

Paradigmas de Linguagens de Programação



Suporte para Programação Orientada a Objeto

Cristiano Lehrer, M.Sc.

Categoria das Linguagens que Suportam POO

- Suporte a POO acrescentado a uma linguagem já existente:
 - **C++** (também suporta programação procedural e orientada a dados).
 - **Ada 95** (também suporta programação procedural e orientada a dados).
 - **CLOS** (também suporta programação funcional).
 - **Scheme** (também suporta programação funcional).
- Suporte a POO, mas com a mesma aparência e usam a mesma estrutura básica das linguagens imperativas mais antigas:
 - **Java** (baseada no **C++**).
- Linguagens de POO puras:
 - **Smalltalk**.
 - **Eiffel**.

Evolução do Paradigma

- Procedural – 1950s - 1970s:
 - Abstração procedural.
- Orientada a dados – início dos anos 80:
 - Orientada a dados.
- POO – final dos anos 80:
 - Herança e *binding* dinâmico.

Origem da Herança

- Observações de meados dos anos 80:
 - Aumento da produtividade pode vir da reutilização.
 - TADs são difíceis de reutilizar:
 - Quase nunca corretamente.
 - Todos os TADs são independentes e estão num mesmo nível.
- Herança resolve ambos:
 - Reutilização de TADs após modificações menores e define classes em uma hierarquia.

Definições sobre POO (1/3)

- TADs são chamados classes.
- Instâncias de classes são chamadas objetos.
- Uma classe herdeira é chamada classe derivada ou uma subclasse.
- A classe da qual outra classe é herdeira é chamada classe-pai ou super-classe.
- Subprogramas que definem operações sobre objetos são chamados métodos.
- A coleção completa de métodos de um objeto é chamada de seu protocolo de mensagens ou interface.

Definições sobre POO (2/3)

- Mensagens possuem duas partes – o nome do método e o objeto de destino.
- No caso mais simples, uma classe herda todas as entidades de seu pai.
- Herança pode ser complicada por controles de acesso para entidades encapsuladas:
 - Uma classe pode ocultar entidades de suas subclasses.
 - Uma classe pode ocultar entidades de seus clientes.
- Além dos métodos padrão de herança, uma classe pode modificar um método herdado:
 - O novo método substitui o herdado.
 - O método na classe pai é sobrepujado.

Definições sobre POO (3/3)

- Existem dois tipos de variáveis em uma classe:
 - Variáveis da classe.
 - Variáveis da instância.
- Existem dois tipos de métodos em uma classe:
 - Métodos da classe – mensagens para a classe.
 - Métodos da instância – mensagens para objetos.
- Herança simples versus herança múltipla.

Polimorfismo em POO

- Uma variável polimórfica pode ser definida em uma classe que é apta a referenciar (ou apontar) a objetos da classe e objetos de qualquer de seus descendentes.
- Quando uma hierarquia de classes inclui classes que sobrepõe métodos que são chamados através de uma variável polimórfica, o *binding* para o método corrente deve ser dinâmico.
- Polimorfismo simplifica a adição de novos métodos.
- Um método virtual é aquele que não inclui uma definição (apenas define um protocolo).
- Uma classe virtual é aquela que inclui ao menos um método virtual.
- Uma classe virtual não pode ser instanciada.

Questões de Projeto para POO (1/5)

- A exclusividade de objetos:
 - Tudo são objetos:
 - Vantagens: elegância e pureza.
 - Desvantagens: operações lentas em objetos simples.
 - Adiciona objetos a um sistema de tipagem completa:
 - Vantagens: operações rápidas em objetos simples.
 - Desvantagens: resulta em um sistema de tipagem confuso.
 - Inclui um sistema de tipagem de estilo Imperativo para primitivas mas considera todo o resto como objeto:
 - Vantagem: operações rápidas em objetos simples e um sistema de tipagem relativamente pequeno.
 - Desvantagem: (ainda) alguma confusão devido aos dois sistemas de tipagem.

Questões de Projeto para POO (2/5)

- Subclasses são subtipos:
 - Um relacionamento “é-um” é mantido entre uma classe pai e um objeto da subclasse.
- Herança de Implementação e de Interface:
 - Se apenas a Interface de uma classe-pai está visível para a subclasse, trata-se de herança de interface:
 - Desvantagem: pode resultar em ineficiências.
 - Se tanto a Interface quanto a Implementação da classe-pai está visível para a subclasse, trata-se de herança de Implementação:
 - Desvantagem: modificações na classe-pai requerem recompilação de subclasses algumas vezes até a modificação das subclasses.

Questões de Projeto para POO (3/5)

- Verificação de tipo e polimorfismo:
 - Polimorfismo pode requerer verificação dinâmica de tipos de parâmetros e do valor retornado:
 - Verificação dinâmica de tipos é onerosa e retarda detecção de erros.
 - Se métodos de sobreposição são restritos a ter os mesmos tipos de parâmetro e valor de retorno, a verificação pode ser estática.

Questões de Projeto para POO (4/5)

- Herança simples e múltipla:
 - Desvantagem de herança múltipla:
 - Complexidade da linguagem e Implementação.
 - Ineficiência potencial – *binding* dinâmico é mais oneroso com herança múltipla (mas não muito).
 - Vantagem:
 - Algumas vezes é extremamente conveniente e valiosa.

Questões de Projeto para POO (5/5)

- Alocação e desalocação de objetos:
 - A partir de onde os objetos são alocados:
 - Se todos estiverem no *heap*, referências a eles são uniformes.
 - A desalocação é explícita ou implícita.
- *Binding* dinâmico:
 - Todas as mensagens para métodos deveriam ser dinâmicas.

Visão Geral do Smalltalk (1/3)

- **Smalltalk** é uma linguagem de POO pura:
 - Tudo são objetos.
 - Toda computação é através de objetos enviando mensagens a objetos.
 - Ela não adota a aparência das linguagens Imperativas.
- O ambiente **Smalltalk**:
 - O primeiro sistema com Interface gráfica (GUI) completo.
 - Um sistema completo para desenvolvimento de software.
 - Todo o código fonte do sistema é disponível para o usuário, que pode modificá-lo, caso deseje.

Visão Geral do Smalltalk (2/3)

```
cont := 1.  
soma := 0.  
[cont <= 20]  
  whileTrue: [  
    soma := soma + cont.  
    cont := cont + 1.  
  ].  
Transcript show: (cont) printString; cr.
```

Visão Geral do Smalltalk (3/3)

```
|fatorial|
  fatorial :=
    [:n |
      n > 1
        ifTrue: [n * (fatorial value:n-1)]
        ifFalse: [n]].
```

Transcript show: (fatorial value:5) printString; cr.

C++ (1/7)

- Características gerais:
 - Sistema de tipagem mista.
 - Construtores e destrutores.
 - Cuidadoso controle de acesso a entidades de classe.
- Herança:
 - Uma classe não necessita ser subclasse de nenhuma classe.
- Controle de acesso para membro são:
 - `private` (visível apenas na classe e “amigos”).
 - `public` (visível nas subclasses e clientes).
 - `protected` (visível na classe e nas subclasses).

C++ (2/7)

- Em adição, o processo de hierarquização em subclasses pode ser declarado com controles de acessos, os quais definem mudanças potenciais no acesso por subclasses.
- Herança múltipla é permitida.
- Binding dinâmico:
 - Um método pode ser definido como virtual, o que significa que ele pode ser chamado através de variáveis polimórficas e dinamicamente ligados (*binding*) a mensagem.
 - Uma função virtual pura não tem definição alguma.
 - Uma classe que tem ao menos uma função virtual pura é uma classe abstrata.

C++ (3/7)

- Avaliação:
 - Provê extensivo controle de acesso.

```
class Classe1 {  
    private:  
        int a;  
        float x;  
    protected:  
        int b;  
        float y;  
    public:  
        int c;  
        float z;  
};
```

```
class Subclasse1: public Classe1  
{...}
```

b e y: protegidos

c e z: públicos

a e x: inalcançáveis

```
class subclasse2: private classe1  
{...}
```

b, c, y e z: privados

a e x: inalcançáveis

C++ (4/7)

- Avaliação:
 - O programador deve decidir em tempo de projeto, quais métodos terão *binding* estático e quais terão *binding* dinâmico.
 - *Binding* estático é mais rápido.

```
class Classe1 {
public:
    void show() { cout << "Classe1"; }
};

class Classe2 : public Classe1 {
public: void show() { cout << "Classe2"; }
};

int main() {
    Classe1 *a = new Classe2();
    a->show(); // estático "Classe1"
}
```

```
class Classe1 {
public:
    virtual void show() { cout << "Classe1"; }
};

class Classe2 : public Classe1 {
public: void show() { cout << "Classe2"; }
};

int main() {
    Classe1 *a = new Classe2();
    a->show(); // dinâmico "Classe2"
}
```

C++ (5/7)

- Avaliação:
 - Provê herança múltipla.

```
class Classe1 {  
    public: virtual void show() { cout << "Classe1"; }  
};
```

```
class Classe2 {  
    public: virtual void show() { cout << "Classe2"; }  
};
```

```
class Classe3 : public Classe1, Classe2 {  
    public: void show() { Classe2::show(); }  
};
```

```
int main() {  
    Classe1 *a = new Classe3();  
    a->show(); // Classe2  
}
```

C++ (6/7)

```
class lista_ligada_simples {
    class vertice {
        friend class lista_ligada_simples;
    private:
        vertice *ligacao;
        int conteudos;
    };
    private:
        vertice *topo;
    public:
        lista_ligada_simples() {topo = 0};
        void insere_no_topo(int);
        void insere_no_fim(int);
        int remove_do_topo();
        int vazio();
};
```

C++ (7/7)

```
class pilha: public lista_ligada_simples {
public:
    pilha() {}
    void push(int valor){
        lista_ligada_simples::insere_no_topo(valor);
    }
    int top(){
        return lista_ligada_simples::remove_do_topo();
    }
};
```

Java (1/3)

- Características gerais:
 - Todos os dados são objetos, exceto os tipos primitivos.
 - Todos os tipos primitivos possuem classes empacotadoras que armazenam um valor (de dados):
 - `int` → `Integer`
 - `double` → `Double`
 - `char` → `Character`
 - Todos os objetos são dinâmicos no *heap*, são referenciados através de referência, e a maioria é alocada com `new`.
 - Objetos empacotados diretamente são armazenados no pool:
 - `String s1 = "abc"; // pool`
 - `String s2 = "abc"; // pool, igual ao s1`
 - `String s3 = new String("abc"); // heap`

Java (2/3)

- Herança:
 - Somente herança simples, mas existe uma categoria de classe abstrata que provê alguns dos benefícios de herança múltipla (interface).
 - Uma Interface pode incluir apenas declarações de métodos e constantes nomeadas.
 - Métodos podem ser final (não podem ser sobrepostos).
 - **public class** Clock **extends** Applet **implements** Runnable
- Binding Dinâmico:
 - Em Java, todas as mensagens tem *binding* dinâmico aos métodos, exceto quando método é final.

Java (3/3)

- Encapsulamento:
 - Dois compostos, classes e pacotes.
 - Pacotes proveem um *contêiner* para classes que são relacionadas.
 - Entidades definidas sem um modificador de escopo (*access*) tem o escopo do pacote, que os faz visíveis através do pacote no qual eles são definidos.
 - Toda classe num pacote é amiga para as entidades no escopo do pacote (que se encontram noutra lugar no pacote).