Tipos de datos abstractos y conceptos de encapsulación

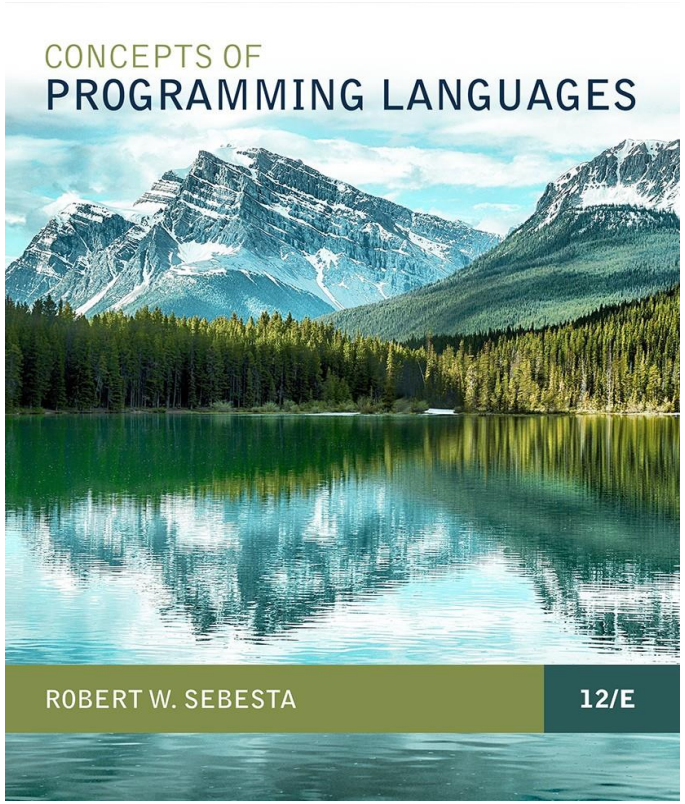
CAPÍTULO 11

PROF. CHRISTIAN VON LÜCKEN

AJUSTADO A LAS PRESENTACIONES DEL LIBRO "CONCEPTS OF PROGRAMMING LANGUAGES", ROBERT SEBESTA, 12/E. PEARSON. 2018. ISBN 0-321-49362-1

Tópicos

- El concepto de abstracción
- Introducción a la abstracción de datos.
- Cuestiones de diseño para Abstract Data Types
- Ejemplos de lenguajes
- TDA parametrizados
- Construcciones de encapsulación
- Nombrando encapsulaciones



Introducción

Una abstracción es una vista o representación de una entidad que incluye sólo los atributos más significativos

El concepto de abstracción es fundamental en programación(y ciencias de la computación)

Prácticamente todos los lenguajes de programación soportan la abstracción de procesos con subprogramas

Prácticamente todos los lenguajes de programación diseñadas desde 1980 soportan abstracción de datos

Introducción a la Abstracción de Datos

Un tipo de dato abstracto es un tipo de dato definido por el usuario que satisface dos condiciones:

1. La representación de los objetos del tipo están ocultos de las unidades de programa que usan tales objetos, entonces las únicas operaciones posibles son aquellas provistas por la definición del tipo
2. La representación de, y las operaciones sobre, objetos del tipo son definidos en una única unidad sintáctica. Otras unidades de programa pueden crear variables del tipo definido.

Ventajas de la abstracción de datos

Ventaja de la primera condición

- Confiabilidad, al esconder la representación del dato, el código del usuario no puede acceder directamente a los objetos del tipo o depender de la representación, permitiendo que la representación pueda modificarse sin afectar al código del usuario
- Reduce el rango de código y variables de los cuales el programador debe estar atento
- Los conflictos de nombre son menos probables

Ventaja de la segunda condición

- Provee un método para la organización del programa
- Ayuda a la modificabilidad (todo lo asociado con una estructura de datos va junta)
- Compilación separada

Requerimientos de lenguaje para ADTs

Una unidad sintáctica en el cual se encapsula la definición de tipo

Un método para hacer los nombres de tipo y los encabezados visibles a los clientes, mientras se esconden las definiciones actuales

Algunas operaciones primitivas deben ser contruidas en el procesador del lenguaje

Cuestiones de diseño

- Pueden los tipos abstractos ser parametrizados?
- Qué tipo de control de acceso se provee?
- La especificación del tipo esta físicamente separada de su implementación?

Ejemplos de lenguajes: Ada

La construcción de encapsulación es llamada paquete *packages*

- Paquete de especificación (la interface)
- Paquete de cuerpo (implementación de las entidades nombradas en la especificación)

Ocultamiento de información

- La representación de tipo aparece en una parte de la especificación llamada la parte privada
  - Más restringido con *limited private types*

Un ejemplo en Ada

```
package Stack_Pack is
 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; elem:in Integer);
 procedure pop(stk: in out stack_type);
 function top(stk: in stack_type) return Integer;

 private  -- hidden from clients
 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
```

Ejemplos de lenguajes: C++

Basado en el tipo **`struct`** de C y las clases de Simula 67

La clase (class) es el dispositivo de encapsulamiento

Una clase es un tipo

Todas las instancias de clases comparten una única copia de las funciones miembro

Cada instancia de una clase tiene su propia copia de los miembros de dato de la clase

Las instancias pueden ser static, stack dynamic, o heap dynamic

Ocultamiento de información

- *Private* clause para entidades cultas
- *Public* clause para entidades interface
- *Protected* clause para herencias

Ejemplo de lenguajes: C++ (Constructores)

- Funciones para inicializar los miembros de datos de las instancias (estas no crean los objetos)
- Pueden también asignar almacenamiento si parte del objeto es heap-dynamic
- Puede incluir parámetros para proveer parametrización de los objetos
- Implícitamente llamados cuando se crea una instancia
- Puede ser llamada explícitamente
- El nombre es el mismo que el nombre de la clase

Ejemplo de lenguajes: C++ (Destructores)

- Funciones para limpieza luego que una instancia se destruye; usualmente solo para reclamar almacenamiento de heap
- Implícitamente llamadas cuando el tiempo de vida del objeto termina
- Puede ser llamada explícitamente
- El nombre es el nombre de la clase precedido por un tilde (~)

Un ejemplo en C++

```
class Stack {
 private:
        int *stackPtr, maxLen, topPtr;
 public:
        Stack() { // a constructor
                stackPtr = new int [100];
                maxLen = 99;
                topPtr = -1;
        };
        ~Stack () {delete [] stackPtr;};
        void push (int number) {
          if (topSub == maxLen)
            cerr << "Error in push - stack is full\n";
          else stackPtr[++topSub] = number;
       };
        void pop () {…};
        int top () {…};
        int empty () {…};
}
```

Un archivo de encabezado para la clase `Stack`

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
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();
}
```

El archivo de código para `Stack`

```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Un ejemplo en C++ (continuación)

Friend functions or classes – proveer acceso a miembros privados a algunas unidades o funciones no relacionadas

- Necesario en C++

Evaluación de ADTs en C++ y Ada

Soporte de C++ para ADTs es similar al poder expresivo de Ada

Ambos proveen mecanismos efectivos para encapsulación y ocultamiento de información

Los paquetes de Ada son encapsulaciones más generales

Ejemplos de lenguajes: Java

Similar a C++, excepto:

- Todos los tipos definidos por el usuario son clases
- Todos los objetos son puestos en el heap y accedidos a través de variables referencia
- Entidades individuales en las clases tienen modificadores de control (private o public), en vez de clausulas
- Recolección de basura implícita de todos los objetos
- Java tiene un segundo mecanismo de alcance, package scope, el cual puede ser utilizado en vez de friends
  - Todas las entidades en todas las clases en un paquete que no tienen modificadores de control de acceso son visibles en el paquete

Un ejemplo en Java

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

Ejemplos de lenguajes: C#

- Basado en C++ y Java
- Agrega dos tipos de modificadores de acceso, *internal* y *protected internal*
- Todas las instancias de clase son heap dynamic
- Constructores por defecto están disponibles para todas las clases
- La recolección de basura es utilizada para la mayor parte de los objetos del heap por lo que raramente se utilizan destructores
- `structs` son clases livianas que no soportan herencia

Ejemplos de lenguajes: C# (cont.)

Tiene una solución común para acceder a los datos miembro: accessors (get, set)

C# provee *propiedades* como una forma de acceder a datos sin requerir una llamada explicita a método

C# Ejemplo de Propiedad

```
public class Weather {
   public int DegreeDays { //** DegreeDays is a property
      get {return degreeDays;}
      set {degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Abstract Data Types en Ruby

La construcción de encapsulación es la clase

Las variables locales tienen nombres “normales”

Las variables de instancia comienzan con el signo “at” (`@`)

Las variables de clase comienzan con dos signos “at” (`@@`)

Los métodos de instancia tienen la sintáxis de las funciones de Ruby (`def` … `end`)

Los constructores son llamados `initialize` (solo uno por clase)—llamados de manera implícita cuando se llama a `new`

- Si se necesitan más constructores, estos deben tener nombres diferentes y deben llamar explícitamente a `new`

Los miembros de clase pueden ser marcados como private o public, siendo públicos por defecto

Las clases son dinámicas

Abstract Data Types in Ruby (continued)

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

   def push(number)
     if @topIndex == @maxLen
       puts "Error in push – stack is full"
     else
       @topIndex = @topIndex + 1
       @stackRef[@topIndex] = number
     end
   end
   def pop … end
   def top … end
   def empty … end
end
```

TDAs parametrizados

Los TDAs parametrizados permiten diseñar un TDA que puede almacenar cualquier tipo de elemento

También conocidos como clases genéricas

C++, Ada, Java 5.0, y C# 2005 proveen soporte para DTAs parametrizados

DTAs parametrizados en Ada

Paquetes genéricos en Ada
- Hacer el tipo stack más flexible haciendo que el tipo del elemento y el tamaño sean genéricos

```
generic
Max_size: Positive;
type Elem_Type is Private;
package Generic_Stack is
…
function Top(Stk: in out StackType) return Elem_type;
…
end Generic_Stack;
```

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

DTAs parametrizados en C++

Las clases pueden ser de alguna forma genéricas con constructores parametrizados

```
Stack (int size) {
  stk_ptr = new int [size];
  max_len = size - 1;
  top = -1;
};
```

Una declaración para un objeto stack :

```
Stack stk(150);
```

ADTs parametrizados en C++ (continuación)

El tipo stack puede ser parametrizado haciendo a la clase un template

```
template <class Type>
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack() {  // Constructor for 100 elements
      stackPtr = new Type[100];
      maxLen = 99;
      topPtr = -1;
    }

      Stack(int size) {  // Constructor for a given number

        stackPtr = new Type[size];

        maxLen = size - 1;

        topSub = -1;

      }
  ...
}
```

- Instanciación: `Stack<int> myIntStack;`

Clases parametrizadas en Java 5.0

Los parámetros genéricos deben ser clases

Los tipos genéricos más comunes son los tipo colección, tales como `LinkedList` y `ArrayList`

Elimina la necesidad de hacer cast a objetos que son eliminados

Elimina el problema de tener múltiples tipos en una estructura

Los usuarios pueden definir clases genéricas

Las clases collection genéricas no pueden almacenar primitivos

La indexación no es soportada

Ejemplo del uso de una clase genérica predefinida:

```
ArrayList <Integer> myArray = new ArrayList <Integer> ();
  myArray.add(0, 47);  // Put an element with subscript 0 in it
```

Clases parametrizadas en Java 5.0 (continuación)

```
import java.util.*;
public class Stack2<T> {
  private ArrayList<T> stackRef;
  private int maxLen;
  public Stack2)( {
    stackRef = new ArrayList<T> ();
    maxLen = 99;
  }
  public void push(T newValue) {
    if (stackRef.size() == maxLen)
      System.out.println("Error in push – stack is full");
    else
      stackRef.add(newValue);
  ...
}
```

- Instanciación: `Stack2<string> myStack = new Stack2<string> ();`

Clases parametrizadas en C# 2005

- Similares a las de Java 5.0
- Predefinidas para Array, List, Stack, Queue, y Dictionary
- Elementos de las estructuras parametrizadas pueden ser accedidas a través del indexamiento

Constructores de encapsulación

Los programas grandes tienen dos necesidades especiales:

- Alguna forma de organización, además de la división en subprogramas
- Alguna forma de compilación parcial (unidades de compilación más pequeñas que todo el programa)

Solución obvia: un agrupamiento de subprogramas relacionados lógicamente en una unidad que puede compilarse separadamente (unidad de compilación)

Tales colecciones son conocidas como *encapsulación*

Subprogramas anidados

- Organizando programas anidando definiciones de subprogramas adentro de subprogramas lógicamente mayores y usarlos
- Los subprogramas anidados son soportados en Ada, Fortran 95, Python, JavaScript, y Ruby

Encapsulación en C

Los archivos conteniendo uno o más subprogramas pueden ser compilados en forma independiente

La interfaz es puesta en una archivo *header*

Problema 1: el linker no chequea los tipos entre el header y la implementación asociada

Problema 2: problemas inherentes a los punteros

Especificación del preprocesador `#include`

Paquetes de Ada

- La especificación de paquetes en Ada puede incluir cualquier número de declaraciones de datos y subprogramas
- Los paquetes de Ada pueden ser compilados de manera independiente
- La especificación de los paquetes y su cuerpo pueden ser compilados separadamente

Encapsulación en C++

Se puede definir archivos header y de código, similar a los de C

También las clases pueden ser utilizadas para la encapsulación

- La clase es utilizada como la interface (prototypes)
- Las definiciones de los miembros se definen en un archivo separado

*Friend*s provee una forma de permitir el acceso a miembros privados de una clase

C# Assemblies

Una colección de archivos que aparecen el los programas de aplicación puede ser una DLL (dynamic link library) o un ejecutable

Cada archivo contienen un módulo que puede ser compilado por separado

Una DLL

A DLL es una colección de clases y métodos que se pueden linkear individualmente a un programa en ejecución

C# tiene un modificador de acceso llamado `internal`; un miembro `internal` de una clase es visible a todas las clases del assembly en el cual aparece

Nombrando encapsulaciones

Programas grandes pueden definir muchos nombres globales; necesitan una manera de dividirlos en grupos lógicos

Una encapsulación con nombre es usada para crear un nuevo alcance para nombres

C++ Namespaces

- Pueden hace que cada librería tenga su propio espacio de nombre y nombres calificados son utilizados fuera con el espacio de nombres
- C# también incluye namespaces

Nombrando encapsulaciones (cont.)

Java Packages

- Packages pueden contener más de una definición de clase
- Los clientes de un paquete pueden usar el nombre calificado o la declaración ***`import`***

Ada Packages

- Packages son definidos en jerarquías que corresponden a la jerarquía de archivos
- Visibilidad desde una unidad de programa se obtiene con la clausula `with`

Nombrando encapsulaciones (cont.)

Ruby Modules:

- Las clases de Ruby son encapsulaciones con nombre, pero Ruby también tiene modulos

- Tipicamente encapsulan colecciones de constantes y métodos

- Los modulos no pueden ser instanciados o tener subclases y no pueden definir variables

- Los métodos definidos en un modulo deben incluir el nombre del modulo

- Se solicita el acceso a los contenidos de un modulo con el método `require`

Resumen

El concepto de DTA y el uso en el diseño de programas ha sido un hito en el desarrollo de lenguajes

Dos características principales de los DTAs son el empaquetado de datos con sus operaciones asociadas y ocultamiento de información

Ada provee paquetes que simulan DTAs

La abstracción de datos de C++ es provista por clases

La abstracción de datos de Java es similar a C++

Ada, C++, Java 5.0, y C# 2005 permiten DTAs parametrizados

C++, C#, Java, Ada y Ruby proveen nombrar encapsulaciones