Tipos abstrados de dados e construções encapsuladas

Linguagens de Programação

Marco A L Barbosa

http://github.com/malbarbo/na-lp-copl

Conteúdo

O conceito de abstração

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que incluí apenas os atributos mais significativos
- A abstração é uma ferramenta poderosa contra a complexidade, o seu propósito é simplificar a programação

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que incluí apenas os atributos mais significativos
- A abstração é uma ferramenta poderosa contra a complexidade, o seu propósito é simplificar a programação
- Tipos de abstração

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que incluí apenas os atributos mais significativos
- A abstração é uma ferramenta poderosa contra a complexidade, o seu propósito é simplificar a programação
- Tipos de abstração
  - Abstração de processos: mais comum na forma de subprogramas
    - `sortInt(list, listLen);`

O conceito de abstração

- Uma **abstração** é uma visão ou representação de uma entidade que incluí apenas os atributos mais significativos
- A abstração é uma ferramenta poderosa contra a complexidade, o seu propósito é simplificar a programação
- Tipos de abstração
  - Abstração de processos: mais comum na forma de subprogramas
    - `sortInt(list, listLen);`
  - Abstração de dados: umas das mais profundas ideias nos últimos 50 anos

Introdução à abstração de dados

Introdução à abstração de dados

- Um **tipo abstrato de dado** é um tipo de dado que satisfaz duas condições

Introdução à abstração de dados

- Um **tipo abstrato de dado** é um tipo de dado que satisfaz duas condições
  - A representação dos objetos do tipo é escondida da unidade de programa que usa o tipo, portanto, as únicas operações diretas possíveis sobre os objetos do tipo são aquelas fornecidas na definição do tipo (encapsulamento / ocultação de informação)

Introdução à abstração de dados

- Um **tipo abstrato de dado** é um tipo de dado que satisfaz duas condições
  - A representação dos objetos do tipo é escondida da unidade de programa que usa o tipo, portanto, as únicas operações diretas possíveis sobre os objetos do tipo são aquelas fornecidas na definição do tipo (encapsulamento / ocultação de informação)
  - A declaração do tipo e dos protocolos das operações sobre objetos do tipo (interface do tipo) estão contidas em uma única unidade sintática

Introdução à abstração de dados

- Um **tipo abstrato de dado** é um tipo de dado que satisfaz duas condições
  - A representação dos objetos do tipo é escondida da unidade de programa que usa o tipo, portanto, as únicas operações diretas possíveis sobre os objetos do tipo são aquelas fornecidas na definição do tipo (encapsulamento / ocultação de informação)
  - A declaração do tipo e dos protocolos das operações sobre objetos do tipo (interface do tipo) estão contidas em uma única unidade sintática

- Vantagens da primeira condição: confiabilidade, os clientes não podem mudar a representação dos objetos diretamente. Evita colisões de nomes. Possibilidade de alterar a representação e implementação sem afetar os clientes

Introdução à abstração de dados

- Um **tipo abstrato de dado** é um tipo de dado que satisfaz duas condições
  - A representação dos objetos do tipo é escondida da unidade de programa que usa o tipo, portanto, as únicas operações diretas possíveis sobre os objetos do tipo são aquelas fornecidas na definição do tipo (encapsulamento / ocultação de informação)
  - A declaração do tipo e dos protocolos das operações sobre objetos do tipo (interface do tipo) estão contidas em uma única unidade sintática

- Vantagens da primeira condição: confiabilidade, os clientes não podem mudar a representação dos objetos diretamente. Evita colisões de nomes. Possibilidade de alterar a representação e implementação sem afetar os clientes
- Vantagens da segunda condição: compilação separada

Introdução à abstração de dados

- Exemplos

Introdução à abstração de dados

- Exemplos
  - Tipo float

Introdução à abstração de dados

- Exemplos
  - Tipo float
  - Tipo pilha, `create(stack)`, `destroy(stack)`, `empty(stack)`, `push(stack, element)`, `pop(stack)`, `top(stack)`

```
...
create(stk1);
push(stk1, color1);
push(stk1, color2);
if (! empty(stk1))
  temp = top(stk1);
...
```

Questões de projeto

Questões de projeto

- Requisitos da linguagem para TAD
  - Uma unidade sintática que encapsula a definição do tipo e dos protótipos das operações
  - Uma maneira de tornar o nomes de tipos visíveis para clientes do código e ocultar a implementação
  - Poucas operações padrões (se alguma) deve ser fornecida (além das fornecidas na definição do tipo)

Questões de projeto

- Requisitos da linguagem para TAD
  - Uma unidade sintática que encapsula a definição do tipo e dos protótipos das operações
  - Uma maneira de tornar o nomes de tipos visíveis para clientes do código e ocultar a implementação
  - Poucas operações padrões (se alguma) deve ser fornecida (além das fornecidas na definição do tipo)
- Qual é a forma do "recipiente" para a interface do tipo?
- Os tipos abstratos podem ser parametrizados?
- Quais mecanismos de controle de acesso são fornecidos e como eles são especificados?

Exemplos de linguagens

Exemplos de linguagens: Ada

- A construção de encapsulamento é chamada de pacote
  - pacote de especificação (define a interface do tipo)
  - pacote de corpo (implementação)

Exemplos de linguagens: Ada

- A construção de encapsulamento é chamada de pacote
  - pacote de especificação (define a interface do tipo)
  - pacote de corpo (implementação)
- Ocultação de informação
  - O pacote de especificação tem uma parte visível ao cliente e uma parte oculta (private)
  - Na parte visível ao cliente é feito a declaração do tipo abstrato, que pode conter também a representação dos tipos não ocultos
  - Na parte privada é especificada a representação do tipo abstrato

Exemplos de linguagens: Ada

- Motivos para ter uma parte privada no pacote de especificação
  - O compilador precisa saber a representação vendo apenas o pacote de especificação
  - O clientes precisam enxergar o nome do tipo, mas não a representação

Exemplos de linguagens: Ada

- Motivos para ter uma parte privada no pacote de especificação
  - O compilador precisa saber a representação vendo apenas o pacote de especificação
  - O clientes precisam enxergar o nome do tipo, mas não a representação
- Ter parte dos detalhes da implementação (a representação) no pacote de especificação não é bom

Exemplos de linguagens: Ada

- Motivos para ter uma parte privada no pacote de especificação
  - O compilador precisa saber a representação vendo apenas o pacote de especificação
  - O clientes precisam enxergar o nome do tipo, mas não a representação
- Ter parte dos detalhes da implementação (a representação) no pacote de especificação não é bom
- Um solução é fazer todos os TADs serem ponteiros. Mas esta solução tem problemas
  - Dificuldades com ponteiros
  - Comparação de objetos
  - O controle da alocação é perdido

```
package Stack_Pack is
-- As entidades visíveis (interface)
  type Stack_Type is limited private;
  Max_Size: constant := 100;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Integer);
  procedure Pop(Stk: in out Stack_Type);
  function Top(Stk: in Stack_Type) return Integer;
-- Parte que é oculta aos clientes
  private
    type List_Type is array (1..Max_Size) of Integer;
    type Stack_Type is record
      List: List_Type;
      Topsub: Integer range 0..Max_Size := 0;
    end record;
end Stack_Pack;
```

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk : in Stack_Type) return Boolean is
    begin
    return Stk,Topsub = 0;
    end Empty;
  ...
end Stack_Pack;

with Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type;
  begin
  Push(Stack, 42);
  Push(Stack, 17);
  Topone := Top(Stack);
  Pop(Stack);
  ...
  end Use_Stacks;
```

Exemplos de linguagens: C++

- C++ foi criado para adicionar orientação a objetos em C, portanto suporta TADs

Exemplos de linguagens: C++

- C++ foi criado para adicionar orientação a objetos em C, portanto suporta TADs
- Os mecanismos de encapsulamento são as classes e estruturas
  - Os dados são chamados de dados membros
  - As funções são chamadas de funções membros
  - Os membros podem ser da classe ou ou da instância
  - Todas as instâncias de uma classe compartilham um cópia das funções membros
  - Cada instância da classe tem sua cópia dos dados membros
  - As instâncias podem ser estáticas, dinâmica na pilha ou dinâmicas no heap (new e delete)
  - Uma função membro pode ser inline (cabeçalho e corpo juntos)

Exemplos de linguagens: C++

- Ocultação de informação
  - private, para entidades ocultas
  - public, para as interfaces
  - protected

Exemplos de linguagens: C++

- Ocultação de informação
  - private, para entidades ocultas
  - public, para as interfaces
  - protected
- Construtores: utilizados para inicializar uma instância da classe

Exemplos de linguagens: C++

- Ocultação de informação
  - private, para entidades ocultas
  - public, para as interfaces
  - protected
- Construtores: utilizados para inicializar uma instância da classe
- Destrutores: chamado implicitamente quando o tempo de vida da instância acaba

```
class Stack {
  private:
    int *stackPtr;
    int maxLen;
    int topPtr;
  public:
    Stack() {   // construtor
      stackPtr = new int [100];
      maxLen = 99;
      topPtr = -1;
    };
    ~Stack () { // destrutor
      delete [] stackPtr;
    };
    void push (int num) {...};
    void pop () {...};
    int top () {...};
    int empty () {...};
}
```

```
void main() {
  int topOne;
  Stack stk;
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  ...
}
```

```
// Stack.h - the header file for the Stack class
class Stack {
private:    //** These members are visible only to other
            //** members and friends (see Section 11.6.4)
  int *stackPtr;
  int maxLen;
  int topPtr;
public:     //** These members are visible to clients
  Stack();  //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

```
// Stack.cpp - the implementation file for Stack class
#include <iostream>
#include "Stack.h"
using std::count;

Stack::Stack() {
  stackPtr = new int[100];
  maxLen = 99;
  topPtr = -1;
}
...
void Stack::push(int number) {
...
}
...
```

Avaliação Ada e C++

- Expressividade similar
- Ambos fornecem mecanismos de encapsulamento e ocultação de informação
- Classes são tipos, os pacotes em Ada são mecanismos mais gerais de encapsulamento

Exemplos de linguagens: Java

- Similar ao C++, exceto que
  - Todos os tipos definidos pelos usuários são classes
  - Todos os objetos são alocados no heap e acessados através de referência
  - Os métodos precisam ser definidos na classe
  - Os modificadores de acesso são especificados em entidades, não em cláusulas
  - Não tem destrutor

```
class Stack {
    private int [] stackRef;
    private int maxLen;
    private int topIndex;
    public Stack() { // a constructor
      stackRef = new int [100];
      maxLen = 99;
      topPtr = -1;
    };
    public void push (int num) {...};
    public void pop () {...};
    public int top () {...};
    public boolean empty () {...};
}
```

```
class TestStack {
  public static void main(String[] args) {
    Stack myStack = new Stack();
    myStack.push(42);
    myStack.push(29);
    ...
    myStack.pop();
    ...
  }
}
```

Exemplos de linguagens: C#

- Baseado em C++ e Java
- Adiciona dois modificadores de acesso internal e protected internal
- As instâncias de classe são dinâmicas no heap
- Destrutores são raramente usados
- Estruturas são semelhantes as classes, mas não podem ter herança, são alocadas na pilha e acessadas como valores
- Suporte a propriedades, que é uma maneira de implementar getters e setters sem requerer a chamada de método explícita

```
public class Weather {
  public int DegreeDays {
    get {return degreeDays;}
    set {
      if(value < 0 || value > 30)
        Console.WriteLine("Value is out of range: {0}",
                          value);
      else degreeDays = value;
    }
  }
  private int degreeDays;
  ...
}
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Exemplos de linguagens: Ruby

- O mecanismo de encapsulamento são as classes
  - Capacidade semelhante as classes em C++ e Java
  - Os nomes das variáveis de instâncias começam com `@` e de classes com `@@`
  - Os métodos são declarados com as mesma sintaxe que as funções
  - O construtor é o `initialize`, que é chamado quando o método `new` da classe é chamado
  - As classes são dinâmicas

Exemplos de linguagens: Ruby

- O mecanismo de encapsulamento são as classes
  - Capacidade semelhante as classes em C++ e Java
  - Os nomes das variáveis de instâncias começam com `@` e de classes com `@@`
  - Os métodos são declarados com as mesma sintaxe que as funções
  - O construtor é o `initialize`, que é chamado quando o método `new` da classe é chamado
  - As classes são dinâmicas
- Ocultação de informação
  - Os membros das classes podem ser públicos ou privados

```ruby
class StackClass
  def initialize
    @stackRef = Array.new
    @maxLen = 100
    @topIndex = -1
  end

  def push(number)
    ...
  end
  def pop
    ...
  end
  def top
    ...
  end
  def empty
    ...
  end
end
```

```ruby
myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop
```

Tipos abstratos de dados parametrizados

Tipos abstratos de dados parametrizados

- Permite a criação de tipos abstratos de dados que podem armazenar dados de qualquer tipo
- Não é uma questão relacionada as linguagens dinâmicas
- Algumas linguagens com suporte a TAD parametrizados: Ada, C++, Java 5, C# 2005

Tipos abstratos de dados parametrizados em Ada

- Os pacotes podem ser parametrizados

```
generic
  Max_Size: Positive;
  type Element_Type is private;
package Generic_Stack is
  Type Stack_Type is limited private;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Element_Type);
  ...
end Generic_Stack;

Package Integer_Stack is new Generic_Stack(100, Integer);
Package Float_Stack is new Generic_Stack(100, Float);
```

Tipos abstratos de dados parametrizados em C++

- As classes podem ser declaradas como templates

```
template <class Type>
class Stack {
  private:
    Type *stackPtr; int maxLen; int topPtr;
  public:
    Stack(int size) {
      stackPtr = new Type[size];
      maxLen = 99;
      topPtr = -1;
    };
    void push (Type valye) {...};
    ...
}

Stack<int> s1;
Stack<float> s2;
```

Tipos abstratos de dados parametrizados em Java 5

- Antes da versão 5, as classes como `LinkedList` e `ArrayList` podiam armazenar qualquer objeto

Tipos abstratos de dados parametrizados em Java 5

- Antes da versão 5, as classes como `LinkedList` e `ArrayList` podiam armazenar qualquer objeto
- Existem 3 problemas com coleção de objetos
  - Todo objeto da coleção precisa da coerção quando é acessado
  - Não é possível fazer checagem de tipo quando os valores são adicionados
  - Não é possível inserir tipos primitivos nas coleções

Tipos abstratos de dados parametrizados em Java 5

- Antes da versão 5, as classes como `LinkedList` e `ArrayList` podiam armazenar qualquer objeto
- Existem 3 problemas com coleção de objetos
  - Todo objeto da coleção precisa da coerção quando é acessado
  - Não é possível fazer checagem de tipo quando os valores são adicionados
  - Não é possível inserir tipos primitivos nas coleções
- O Java 5 tentou resolver estes problemas, adicionado genéricos (e autoboxing) a linguagem
  - As classes genéricas resolveram o primeiro e o segundo problema, mas não o terceiro, porque os parâmetros genéricos tem quer classe
  - O autoboxing resolveu o terceiro problema

```
class Stack<T> {
    private T[] stackRef;
    private int maxLen;
    private int topIndex;
    ...
    public void push (T value) {...};
    ...
}

Stack<Integer> s1 = new Stack<Integer>();
int x = 10;
s1.push(x);
int y = s1.top();

Stack<Float> s2 = new Stack<Float>();
```

Tipos abstratos de dados parametrizados em C# 2005

- Assim como Java, nas primeiras versão do C# as coleção armazenavam objetos
- Classes genéricas foram adicionadas ao C# 2005.
- Diferente do Java, os elementos de coleções genéricas podem ser acessados através de índices

Construções de encapsulamento

Construções de encapsulamento

- Grandes softwares têm duas necessidades especiais
  - Alguma maneira de organização, além da simples divisão em subprogramas
  - Alguma maneira de realizar compilação parcial

Construções de encapsulamento

- Grandes softwares têm duas necessidades especiais
  - Alguma maneira de organização, além da simples divisão em subprogramas
  - Alguma maneira de realizar compilação parcial
- Solução: encapsulamento
  - Agrupar os códigos e dados logicamente relacionados em uma unidade que possa ser compilada separadamente
- Existem várias formas de encapsulamento

Construções de encapsulamento

- Subprogramas aninhados

Construções de encapsulamento

- Subprogramas aninhados
- Encapsulamento em C
  - Um ou mais subprogramas e tipos são colocados em arquivos que podem ser compilados independentemente
  - A interface é colocada em um arquivo de cabeçalho, e a implementação em outro arquivo
  - #include é utilizado para incluir o cabeçalho
  - O ligador não checa os tipos entre o cabeçalho e a implementação

Construções de encapsulamento

- Encapsulamento em C++
  - Permite definir arquivos de cabeçalho e implementação (semelhante ao C)
  - Os arquivos de cabeçalho de templates em geral incluem a declaração e a definição
  - Friends fornecem um mecanismo para permitir acesso a membros privados

Construções de encapsulamento

- Encapsulamento em C++
  - Permite definir arquivos de cabeçalho e implementação (semelhante ao C)
  - Os arquivos de cabeçalho de templates em geral incluem a declaração e a definição
  - Friends fornecem um mecanismo para permitir acesso a membros privados
- Pacotes Ada
  - Podem incluir várias declarações de tipos e subprogramas
  - Podem ser compilados separadamente
  - Os pacotes de especificação e corpo podem ser compilados separadamente

Construções de encapsulamento

- Assemblies em C#
  - Uma coleção de arquivos que aparentam ser uma DLL ou executável
  - Cada arquivo define um módulo que pode ser compilado separadamente
  - Uma DLL é uma coleção de classes e subprogramas que são ligados individualmente a um executável
  - Contém outras informações, como dependências e versão
  - Podem ser privados ou públicos
  - O modificador de acesso internal especifica que um membro é visível a todos no mesmo assembly

Encapsulamento de nomes

Encapsulamento de nomes

- Como desenvolvedores trabalhando independentemente podem criar nomes para variáveis, classes, etc, sem acidentalmente usar um nome já em uso?

Encapsulamento de nomes

- Como desenvolvedores trabalhando independentemente podem criar nomes para variáveis, classes, etc, sem acidentalmente usar um nome já em uso?
  - Usando um **encapsulamento de nome**, que cria um novo escopo de nomes

Encapsulamento de nomes

- Como desenvolvedores trabalhando independentemente podem criar nomes para variáveis, classes, etc, sem acidentalmente usar um nome já em uso?
  - Usando um **encapsulamento de nome**, que cria um novo escopo de nomes
- Namespaces em C++
- Pacotes em Java
- Pacotes em Ada
- Módulos em Ruby

Referências

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 11.