Orientação a Objetos

Encapsulando dados e algoritmos

- Procedimentos: escondem detalhes da computação tornando código mais modular
- Anos 60 e 70(!!!): tem início movimento no sentido de esconder detalhes da representação dos dados

  → abstração de dados
- Uma das linguagens mais importantes da época, ADA, tinha como um de seus pontos fortes o suporte a abstração de dados e polimorfismo
  - ADA é OO?

Abstração de dados

- Objetivo é esconder a representação dos dados
- Permite ao usuário pensar em termos das características ABSTRATAS dos tipos que pretende manipular nos programas e não da expressão concreta de uma representação na máquina
- Ex: inteiros
  - Ponto de vista abstrato: inteiros da matemática, com suas propriedades
  - Ponto de vista concreto: sequências de 32 bits (precisão limitada) ou sequência de caracteres (precisão ilimitada)
- Pontos
  - Abstrato: localização no plano
  - Concreto: registro com dois campos: *coordenadaX* e *coordenadaY* ou registro com dois campos *módulo* e *ângulo*

Abstração de dados: a noção de objetos

- Objeto é um conjunto com um estado interno (potencialmente vazio) e um conjunto de procedimentos de interface bem definida
  - Alteração do estado interno
  - Produção de valores
- Acesso aos dados apenas é possível pela *interface* de funções
  - “Perfil” da função garante abtração
  - Mudança de representação interna pode ser feita sem mudança de código de usuários do objeto, SE MANTIDO O PERFIL DAS FUNÇÕES
  - Funções devem garantir que modificações no estado sempre produzem novos estados consistentes.

Objetos vs programação funcional

- Programação funcional
  - Centrada na definição de funções
  - Em seu modo mais puro, sem efeitos colaterais
  - Processamento visto como fluxo de dados através de funções matemáticas para produzir um resultado
  - Muito apropriado para escopos onde temos uma visão matemática madura do problema
- Objetos
  - Centrada na definição de conjuntos de dados
  - Efeitos colaterais são parte integrante
  - Encapsulamento garante abstração e manutenção da consistência
  - Mais adequado a escopos sem visão matemática madura: mais próximo de nossa visão de mundo

Um pouco mais de do modelo de funcionamento de objetos

- O nosso domínio de problema é visto como conjuntos de dados que reagem a "mensagens":
  - Mudança de estado interno
  - Envio de mensagens a outros objetos
  - Produção de um resultado

Um pouco de nomenclatura

- Objetos: unidade básica de nossa modelagem
  - Conjunto de informação e de processos para manipulação desta informação
- Métodos: descrição dos procedimentos que podem ser aplicados aos dados internos
  - Eventualmente este processo pode envolver comunicação com outros objetos
- Mensagens: invocação de um dos métodos de um objeto.
- Variáveis de instância:
  - Ambiente com valores do estado interno do objeto associados a nomes
  - Variáveis contém outros objetos

Orientação a objetos

- Até agora definimos programação “baseada” em objetos
- Programação *orientada a objetos*
  - estrutura da linguagem define categorias de objetos, chamadas *classes*
  - *Classes* são entidades que criam objetos (fábricas) com uma estrutura uniforme
    - Escopo interno com mesmos nomes
    - Mesmos métodos
  - Classes são dispostas em uma *hierarquia:*
    - Objetos de uma classe *filha* têm como base mesma estrutura da classe mãe (métodos, variáveis de instância) mas podem definir novas variáveis de instância e métodos
      - Incremento
      - Redefinição

Programando orientado a objetos

- Programação Orientada a Objetos é tratada extensivamente em curso específico
- Porém alguns detalhes são importantes

Cuidados com programação OO

- Em seu modelo puro, estrutura de classes é feita para herança de **IMPLEMENTAÇÃO!!!!!**
  - Variáveis de instância e métodos de objetos da classe filha (subclasse) incluem v.i. e métodos dos objetos da classe mãe (superclasse)
  - Duas classes independentes podem implementar métodos com mesmo nome e mesmo perfil mas códigos completamente diferentes
    - Classes separadas, sem relação de hierarquia
    - CLASSE != TIPO
- No caso geral, herança só se justifica com reutilização de código ou estrutura de dados
  - Em geral só com o primeiro caso

Programação OO em seu estado puro

- Tudo são objetos
  - Booleanos
  - Inteiros
  - Classes
  - Métodos
- Todo o processamento se dá por envio de mensagens
  - Aritmética
  - Controle de fluxo
  - Etc.

Poucas linguagens OO puras

- Condicionais
- Aritmética de máquina
  - Inteiros de 8, 16, 32 e 64 bits vs Inteiros
  - Ponto flutuante vs Números reais

OO Puras: é possível?

- Claro! ➔ Smalltalk
- Como?
  - Classes primitivas
    - Inteiros “pequenos” (subclasse de inteiros)
      - Computações eficientes
    - Reais “pequenos “ (subclasse de reais)
    - Booleanos
  - Controle de fluxo:
    - Resolução de mensagens + objetos tipo “closure”
    - Ex if-then-else em Smalltalk

      <booleano> ifTrue:<fechamento True> ifFalse:<fechamentoFalse>

      ➔ Na classe primitiva *True* o método ifTrue:ifFalse manda a mensagem “value” para o primeiro fechamento

      ➔ Na calsse primitiva *False* o método ifTrue:ifFalse manda a mensagem “value” para o segundo fechamento

OO Puras: é possível?

- Claro! ➔ Smalltalk
- Como?
  - Classes primitivas
    - Inteiros “pequenos” (subclasse de inteiros)
      - Computações eficientes
      - Overflow em conta acarreta em geração de inteiro geral
    - Reais “pequenos “ (subclasse de reais)
    - Booleanos

OO Puras: é possível?

- Claro! → Smalltalk
- Como?
  - Aritmética tem nova visão

    1 + 2 ~ 1.+(2)

  → Para classes de números “eficientes” quando número excede capacidade de representação a conta gera um número de classe mais geral.

  → Compilador garante implementação eficiente para números “pequenos”

OO Puras: é possível?

- Claro! ➔ Smalltalk
- Como?
  - Controle de fluxo:
    - Resolução de mensagens + objetos tipo “closure”
    - Ex if-then-else em Smalltalk

      <booleano> ifTrue:<fechamento True>

      ifFalse:<fechamentoFalse>

      ➔Na classe primitiva *True* o método ifTrue:ifFalse manda a mensagem “value” para o primeiro fechamento

      ➔Na classe primitiva *False* o método ifTrue:ifFalse manda a mensagem “value” para o segundo fechamento

      –

Vejamos agora a implementação em nosso sistema

- Nota: visão “hibrida” → iremos estender linguagem anterior
  - Inteiros e booleanos não são objetos
  - Algumas operações não são envio de mensagens

OO no duro mundo real: Smalltalk vs. Java vs. C++

Conceitos básicos

- Código de programas orientado a objetos organizado em *classes*
- Classes:
  - definição de unidades procedurais *(métodos)* **+**
  - definição de conjunto de dados (*variáveis de instância)*
- Durante a execução de um programa Classes são instanciadas em *Objetos.*
  - Cada objeto possui um conjunto de dados iguais aos especificados na sua Classe.

Conceitos básicos - 2

- Execução de processos corresponde a *envio de mensagens* a um objeto
- Cada mensagem deve corresponder a um *método* especificado na classe do objeto
- *Classes* organizadas em uma hierarquia (dag):]
  - uma classe pode ter uma ou mais *Superclasses.*
  - uma classe pode ter uma ou mais *Subclasses.*
- Uma classe estende sua superclasse: seus objetos também respondem aos métodos da superclasse e possuem as variáveis de instância especificadas na superclasse.

Implementação: conceitos básicos

- Um objeto é algo semelhante a um registro de linguagens procedurais
- Objeto armazena conteúdo das variáveis de instância
- Código dos métodos ocorre apenas uma vez em cada programa, instâncias do mesmo objeto compartilham implementação do método
- Objeto corresponde a um parâmetro implícito do método.

Exemplo

- sistema de interpretação de linguagem funcional (obs.: padrão “Interpreter”)
- Classe: OperadorEstrito
  - variáveis: lista de argumentos
  - método: avalia(lista_expressões, ambiente)
- Subclasses: OperadorIf, OperadorBegin, OperadorWhile
  - método avalia_com_valores(lista_valores)

Herança: reutilização de código

- em sua forma pura o objetivo da herança é reutilização de código
- alternativa: delegação
- vantagens da herança:
  - refinamento de comportamento com granularidade fina
  - redefinição de funcionalidade de métodos
- Classes abstratas
  - classes que não podem ser instanciadas em objetos
  - não possuem todos os métodos necessários (ex. classe OperadorEstrito)
  - classes abstratas sem métodos são apenas repositórios de variáveis de instância

Herança vs. Tipos

- tipos: conceito abstrato que indica operações que podem ser executadas
- linguagens orientadas a objeto: tipos podem estar associados ou não a classes.
- C++: tipo = classe
- Java: tipo = classe ou interface
- Smalltalk80: tipo = conjunto de métodos
- Definição de tipos tem relação com princípios de programação e com implementação de herança

Implementação de Herança

- Java e Smalltalk: busca dinâmica de métodos
  - representação do objeto indica sua classe
  - classe possui associação nome -> código do método
  - envio de mensagem: método procurado na classe e nas suas superclasses
- C++: execução direta do código
  - representação do objeto inclui endereço dos métodos de suas classes
  - envio de mensagem corresponde a chamada indireta em tempo de execução (tempo constante)

Tipos em C++

- implementação eficiente de envio de mensagem estabelece tipo=classe
- redefinição de métodos precisa ser explícita: virtual
- herança múltipla necessária a polimorfismo (classes abstratas)

Tipos em Smalltalk

- Classe objeto trata de mensagens não implementadas
- Tipo é apenas um conjunto de métodos, basta que classes do receptor implementem método de mesmo nome e número de argumentos
- Verificação de tipo feita dinamicamente
- Polimorfismo natural

Tipos em Java

- tipo = classe + *interface*
- classes *implementam* interfaces
- verificação estática de tipo + resolução dinâmica de métodos
- polimorfismo garantido por interfaces.
- *garante* que objetos entendam mensagens enviadas a eles

Conclusões

- herança: mecanismo de reutilização de código de granularidade fina
- herança se contrapõe a delegação
- tipos: mecanismo de verificação
- implementação eficiente de herança em C++ tem implicações na estrutura de tipos e na necessidade de herança múltipla
- Java consegue unir flexibilidade da resolução dinâmica às vantagens da tipagem estática