flexibilidade para o programador decidir a aplicabilidade dos métodos e/ou atributos herdados.
- A vantagem da herança múltipla é poderosa para a especificação de classes e para o aumento da oportunidade de reutilização de comportamento.
- A desvantagem é uma perda da simplicidade conceitual e de implementação.
- Entretanto, ela pode ser muito útil se os cuidados apropriados forem tomados na definição de uma semântica correta e consistente quando os conflitos ocorrerem.

Herança

- Exemplo de herança múltipla



Herança

- Exemplo de herança múltipla em C++ (código)

```
class Clock {
protected:
int hr; int min; int sec; int is_pm;
public:
Clock (int h, int s, int m, int pm);
void set_clock (int h, int m, int s, int pm);
void read_clock (int &h, int &m, int &s, int &pm);
void advance( );
};
```

Herança

- Exemplo de herança múltipla em C++ (código)

```
class Calendar {
protected:
int mo; int day; int yr;
public:
Calendar (int m, int d, int y);
void set_calendar (int m, int d, int y);
void read_calendar (int &m, int &d, int &y);
void advance( );
};
```

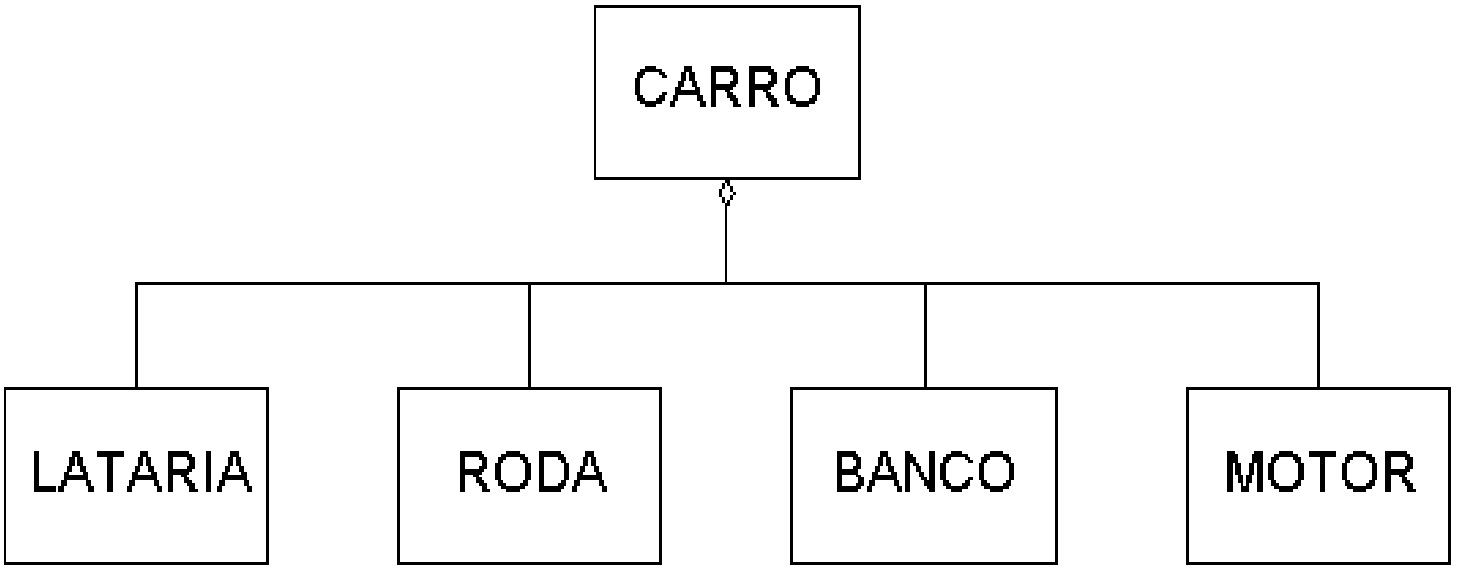
Herança

- Exemplo de herança múltipla em C++ (código)

```
class Clock_Calendar {
public:
Clock_Calendar (int mt, int d, int y, int h,
                int mn, int s, int pm);
void advance( ) {Clock::advance ( );
                 Calendar::advance ( );}
};
```

Herança

- Agregação vs. Generalização
  - O uso de heranças longas ou de herança múltipla podem confundir programadores.
  - Exemplo:



Herança

- A herança é uma forma de reutilização de software em que novas classes são criadas a partir das classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos que as novas classes exigem.
- Em relação ao nível, a hierarquia pode ser realizada de dois modos:
  - Direta, em que uma subclasse herda explicitamente da superclasse, através da palavra-chave **extends.**
  - Indireta, em que uma subclasse herda de dois ou mais subníveis acima na hierarquia da classe (pacotes).

Herança

- O ocultamente de informações é mantido através do controle de acesso, com o uso da palavra **private**.
- A subclasse pode, entretanto, acessar os membros **public** e **protected** de sua superclasse.
- A subclasse também pode usar os membros com acesso de pacote de sua superclasse se a subclasse e a superclasse estiverem no mesmo pacote.

Herança

- Quando um método de subclasse sobrescreve um método de uma superclasse, o método de superclasse pode ser acessado a partir da subclasse que precede o nome do método de superclasse com a palavra-chave **super**, seguida pelo operador ponto (**.**).
- Membros de classe definidos como **final** possuem valor fixo e podem ser usados fora da classe. Assim, sendo recomenda-se que sejam declarados como **public**.

Herança

- O objeto de uma subclasse pode ser tratado como um objeto de sua superclasse, mas o contrário não é necessariamente verdadeiro.
- Assim como o projetista de sistemas não-orientados a objetos deve evitar a proliferação desnecessária de funções, o projetista de sistemas orientados a objetos deve evitar a proliferação desnecessária de classes.
- A proliferação pode classes cria problemas de gerenciamento e pode impedir a reutilização de softwares.

Polimorfismo

- O que é polimorfismo?
  - Significa que variáveis podem referenciar mais do que um tipo.
  - Não é um conceito novo e várias linguagens de programação aplicam.
  - Funções são polimórficas quando seus operandos (parâmetros atuais) podem ter mais do que um tipo.
  - Tipos são ditos polimórficos se suas operações podem ser aplicadas a operandos de mais de um tipo.

Polimorfismo

- Exemplo de função polimórfica:

`comprimento::[A] -> NUM, para todos os tipo A.`

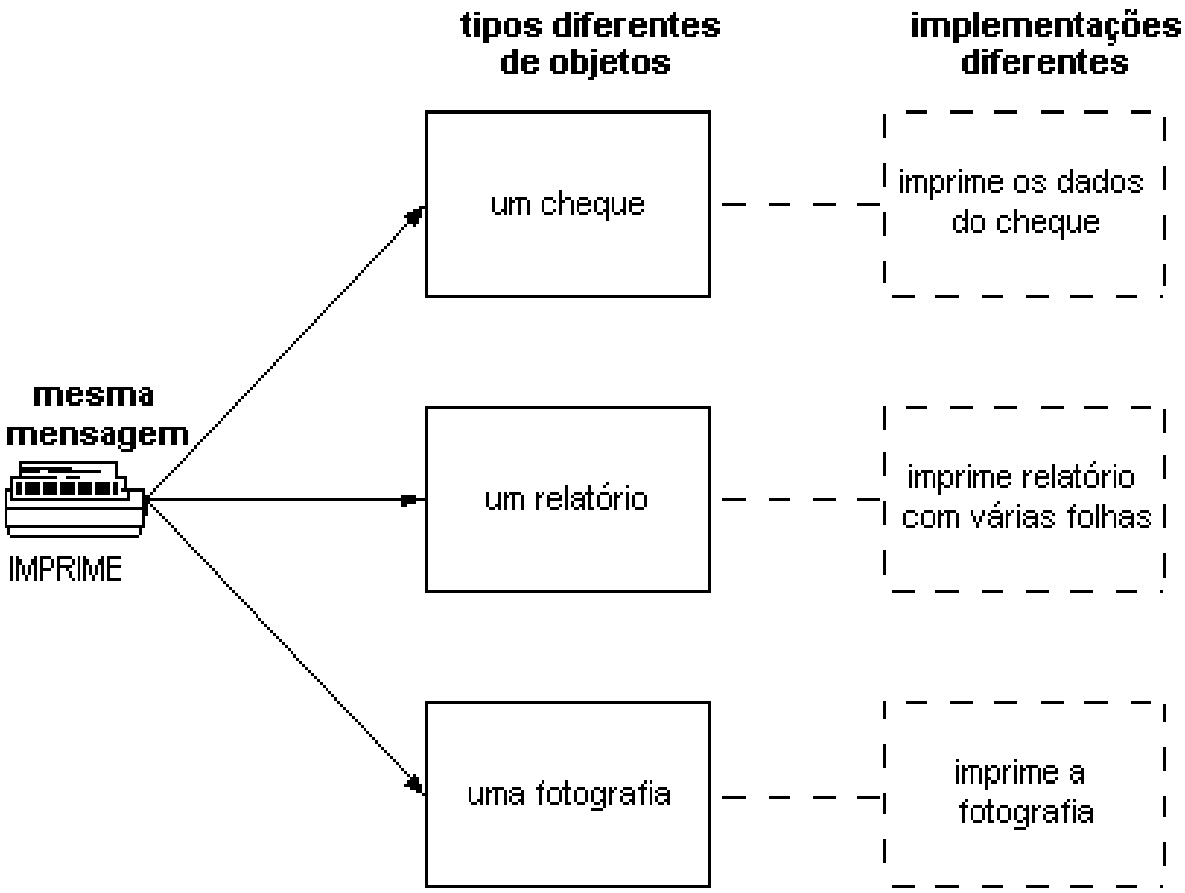
- Portanto:
  - Comprimento é uma função cujo parâmetro de entrada é uma lista (simbolizada pelos colchetes).
  - O tipo do conteúdo da lista não importa (A).
  - A função devolve um inteiro como saída.
- Uma linguagem de tipos monomórficos forçaria o programador a definir diferentes funções para retornar o comprimento de uma lista de inteiros, de uma lista de reais e assim por diante. Exemplo: Pascal e Algo68.

Polimorfismo

- Em particular, no contexto do modelo de objetos, polimorfismo significa que diferentes tipos de objetos podem responder a uma mesma mensagem de maneiras diferentes.

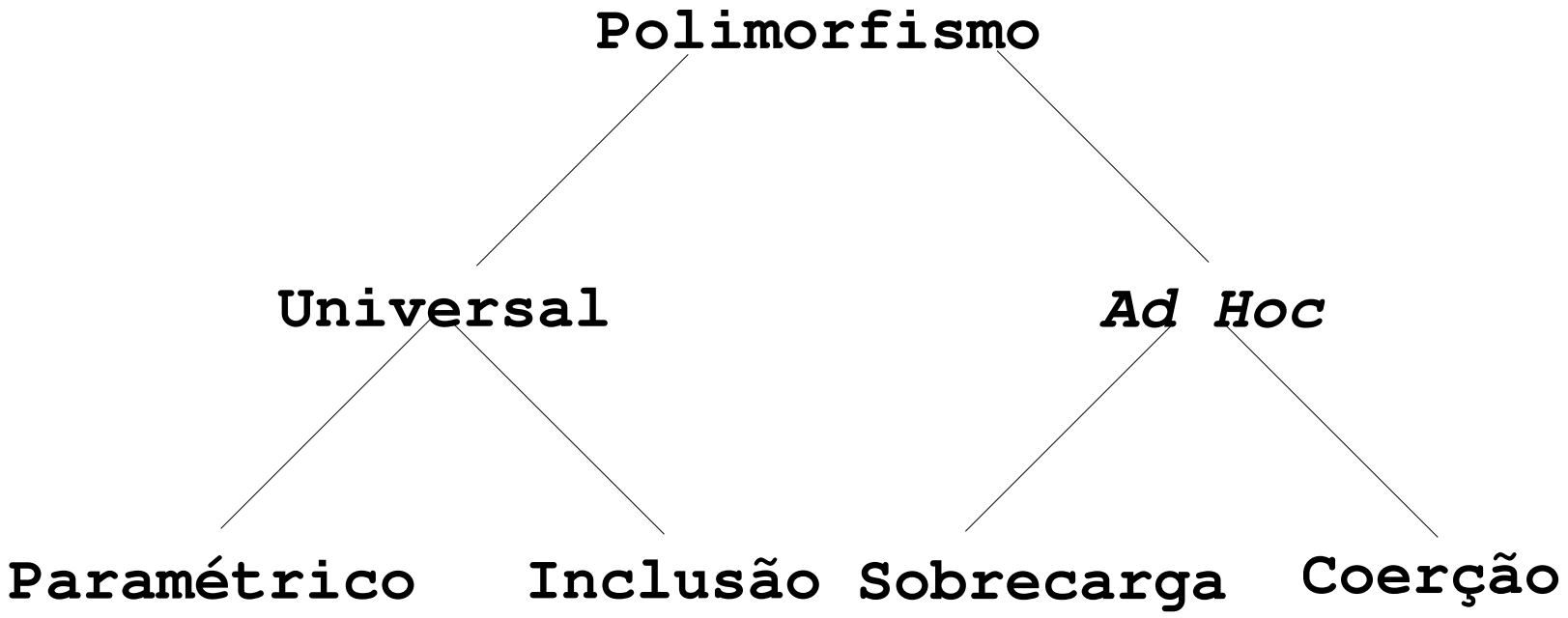
Polimorfismo

- Exemplo de método polimórfio



Polimorfismo

- Formas de polimorfismo
  - Uma taxonomia amplamente utilizada:



Polimorfismo

- Polimorfismo *ad hoc*
  - Não existe um modo único e sistemático de determinar o tipo de resultado de uma função em termos dos tipos dos seus argumentos de entrada.
  - É uma forma limitada de polimorfismo.
  - Possui duas formas: coerção e sobrecarga.
- Polimorfismo universal
  - Trabalha potencialmente num conjunto infinito de tipos de modo disciplinado.
  - Possui duas formas: paramétrico e inclusão.

Polimorfismo

- Coerção
  - Meio para contornar a rigidez dos tipos monomórficos.
  - Existe um mapeamento interno entre tipos.
  - Exemplo:
    - Se o operador soma é definido como tendo 2 parâmetros reais e um inteiro e um real são passados como parâmetros, o inteiro é "coargido" para um real.
- Sobrecarga
  - Permite que um nome de função seja utilizado mais do que uma vez com diferentes tipos de parâmetros.
  - Exemplo:
    - Uma função soma pode ser sobrecarregada para operar com dois parâmetros inteiros e dois reais.

Polimorfismo

- Paramétrico
  - Uma única função é codificada e ela trabalhará uniformemente num intervalo de tipos.
  - Funções paramétricas também são chamadas de funções genéricas.
  - Exemplo:
    - A função comprimento apresentada anteriormente.
- Inclusão
  - É encontrada somente em linguagens orientadas a objetos e está relacionada com a noção de subtipo.
  - Exemplo:
    - Hierarquia de classes.

Polimorfismo

- Em C++, por exemplo, temos o polimorfismo parámetrico (através do uso da palavra reservada template), o polimorfismo de sobrecarga e o polimorfismo de inclusão.
- Em C++, o uso de funções membros virtuais numa hierarquia de classes permite que a seleção em tempo de execução da versão mais adequada do método.

Acoplamento Dinâmico

- O que é acoplamento?
  - Acoplamento é uma associação possivelmente entre um atributo e uma entidade.
  - Exemplo:
    - Variáveis são entidades de programas e seus atributos são seu nome, seu tipo e sua área de armazenamento.
  - O tempo em que o acoplamento entre a entidade e seus atributos ocorre é chamado de tempo de acoplamento.
  - Um acoplamento pode ser estático ou dinâmico.

Acoplamento Dinâmico

- Um acoplamento é estático ou adiantado se ocorre antes do tempo de execução e permanece inalterado durante a execução do programa.
- Um acoplamento é dinâmico ou atrasado se ocorre durante o tempo de execução e muda no curso da execução do programa.
- A maior vantagem do acoplamento dinâmico é que as associações podem ser alteradas em tempo de execução.

Acoplamento Dinâmico

- Em programação orientada a objetos, temos o uso de verificação estática de tipos associada com acoplamento dinâmico.
- O acoplamento dinâmico é usado para a associação do nome de uma operação à sua implementação.
- Essa associação é decidida apenas em tempo de execução, pois a implementação a ser executada depende do tipo do objeto que recebe a mensagem.

Acoplamento Dinâmico

- Classes Abstratas vs. Classes Concretas
  - Uma classe abstrata é uma classe que não tem instâncias diretas, mas cujas classes descendentes têm instâncias diretas.
  - Uma classe concreta é uma classe que pode ser instanciada.
  - Existem duas idéias principais envolvidas com a noção de classe abstrata:
    - Presenção de operações abstratas.
    - Habilidade de criar instâncias.

Acoplamento Dinâmico

- Se uma classe contém operações abstratas então a classe não pode ser instanciada porque se um objeto for criado, ele não será capaz de responder a todas as mensagens.
- Portanto, a presença de uma operação abstrata implica na inabilidade de instanciar objetos.
- Entretanto o contrário não é verdadeiro.

Acoplamento Dinâmico

- O que é delegação?
  - Mecanismo parecido com o de herança, mas que opera diretamente entre objetos.
  - Exemplos de linguagens de programação:
    - Self.
    - Actor.
  - Neste modelo, os objeto são vistos como protótipos ou exemplares que delegam seu comportamento para objetos relacionados chamados delegados.
  - Um relacionamento delaga-para pode ser estabelecido dinamicamente.

Acoplamento Dinâmico

- A conseqüência é que as alterações efetuadas nos métodos e estrutura da classe podem ser automaticamente passadas para todas as suas instâncias.
- Este mecanismo implícito de alteração permite que sistemas possam ser atualizados com base em grupos.
- Neste modelo, as distinções entre classes e objetos são removidas. Na verdade, a noção de classe é eliminada.
- O projetista inicialmente considera um protótipo em particular e depois descobre similaridades e/ou diferenças com outros objetos.
- A idéia é começar com casos particulares e depois generalizá-los ou especializá-los.

Acoplamento Dinâmico

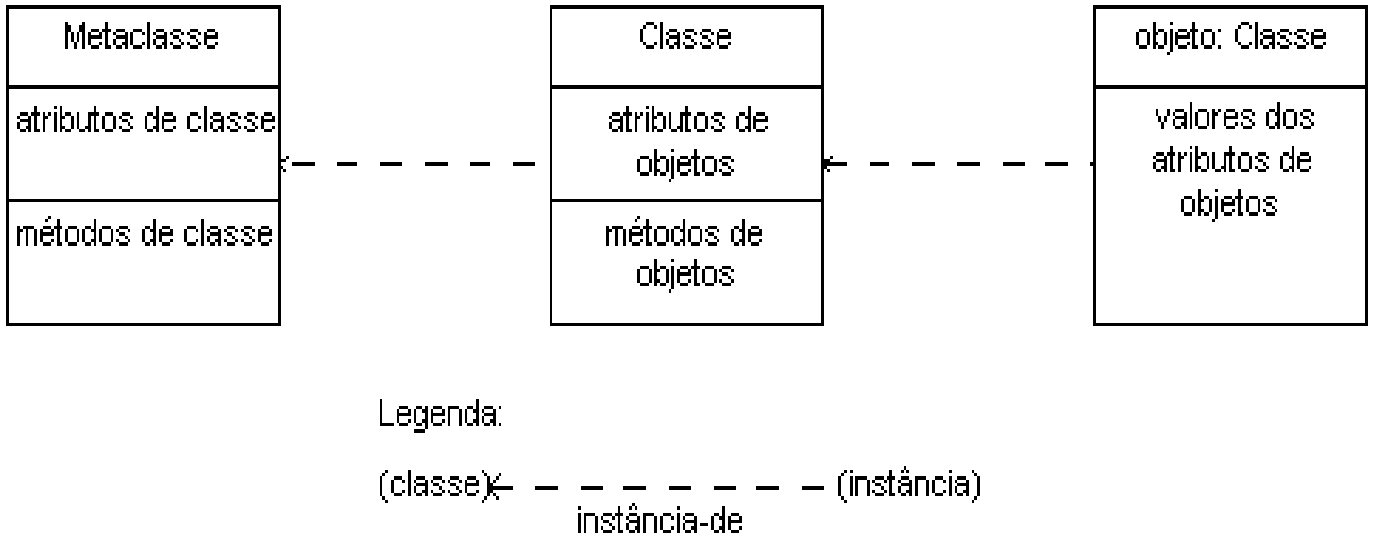
- A principal vantagem de delegação sobre herança é que delegação facilita a mudança de implementação de operações em tempo de execução.

Acoplamento Dinâmico

- Metaclasses
  - Metainformação é um termo genérico aplicado a qualquer dado que descreve outro dado.
  - As classes não são objetos porque elas próprias não são instâncias de classes.
  - Entretanto, em algumas linguagens orientadas a objetos, tal como Smalltalk, existem dois tipos de elementos geradores no modelo de objetos:
    - Metaclasses que geram classes.
    - Classes que geram instâncias terminais.
  - Portanto, nesse caso, todas as entidades do modelo de objetos são objetos, inclusive as classes.

Acoplamento Dinâmico

- Conceito de Metaclasse:



- Mais especificamente, metaclasse é uma classe que descreve outra classe, isto é, ela é uma classe cujas instâncias são classes. Ela guarda metainformação de uma classe.

Acoplamento Dinâmico

- Existem pelo menos dois benefícios da representação de classes como objetos:
  - As informações globais relativas a todos os objetos de uma classe podem ser armazenadas nos atributos de classe.
  - A segunda vantagem é o seu uso para criação/iniciação de novas instâncias da classe.
- Linguagens como C++ e Java não dão apoio direto para a abordagem classes como instâncias de metaclasses.

Acoplamento Dinâmico

- De modo geral, pode-se dizer que o paradigma orientado a objetos dá suporte a pelo menos três níveis de abstrações:
  - Abstração de dados para comunicação de objetos.
  - Super-abstração (herança) para o compartilhamento de comportamento e gerência de objetos.
  - Meta-abstração (metaclasse) como base para a auto-representação do sistema.

Pacotes

- Introdução
  - À medida que os aplicativos se tornam mais complexos, os pacotes ajudam os programadores a administrar a complexidade dos componentes.
  - Os pacotes também facilitam a reutilização de software, permitindo que os programas importem classes de outros pacotes.
  - Outro benefício dos pacotes é que eles fornecem uma convenção para nomes de classe únicos. Isto evita os conflitos de nomes de classes desenvolvidas por programadores diferentes.

Pacotes

- Passos para criação de uma classe reutilizável:

1. Definir uma classe **public**. Se a classe não for **public**, ela pode ser utilizada somente por outras classes no mesmo pacote.
2. Escolher um nome de pacote e adicionar uma instrução **package** ao arquivo do código-fonte para a definição da classe reutilizável.
3. Compilar a classe de modo que ela seja colocada na estrutura de diretórios de pacotes apropriada.
4. Importar a classe reutilizável para dentro de um programa e utilizar a classe.

Pacotes

```
1. package nome_do_pacote;
2. import nome_do_pacote;
3. public classe nome_classe {
4.   declaração_das_variáveis;
5.   métodos{
6.     }
7. }
```

Pacotes

- Comentários (passo 2)
  - A Sun Microsystems especifica uma convenção para nomes de pacotes:
    - Cada nome de pacote deve iniciar com seu nome de domínio na Internet na ordem inversa.
    - Depois disso, o programador pode escolher qualquer outro nome para o pacote.
    - Exemplo: org.comunidadesol.programacao.

Pacotes

- Comentários (passo 3)
  - Após a compilação, o arquivo **.class** resultante é colocado no diretório especificado pela instrução **package**.
  - Se os diretórios não existirem, antes da classe ser compilada, o compilador os cria.
  - A opção **-d** deve ser passada para o compilador para indicar onde criar (localizar) os diretórios na instrução **package**.
  - Exemplo: javac -d . *nome_do_arquivo*
  - Os nomes de diretório em **package** tornam-se parte do nome de classe quando a classe é compilada. Isto evita o conflito de nomes.

Pacotes

- Comentários (passo 4)
  - O compilador segue uma ordem específica para localizar as classes de que ele precisa.
  - A seqüência é:
    - Ele começa procurando as classes-padrão de Java que estão incluídas no J2SDK.
    - A seguir, ele procura as classes de extensão.
    - Caso não encontre, o compilador pesquisa no caminho para as classes. Por default, o caminho para as classes consiste apenas no diretório corrente. Entretanto, este diretório pode ser modificado.

Pacotes

- Pode-se mudar da seguinte maneira:
  - Fornecendo a opção ***-classpath*** para o compilador javac.
  - Configurando a variável de ambiente CLASSPATH.

Construtores

- Quando um objeto é criado, seus membros podem ser inicializados por um método construtor.
- O construtor é um método com o mesmo nome que a classe.
- As variáveis podem ser inicializadas de três formas:
  - Com seus valores default.
    - 0 para valores numéricos.
    - False para valores lógicos.
    - Null para referências.
  - Em um construtor da classe.
  - Ou seus valores podem ser configurados mais tarde, depois que o objeto for criado.

Construtores

- Os construtores nunca são herdados – eles são específicos para a classe em que são definidos.
- Quando um objeto de uma subclasse é instanciado, o construtor da superclasse deve ser chamado para fazer qualquer inicialização necessária das variáveis de instância da superclasse do objeto de subclasse.
- Uma chamada explícita ao construtor de superclasse (através da referência **super**) pode ser fornecida como a primeira instrução no construtor de subclasse.
- Caso contrário, o construtor de subclasse chamará o construtor default da superclasse ou o construtor sem argumentos implicitamente.

Construtores

- Métodos que fazem a interface externa de uma classe devem ser declarados como **public**, sendo herdados pelas sub-classes.
- Variáveis devem ser **private**.
- Se tiver o objetivo de que a classe seja usada como super-classe por outras pessoas, deve-se manter as variáveis como **private** e prover métodos de acesso e manipulação como **public**. Controla-se assim, a manipulação da classe base por parte da classe derivada.
- Usamos o atributo protected quando definimos classes dentro de um package e desejamos dar ao usuário do pacote acesso apenas às sub-classes.

Construtores – Exemplo 01

```
public class Animal {

private String tipo;

public Animal(String tipo1){

   tipo = new String(tipo1);

   }

public void exibir() {

   System.out.println("Eu sou um" + tipo);

   }

}
```

Construtores – Exemplo 01

```
public class Cachorro extends Animal {
private String nome, raca;
public Cachorro(String nome1){
   super(“cachorro”); //construtor
   nome = nome1;
   raca = “SRD”; //raça default }
public Cachorro(String nome1, String r){
   super(“cachorro”); //construtor
   nome = nome1;
   raca = r; }
}
```

Construtores – Exemplo 01

```
public class Teste {

public static void main(String[] args) {

Cachorro cao = new Cachorro (“Lassie”, “Colie”);

Cachorro outro = new Cachorro (“Rintintim”);

cao.show();

outro.show();

   }

}
```

Construtores – Exemplo 01

- Sobrecarregando métodos da classe base:
  - Inclua na classe Cachorro o método:

```
public void exibir() {
super.show(); //chama o método base
System.out.println(nome + “é um” + raca);
}
```

Construtores – Exemplo 02

```
public class Point extends Object {
   protected int x,y;
   public Point () {
      x=0; y=0; }
   public Point (int xCoord, int yCoord) {
      x = xCoord; y = yCoord; }
   public String toString() {
      return "[" + x + "," + y + "]"; }
}
```

Construtores – Exemplo 02

```
public class Circulo extends Point {
   protected double raio;
   public Circulo() {
      raio = 0;}
   public Circulo (double Raio, int xCoord,
     int yCoord){
      super (xCoord, yCoord);
      raio = Raio; }
   public String toString() {
      return "Centro = " + super.toString() +
       "; Raio = " + raio; }
}
```

Construtores – Exemplo 02

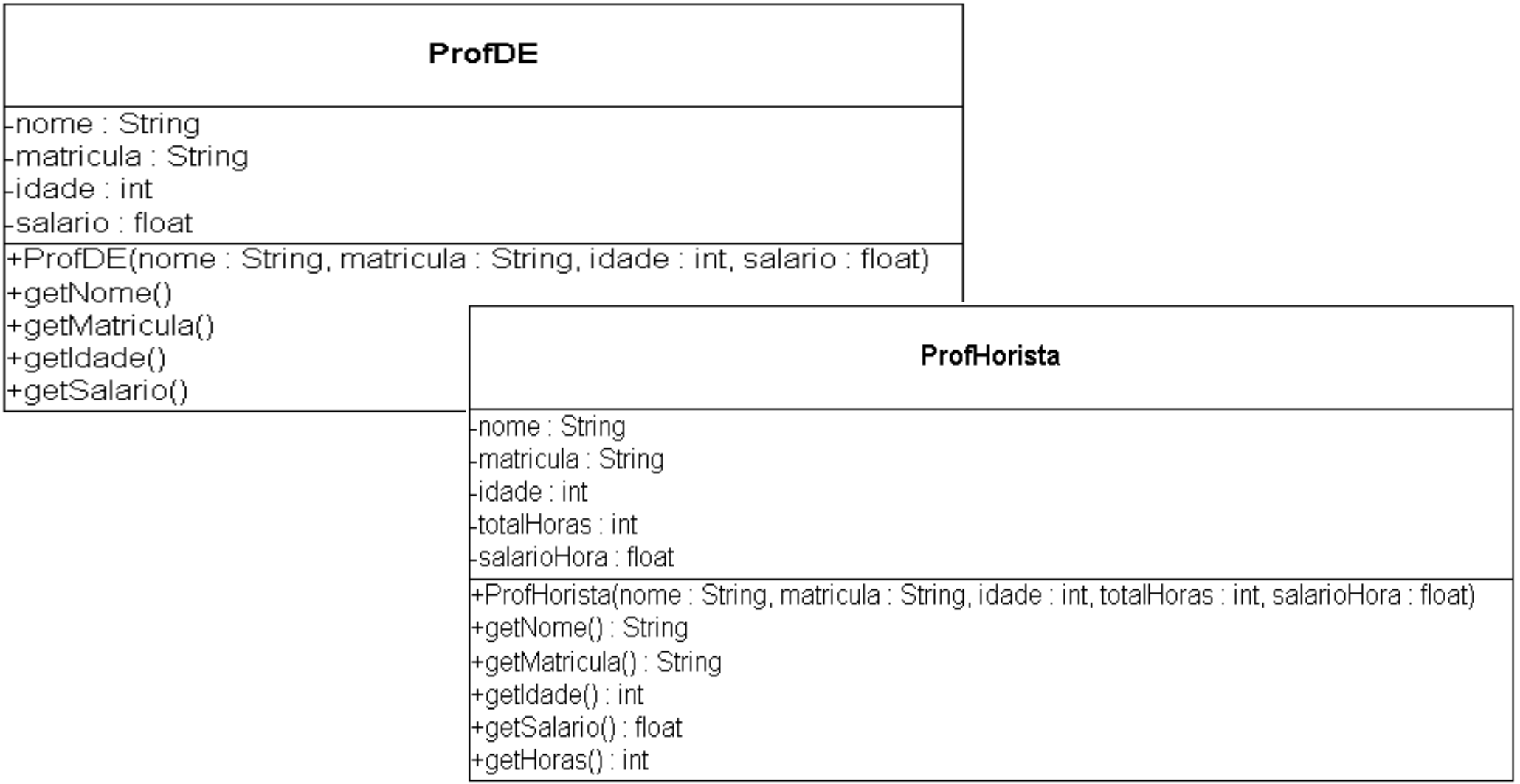
```
public class Teste {
   public static void main (String args[] ) {
      Circulo circulo1, circulo2;

      circulo1 = new Circulo (4.5, 72, 29);
      circulo2 = new Circulo (10, 5, 5);

      circulo1 = null;
      circulo2 = null;
      System.gc();
      }
}
```

Estudo de Caso de Herança

- Os professores de uma universidade se dividem em dois grupos:
  - Professores de dedicação exclusiva (DE).
  - Professores horistas.
- Professores de dedicação exclusiva possuem um salário fixo para 40 horas de atividades semanais.
- Professores horistas recebem um valor estipulado por hora.
- O cadastro de professores desta universidade armazena: nome, idade, matrícula e informação de salário.
- O problema pode ser solucionado através da seguinte modelagem.

Estudo de Caso de Herança

- Classes ProfDE e ProfHorista



Estudo de Caso de Herança

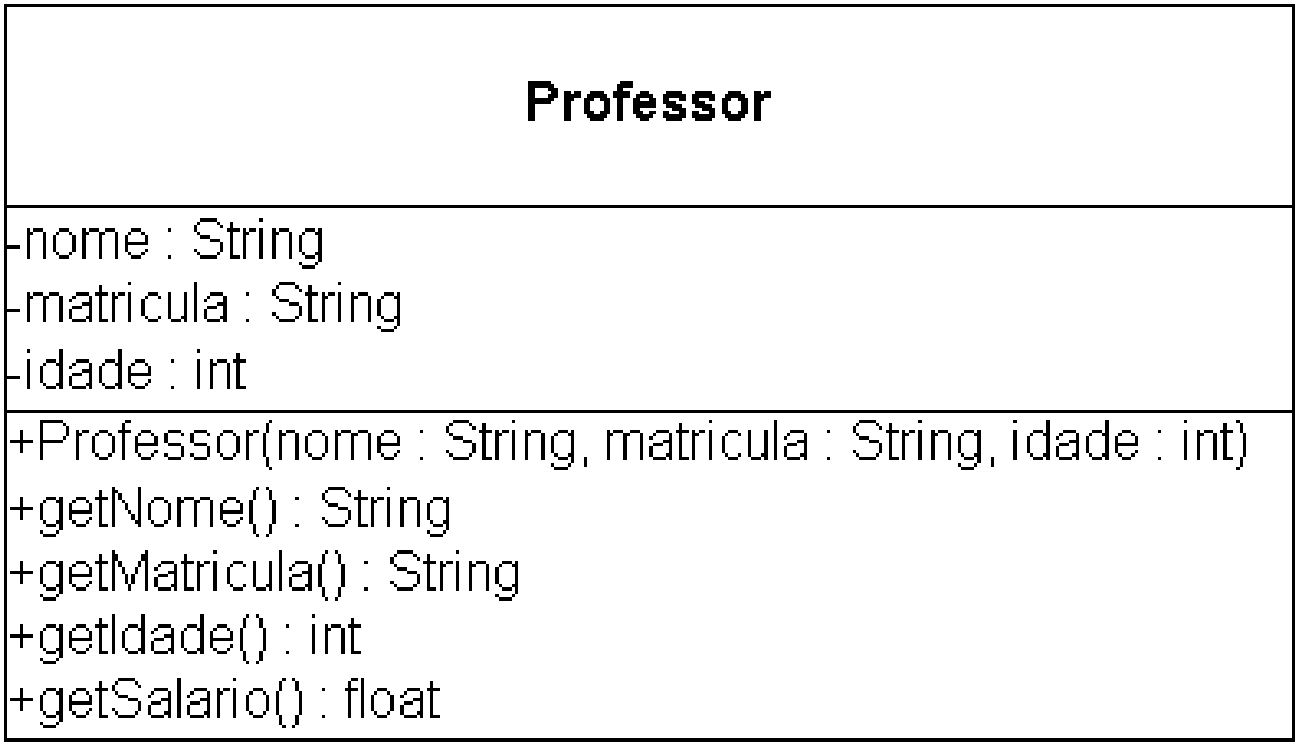
- Observações:
  - As classes têm alguns atributos e métodos iguais.
  - O que acontece se precisarmos alterar a representação de algum atributo, como por exemplo, o número de matrícula para inteiro ao invés de uma string?
    - Será necessário alterar os construtores e métodos getMatrícula( ) nas duas classes, o que é ruim para programação.
  - Motivo: código redundante.
  - A herança é indicada neste caso.

Estudo de Caso de Herança

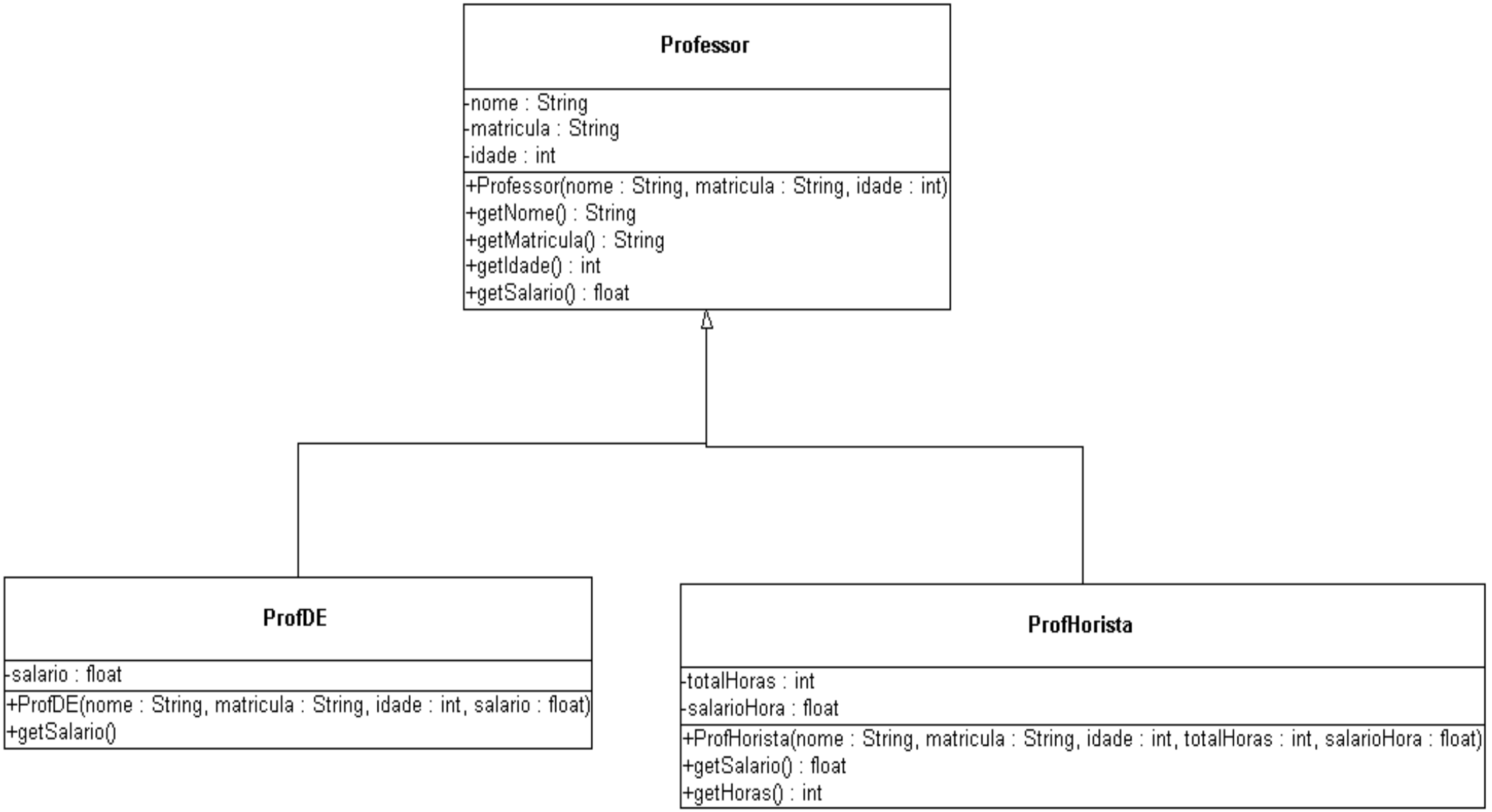
- A solução é criar uma classe Professor que contém os atributos e métodos comuns aos dois tipos de professores.
- A partir dela, criam-se duas novas classes que representam os professores horistas e DE.
- Para isso, essas classes deveriam herdar os atributos e métodos declarados pela classe Professor.

Estudo de Caso de Herança

- Classe Professor



Estudo de Caso de Herança



Estudo de Caso de Herança

```
class Professor {
String nome, matricula;
int idade;
public Professor(String n, String m, int i){
   nome = n; matricula = m; idade = i;
   }
public String getNome() {return nome; }
public String getMatricula() {return
  matricula;}
public int getIdade() {return idade;}
}
```

Estudo de Caso de Herança

```
class ProfDE extends Professor {

float salario;

public ProfDE(String n, String m, int i, float
  vs){
   super(n, m, i);
   if (vs > 0) salario = vs;
   }
public float getSalario(){
   return salario - (salario * 0,16);
   }
}
```

Estudo de Caso de Herança

```
class ProfHorista extends Professor {

float salarioHora; int totalHoras;

public ProfHorista(String n, String m, int i,
  float vs){

   super(n, m, i);

   if (h > 0) totalHoras = h;

   if (vs > 0) salarioHora = vs; }

public float getSalario(){

   float salarioBase = salarioHora *
     totalHoras;

   return salarioBase - (salarioBase * 0,16);}

}
```

Herança Múltipla

- A herança múltipla em Java é realizada através das interfaces.
- As interfaces são usadas quando o programador quer definir uma certa funcionalidade para ser usada em várias classes, mas não tem certeza de como exatamente essa certa funcionalidade será definida por essas classes.
- As interfaces definem apenas os métodos abstratos e os campos finais, mas não podem especificar qualquer implementação para esses métodos.

Herança Múltipla

- A sintaxe para a criação de uma interface é semelhante àquela para a criação de uma classe.
- As declarações de interface têm a seguinte sintaxe:

  public interface nomeinterface **extends** outrainterface
- Por padrão, as interfaces podem ser implementadas por todas as classes no mesmo pacote.
- Ao tornar a interface pública, você permite que as classes e objetos fora do determinado pacote também a implementem.

Herança Múltipla

- Apesar de não poder especificar nenhuma implementação, o corpo de uma interfce especifica suas propriedades.
- Embora a maioria dos benefícios das interfaces provenha de sua capacidade de declarar métodos, as interfaces também podem ter variáveis finais.

Polimorfismo

- Polimorfismo é a capacidade de assumir formas diferentes.
- O polimorfismo permite-nos escrever programas de uma forma geral para tratar uma ampla variedade de classes relacionadas existentes e ainda a serem especificadas.
- Usa-se uma variável de um tipo único (normalmente tipo da superclasse) para referenciar objetos variados do tipo das subclasses.
- Envolve o uso automático do objeto armazenado na super-classe para selecionar um método de uma das sub-classes. O tipo do objeto armazenado não é conhecido até a execução do programa. A escolha do método a ser executado é feita dinamicamente.

Estudo de Caso de Polimorfismo

```
public class Animal {
   private String tipo;
   public Animal(String tipo1){
      tipo = new String(tipo1);
      }
   public void exibir(){
      System.out.println("Eu sou um" + tipo);
      }
   //Método a ser implementado nas sub-classes.
   public void sound(){ }
   }
```

Estudo de Caso de Polimorfismo

```
public class Cachorro extends Animal {
   private String nome, raca;
   public Cachorro(String nome1){
      super(“cachorro”);
      nome = nome1; raca = “SRD”; }
   public Cachorro(String nome1, String raca1){
      super(“cachorro”);
      nome = nome1; raca = raca1; }
   public void sound() {
      System.out.println(“Au, Au”);
      }
   }
```

Estudo de Caso de Polimorfismo

```
public class TestePolimorfismo {

   public static void main(String[] args){

      Animal[] bichos = {new Cachorro("Rintintin",
       "Pastor Alemao"), new Gato("Garfield"), new
       Pato("Donald")};

      Animal mascote;

      for (int i=0; i<5; i++){

         int indice = (int) (bichos.length *
          Math.random() - 0.001);

         mascote = bichos[indice];

         System.out.println("\n Escolha: ");

         mascote.exibir();

         mascote.sound(); }

      }
```

Polimorfismo

- Retorna **true** caso o objeto **a** seja uma instância da classe Gato.
  - Animal a.
  - if (a instanceof Gato)
  - // vacinar

Polimorfismo

- Consideremos o método **sound**() desenvolvido na classe Animal e que não possui nenhuma instrução, não fazendo sentido.
- Este tipo de situação (criar uma superclasse a partir da qual subclasses serão derivadas) acontece muito freqüentemente em programação orientadas a objetos, quando o objetivo é apenas tirar vantagem do polimorfismo.
- Uma outra alternativa em Java é o uso de classes abstratas. Uma classe abstrata é a classe na qual um ou mais métodos são declarados, mas não implementados.

Classes Abstratas

```
public abstract class Animal {
   private String tipo;
   public Animal(String tipo){
      tipo = new String(tipo); }
   public abstract void sound();}
   }
```

- Não é possível instanciar um objeto de uma classe abstrata, mas pode-se declarar uma variável deste tipo.

Reutilização de Software

- Os programadores de Java concentram-se na elaboração de novas classes e na reutilização de classes existentes.
- A reutilização não apenas reduz o tempo de desenvolvimento, mas também melhora a capacidade de depurar e manter aplicativos dos programadores.
- Deve-se observar, porém, para tirar proveito dos inúmeros recursos de Java, é essencial que os programadores dediquem tempo para familiarizar-se com a grande variedade de pacotes e classes da Java API.

Bibliografia

- C. M. F. Rubira. *Introdução à Análise Orientada a Objetos.* Editora da Unicamp, 2002.
- H. M. Deitel e P. J. Deitel. *Java – Como Programar*. 4a. ed. Bookman, 2003.