Disciplina de Modelos de Linguagens de Programação

prof. Leandro Krug Wives

1 Tipos Abstratos de Dados

Tipos Abstratos de Dados (TAD) são tipos definidos pelo usuário (programador) que atendem a restrições específicas. Antes de analisarmos detalhadamente o conceito de TAD, vamos estudar o conceito de abstração e a sua importância na programação. Após isso e da definição de TAD, vamos recapitular o conceito de encapsulamento, proteção e o princípio da ocultação dentro do contexto de TAD, destacando sua importância.

1.1 Conceito de Abstração

Do dicionário Aulete Digital, abstração é a "operação intelectual por meio da qual se separam, apenas no pensamento, elementos ou aspectos de uma totalidade que não podem subsistir isoladamente". Ela corresponde às "idéias ou conceitos produzidos por meio dessa operação".

Ou seja, a abstração permite representar uma coleção de entidades, focando somente nos atributos mais significativos ou importantes para determinada situação ou contexto.

Na programação de computadores, diz-se que a abstração é uma arma contra a complexidade, pois permite focar nos atributos essenciais e ignorar os demais. Tal idéia vai ao encontro dos princípios básicos do processo de criação de programas, que corresponde em:

- Determinar os elementos essenciais à solução do problema;
- Ignorar os elementos supérfluos em cada etapa do processo de criação.

Nesse caso, é importante realizar a separação dos aspectos relacionados com "o que" deve ser feito (utilização, objetivo) dos relacionados com "como" pode ser feito (implementação).

Exemplo:

- Exemplo de utilização ou necessidade: ordenação de um conjunto de valores;
- Exemplo de implementação: método InsertionSort, BubbleSort, QuickSort, etc.

Na programação, os seguintes tipos de abstração ocorrem:

- Abstração de Processos: funções abstraem valores (expressões) e procedimentos abstraem comandos. Ambos podem ser parametrizados na sua invocação.
- Abstração de Dados: representam os elementos que podem ser armazenados no computador e as operações válidas sobre eles. As abstrações de dados mais comuns são: tipos primitivos, tipos definidos pelo usuário e TAD.

1.2 TAD: Definição

Um Tipo Abstrato de Dados é um tipo de dado definido pelo usuário (programador) que serve para definir variáveis e outras entidades de programação. No entanto, é um tipo de dados que satisfaz as seguintes condições:

- Deve haver uma definição de tipo que indique sua estrutura e também as operações válidas sobre ele;
- A representação ou definição do tipo e das operações sobre entidades do mesmo tipo estão contidas em uma única unidade sintática;
- Outras unidades do programa (clientes) podem ter permissão para criar variáveis ou entidades com base no tipo;

- A representação das entidades criadas não pode ser visível às unidades do programa que usam o tipo, ou seja, escondem os mecanismos internos de seu funcionamento, permitindo interação somente através das operações “publicadas” por eles mesmos (através de encapsulamento);
- Podem ser definidos pelo usuário (ou estar disponíveis em bibliotecas), oferecendo um significante benefício ao estender o sistema de tipos primitivos da linguagem;
- São igualmente de primeira classe, e possuem os mesmos “privilégios” de um tipo primitivo (através de sobrecarga, polimorfismo).

Por esse motivo, o mecanismo de orientação a objetos favorece bastante a definição e o uso de TADs, visto que com ele é que muitas das condições anteriores são obtidas ou garantidas de maneira simples (o que não significa que outros paradigmas não possam definir ou usar TADs).

1.3 Características dos TAD

Os TAD apresentam algumas características ou vantagens:

- **Redução da carga conceitual**. Como escondem o que não é importante, abstraem os detalhes que não interessam ao cliente. Como consequência, oferecem maior legibilidade, expressividade e capacidade de expressão.

  Exemplo:

  ```
  Pilha p;
  p.pop;
  p.push;
  ```

- **Maior confiabilidade** (contenção de falhas), também pelo fato de restringirem o acesso à sua representação interna. Com isso, o código torna-se independente da representação interna e das eventuais mudanças que venham ocorrer na implementação do tipo. Como consequência, o código permanece seguro contra modificações, evitando acesso a trechos que não deveriam ser modificados.

- **Maior Portabilidade**.

- **Maior Ortogonalidade**.

2 Encapsulamento e ocultação

O encapsulamento é o mecanismo importante e que dá suporte à abstração de dados. Com ele é possível ocultar informações (princípio do *information hiding*), protegendo os detalhes internos de uma estrutura ou tipo de dados.

Ele é importante porque os programas foram ficando cada vez maiores com o tempo, e sua compreensão e organização tornou-se muito difícil. A primeira forma desenvolvida para resolver esse problema foi a divisão de um programa em subprogramas, mas ela não é suficiente. A recompilação de diferentes subprogramas pode ser uma operação custosa e demorada. Uma forma de minimizar este problema foi a organização de programas em diferentes módulos ou unidades sintáticas. Cada unidade ou container sintático pode representar ou agregar diferentes grupos de subprogramas e dados logicamente relacionados. As unidades sintáticas podem ser compiladas separadamente e agregadas em um único programa executável.

Outro problema relacionado é a segurança das informações (quem acessa o que e como). Quando um programa não possui divisões, todo o código pode acessar as diferentes estruturas. Com a subdivisão em unidades sintáticas, cada unidade pode ter um escopo e um nível de proteção diferenciado (módulos isolados).

A evolução disso exige com que as unidades sintáticas sejam descritas em dois níveis, separando a implementação da sua "interface" (especificação do que ela oferece e como se comunica com outras unidades, i.e., protocolo). Isso permite esconder os detalhes internos de uma estrutura ou tipo de dados.

Muitas linguagens não orientadas a objetos permitem separar a "interface" da implementação de uma estrutura sintática, mas as formas mais simples e consistentes para isso surgiram com as linguagens orientadas a objetos (i.e., com as classes).

Um TAD definido através de uma classe é capaz de representar uma organização lógica de dados e suas respectivas operações, separando seu uso (interface) de sua implementação. O código de descrição dos dados e operações pode ser compilado separadamente do seu uso. Um TAD encapsula um único tipo de dados e os subprogramas que provêem acesso (operações) a ele. Os mecanismos de controle de acesso garantem que o esquema interno de representação não será acessado de fora da classe. Com isso, variáveis poderão ser definidas com base no TAD, mas sua representação interna fica escondida.

2.1 Proteção de dados

Os mecanismos de proteção de dados permitem garantir que uma determinada coleção somente seja acessada pelas operações definidas sobre o tipo no ambiente encapsulado.

- Níveis de proteção: O ambiente encapsulado pode 'exportar' dados para outros ambientes, restringindo ou escondendo o acesso aos mesmos através de atributos de visibilidade, tais como *public*, *private*, *protected* (no CTS do .Net há também: family, assembly/internal).
- Exemplos de cápsulas:
  - Modelo procedural: unidades (units do pascal)
  - Modelo de objetos:
    - Classes (Java, C++, etc.)
    - Packages (Java), namespaces (.Net)
    - Assemblies (.Net)

3 Interfaces

Os controles de uma televisão (i.e., botões) são a interface entre você (usuário) e os componentes eletrônicos que ficam dentro da TV. Quando você usa um botão para aumentar o volume, quer simplesmente que isso aconteça, sem ter que se preocupar com "como" os componentes internos fazem isso acontecer.

Objetos, portanto, definem a sua interação com o mundo externo através de métodos que eles expõem. No caso de programação orientada a objetos, os métodos formam a interface do objeto com o mundo externo (classes clientes).

No caso *interfaces* Java, a ideia é a mesma, mas com uma semântica específica (mais determinada e abstrata). Uma interface em Java compreende um conjunto de métodos que não possuem corpos (sem implementação). Uma *interface* serve para especificar o comportamento de um objeto sem mostrar como ele efetivamente funciona (ou seja, indica as operações válidas, mas não como elas funcionam).

Exemplo:

```
interface Bicycle {

        void changeCadence(int newValue);

        void changeGear(int newValue);
```

```
        void speedUp(int increment);

        void applyBrakes(int decrement);
}
```

As interfaces servem, portanto, para estabelecer padrões (a serem seguidos por um conjunto de classes filhas) ou contratos (indicando ao mundo externo o que ele pode fazer com a classe). Tais contratos são validados e garantidos pelo compilador.

Uma interface não pode ser utilizada diretamente. Uma interface é abstrata (pois não especifica a implementação dos métodos) e deve ser implementada em outra classe (filha ou herdeira). Para implementar a interface, o nome da classe deve mudar, e você deve especificar a palavra-chave “implements” na declaração da nova classe:

```
class ACMEBicycle implements Bicycle {

   // mesmos métodos, mas contendo implementação que os faz funcionar.

}
```

A nova classe possui um comportamento mais formal, portanto.

Maiores informações em: http://java.sun.com/docs/books/tutorial/java/IandI/createinterface.html