Universidad Autónoma de Madrid

Escuela Politécnica Superior

Estructuras de Datos

Tipos Abstractos de Datos

Índice

Tipos Abstractos de Datos (I)

Un **Tipo Abstracto de Datos (TAD)** es un conjunto de datos con entidad propia (identidad definida) y un conjunto de operaciones (primitivas) aplicables sobre esos datos

Ejemplo de TAD: ComplexNumber

- **Datos:**
  - Parte real
  - Parte imaginaria
- **Primitivas:**
  - `Create`
  - `Destroy`
  - `Add`
  - `Multiply`
  - `Conjugate`
  - `GetModulus`

Tipos Abstractos de Datos (II)

Una **Estructura de Datos (EdD)** es un tipo usado para representar los datos en el TAD

Ejemplo: EdD para ComplexNumber (V1)

```
Real real_part;
Real imag_part;
```

Ejemplo: EdD para ComplexNumber (V2)

```
Real components[2];
```

La implementación de las primitivas dependerá de la EdD elegida

Tipos Abstractos de Datos (III)

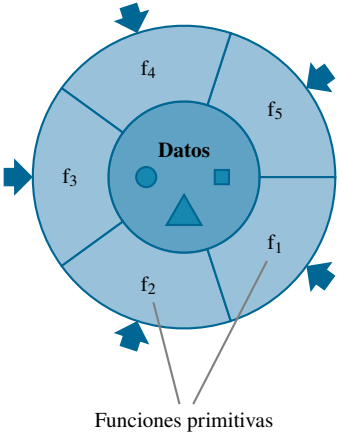
- Toda interacción con el TAD debe hacerse a través de las primitivas
- Cualquier función que opera sobre el TAD debe construirse realizando llamadas a las primitivas

Abstracción:

- La definición del TAD no especifica ni cómo se organizan los datos (estructura de datos) ni cómo se programan las primitivas (implementación)
- Lo único que el usuario necesita saber es lo que el TAD puede hacer (primitivas), pero no cómo se hace

Interfaz pública del TAD

El conjunto de funciones primitivas constituye la **interfaz pública** del TAD



Abstracción

Abstracción de los datos:

- La definición del TAD no especifica cómo se organizan/almacenan los datos
- Encapsulación: los datos están ocultos y solo se puede acceder a ellos a través de las primitivas

Abstracción de la funcionalidad:

- La definición del TAD no especifica cómo están implementadas las primitivas
- La interfaz pública determina las operaciones disponibles (pero no su implementación)

Diseño e implementación de un TAD

1. Especificación del TAD
   - Nombre
   - Datos
   - Funciones primitivas
   - Otras funciones (derivadas)
2. Definición de la EdD
   - Cómo organizar los datos
3. Implementación de las primitivas y funciones derivadas
   - Dependerá de la EdD elegida
4. Documentación
   - Debe tenerse especial cuidado con la interfaz pública

Ejemplo: ComplexNumber

Representa un número complejo de la forma $a + b \cdot i$

Datos

Dos números reales a y b que representan las partes real e imaginaria

Primitivas

```
ComplexNumber cn_create(Real re, Real im)
Void cn_free(ComplexNumber c)
Real cn_get_real_part(ComplexNumber c)
Real cn_get_imag_part(ComplexNumber c)
ComplexNumber cn_update(ComplexNumber c, Real re, Real im)
```

Ejemplo: ComplexNumber V1 (I)

Primitivas

```
ComplexNumber cn_create(Real re, Real im)
Void cn_free(ComplexNumber c)
Real cn_get_real_part(ComplexNumber c)
Real cn_get_imag_part(ComplexNumber c)
ComplexNumber cn_update(ComplexNumber c, Real re, Real im)
```

Otras funciones públicas

```
ComplexNumber cn_add(ComplexNumber c1, ComplexNumber c2)
ComplexNumber cn_multiply(ComplexNumber c1, ComplexNumber c2)
ComplexNumber cn_conjugate(ComplexNumber c)
Real cn_get_modulus(ComplexNumber c)
```

Ejemplo: ComplexNumber V1 (II)

Implementación de la función `cn_conjugate`:

Función `cn_conjugate`

```
ComplexNumber cn_conjugate(ComplexNumber c):
  if invalid c:
    return ERROR

  re = cn_get_real_part(c)
  im = cn_get_imag_part(c)

  return cn_create(re, -im)
```

Ejemplo: ComplexNumber V1 (III)

Implementación de la función `cn_multiply`:

Función `cn_multilpy`

```
ComplexNumber cn_multiply(ComplexNumber c1, ComplexNumber c2):
  if invalid c1 or invalid c2:
    return ERROR

  re1 = cn_get_real_part(c1)
  im1 = cn_get_imag_part(c1)
  re2 = cn_get_real_part(c2)
  im2 = cn_get_imag_part(c2)

  re = re1 * re2 - im1 * im2
  im = re1 * im2 + im1 * re2

  return cn_create(re, im)
```

Ejemplo: ComplexNumber V2 (I)

Primitivas

```
ComplexNumber cn_create(Real re, Real im)
Void cn_free(ComplexNumber c)
Real cn_get_real_part(ComplexNumber c)
Real cn_get_imag_part(ComplexNumber c)
Status cn_update(ComplexNumber c, Real re, Real im)
```

Otras funciones públicas

```
Status cn_add(ComplexNumber c1, ComplexNumber c2, ComplexNumber res)
Status cn_multiply(ComplexNumber c1, ComplexNumber c2, ComplexNumber res)
Status cn_conjugate(ComplexNumber c, ComplexNumber res)
Status cn_get_modulus(ComplexNumber c, Real res)
```

Ejemplo: ComplexNumber V2 (II)

Implementación de la función `cn_conjugate`**:**

Función `cn_conjugate`

```
Status cn_conjugate(ComplexNumber c, ComplexNumber res):
  if invalid c or invalid res:
    return ERROR

  re = cn_get_real_part(c)
  im = cn_get_imag_part(c)

  return cn_update(res, re, -im)
```

Ejemplo: ComplexNumber V2 (III)

Implementación de la función `cn_multiply`:

Función `cn_multilpy`

```
Status cn_multiply(ComplexNumber c1, ComplexNumber c2, ComplexNumber r):
  if invalid c1 or invalid c2 or invalid r:
    return ERROR

  re1 = cn_get_real_part(c1)
  im1 = cn_get_imag_part(c1)
  re2 = cn_get_real_part(c2)
  im2 = cn_get_imag_part(c2)

  re = re1 * re2 - im1 * im2
  im = re1 * im2 + im1 * re2

  return cn_update(r, re, im)
```

Ejemplo: Implementación en C de ComplexNumber (I)

Fichero complex_number.h

```c
/* Declaración de EdD. */

typedef struct _ComplexNumber ComplexNumber;

/* Cabeceras de funciones de la interfaz del TAD. */

ComplexNumber *cn_create(float re, float im);

void cn_free(ComplexNumber *pc);

float cn_get_real_part(const ComplexNumber *pc);

float cn_get_imag_part(const ComplexNumber *pc);
```

Ejemplo: Implementación en C de ComplexNumber (II)

Fichero complex_number.c

```c
#include "complex_number.h"

/* Definición de EdD. */
struct _ComplexNumber {
  float re, im;
};

/* Implementación de funciones públicas y privadas. */
ComplexNumber *cn_create(float re, float im) {
  ComplexNumber *pc = (ComplexNumber *)malloc(sizeof(ComplexNumber));
  if (!pc) return NULL;
  pc->re = re;
  pc->im = im;
  return pc;
}
```

...

Ejemplo: Implementación en C de ComplexNumber (III)

Fichero `complex_number.c`

```
...
void cn_free(ComplexNumber *pc) {
  free(pc);
}

float cn_get_real_part(const ComplexNumber *pc) {
  // Comprobar que PC no es NULL.
  return pc->re;
}

float cn_get_imag_part(const ComplexNumber *pc) {
  // Comprobar que PC no es NULL.
  return pc->im;
}
```

Ejemplo: Implementación en C de ComplexNumber (IV)

Reflexión importante: Si se quisiera cambiar la EdD para el TAD, por ejemplo a esta:

```
struct _ComplexNumber {
  float v[2];
};
```

¿Qué funciones sería necesario volver a implementar?

Ejemplo: Set (I)

Representa un conjunto de elementos

Datos

Una colección no ordenada de objetos de tipo `Element`, sin repeticiones

Primitivas

```
Set set_create()
Void set_free(Set c)
Integer set_cardinality(Set c)
Boolean set_is_empty(Set c)
Element set_get_element(Set c, Position p)
Status set_insert(Set c, Element e)
Boolean set_belongs_to(Set c, Element e)
Set set_union(Set a, Set b)
Set set_intersection(Set a, Set b)
```

Ejemplo: Set (II)

Implementación de la primitiva `set_belongs_to`:

Función `set_belongs_to`

```
Boolean set_belongs_to(Set c, Element e):
  if invalid c or invalid e:
    return FALSE

  for each Element x in c:
    if x == e:
      return TRUE

  return FALSE
```

Ejemplo: Set (III)

Implementación de la primitiva `set_insert`:

Función `set_insert`

```
Status set_insert(Set c, Element e):
  if invalid c or invalid e:
    return ERROR

  if set_belongs_to(c, e) == FALSE:
    # Añadir elemento al conjunto; depende de la defínción de la EdD.

  return OK
```

Ejemplo: Set (IV)

Implementación de la primitiva `set_union`:

Función `set_union`

```
Set set_union(Set a, Set b):
  Set c = set_create()

  for each Element x in a:
    if set_insert(c, x) == ERROR:
      set_free(c)
      return ERROR

  for each Element x in b:
    if set_insert(c, x) == ERROR:
      set_free(c)
      return ERROR

  return c
```

Consideraciones adicionales (I)

Objetivos de un TAD:

1. Encapsulación de los datos internos: los datos son manipulados únicamente a través de las primitivas
2. Encapsulación de la funcionalidad

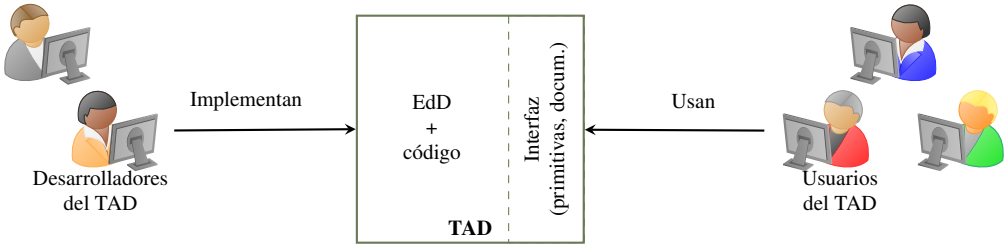
Consideraciones adicionales (II)

Desarrollador del TAD:

- Define la EdD
- Implementa las primitivas

Usuario del TAD:

- Trabaja sobre la interfaz pública del TAD (funciones públicas y documentación)



Implementación en C de un TAD

Fichero `my_adt.h`

```
// Definición de constantes.
#define ...

// Declaración de EdD.
typedef struct ...

// Cabeceras de funciones públicas.
...
```

Fichero `my_adt.c`

```
#include "my_adt.h"

// Definición de EdD.
struct ...

// Implementación de funciones públicas
// y privadas.
...
```

Otros ficheros `.c` deben incluir `my_adt.h` y acceder al TAD solo a través de las funciones declaradas en la interfaz

Ventajas e inconvenientes

Ventajas: El código es más modular y más portable

- Más fácil de desarrollar: los diferentes módulos se pueden desarrollar en paralelo
- Más fácil de depurar: los diferentes módulos se pueden probar de forma aislada
- Más fácil de actualizar: no existen grandes dependencias entre los diferentes módulos
- Más fácil de reutilizar: los diferentes módulos proporcionan funcionalidades independientes

Inconvenientes:

- La etapa de diseño es más compleja y requiere más tiempo y esfuerzo

Programación Orientada a Objetos

La evolución de los lenguajes de programación tiende hacia modelos cada vez más abstractos

Programación Orientada a Objetos

Entidades abstractas llamadas objetos que contienen:

- Atributos (datos)
- Métodos (operaciones)