

Programación II

Tema 1. Tipos Abstractos de Datos

Iván Cantador y Rosa M. Carro

Escuela Politécnica Superior

Universidad Autónoma de Madrid

- Tipos Abstractos de Datos (TAD)
- Ejemplos de TAD
 - TAD Cadena de caracteres
 - TAD Número complejo
 - TAD Conjunto
- Implicaciones de los TAD
- Programación Orientada a Objetos

- **Tipos Abstractos de Datos (TAD)**
- Ejemplos de TAD
 - TAD Cadena de caracteres
 - TAD Número complejo
 - TAD Conjunto
- Implicaciones de los TAD
- Programación Orientada a Objetos

- **Tipo Abstracto de Datos (TAD)**

- Conjunto de datos con entidad propia (identidad definida) y funciones primitivas (operaciones básicas) aplicables sobre esos datos
 - Ej. de TAD: **NumeroComplejo**
 - Datos: parte real + parte imaginaria
 - Primitivas: conjugado, módulo, suma, resta, ...
- Importante: funciones adicionales del TAD se construirán sólo a partir de sus primitivas

- **Tipo Abstracto de Datos (TAD)**

- Conjunto de datos con entidad propia (identidad definida) y funciones primitivas (operaciones básicas) aplicables sobre esos datos
 - Ej. de TAD: **NumeroComplejo**
 - Datos: parte real + parte imaginaria
 - Primitivas: conjugado, módulo, suma, resta, ...
 - Importante: funciones adicionales del TAD se construirán sólo a partir de sus primitivas

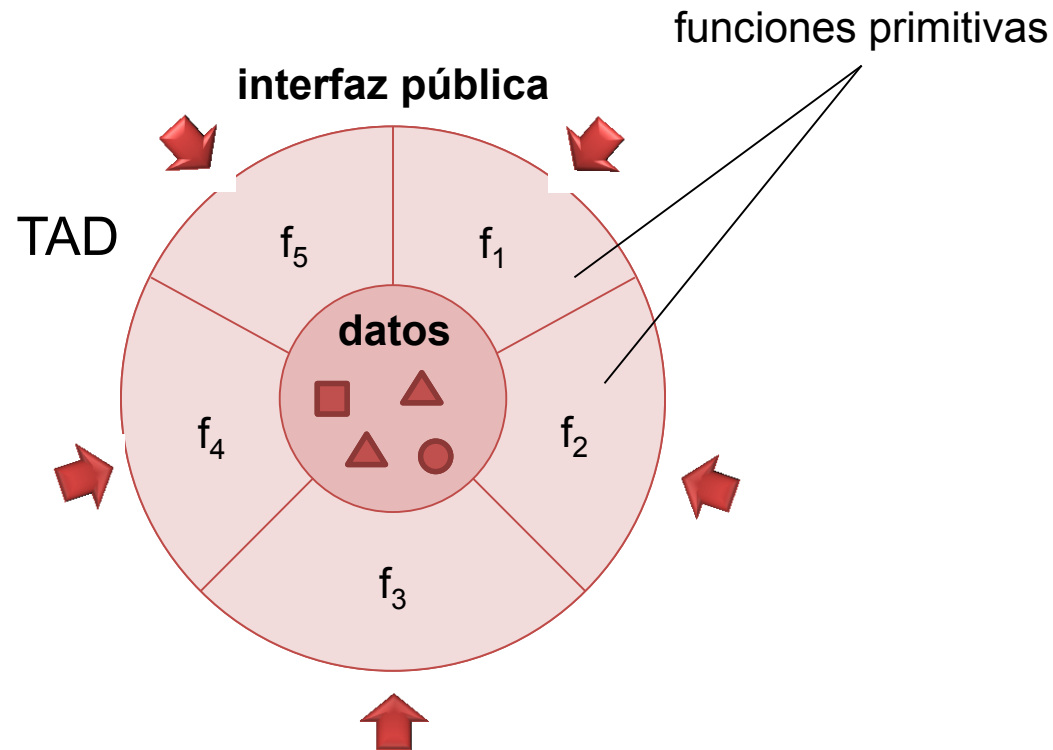
- **Estructura de Datos (EdD)**

- Tipos de datos concretos con los que representar los datos del TAD y a partir de los que implementar las funciones primitivas
 - Ej. de EdD para el TAD NumeroComplejo:
 - float real; float imaginaria;

- Todo TAD se definirá como un modelo abstracto de unos **elementos (datos)** y unas **operaciones (primitivas)** *sin tener en cuenta sus estructura e implementación particulares*
 - **TAD Número complejo**: módulo, conjugado, suma, resta, producto, cociente, ...
 - **TAD Conjunto** : tamaño, complemento, unión, intersección, ...
 - **TAD Lista**: inserción de un elemento, búsqueda de un elemento, eliminación de un elemento, visualización, ...
 - **TAD Grafo**: creación de un grafo vacío, inserción de un vértice, inserción de una arista, eliminación de un vértice, eliminación de una arista, distancia entre dos vértices, ...

- **Abstracciones en un TAD**

- Abstracción de los datos
- Abstracción de las funcionalidades



Tipos Abstractos de Datos (TAD)

7



- **Abstracciones en un TAD**

- **Abstracción de los datos**

- Encapsulamiento de los datos, sin permitir su acceso directo, que se realizará sólo mediante funcionalidades definidas al efecto
 - Ej.:

```
c = complejo_crear(2.0, -5.0)  
r = complejo_getParteReal(c)
```

- **Abstracción de las funcionalidades**

- Ocultación de la implementación de las funcionalidades a través de funciones primitivas, constituyendo la *interfaz* de acceso y manejo de los datos internos
 - Ej.:

```
m = complejo_modulo(c)  
c = complejo_sumar(c1, c2)
```


- **Abstracción de los datos**

- *Posibilita la definición y posterior **(re)utilización** de nuevos tipos de datos, dando a conocer sus posibles valores y las operaciones que trabajen sobre ellos, y evitando tener que conocer/entender su estructura interna*
 - El acceso a los valores de dichos datos se realiza sólo mediante las operaciones definidas sobre dicho TAD, sin preocuparnos de cómo son representados y tratados internamente.

- **Abstracción de las funcionalidades**

- *Permite la realización de determinadas **acciones** sobre los datos sin tener que conocer cómo se hacen, en qué tiempo y con qué recursos; Dichas acciones:*
 - Representan las operaciones más significativas
 - Tienen normalmente una relación casi directa entre las abstracciones funcionales obtenidas en el diseño descendente: subproblemas

1. Especificación del TAD

- ¿Qué **nombre** darle?
- ¿Qué **datos** (nombres, tipos, valores restringidos) tiene?
- ¿Cuáles (nombres, entradas, salidas) son sus **funciones primitivas**, que acceden/modifican sus atributos?
- ¿Cuáles son **funciones derivadas** que se pueden implementar a partir de las fundamentales?

2. Definición de la EdD

3. Atendiendo a la EdD elegida, pseudocódigo e implementación de las primitivas fundamentales y derivadas

4. Documentación del código (especialmente la *interfaz*)

- **TAD Cadena de caracteres**
 - Lista ordenada de caracteres
- **TAD Número complejo**
 - Número complejo, con parte real y parte imaginaria
- **TAD Conjunto**
 - Colección de elementos no repetidos
- *Algunos otros TAD*
 - TAD Pila
 - TAD Cola
 - TAD Lista enlazada
 - TAD Árbol
 - TAD Grafo

Este
tema

Programación II

- Tipos Abstractos de Datos (TAD)
- **Ejemplos de TAD**
 - **TAD Cadena de caracteres**
 - TAD Número complejo
 - TAD Conjunto
- Implicaciones de los TAD
- Programación Orientada a Objetos

- **EdD**

- Colección finita de caracteres pertenecientes a un alfabeto Σ

$$s = [c_1 \ c_2 \ \dots \ c_n] \ c_i \in \Sigma$$

$$\text{longitud}(s) \geq 0 \qquad \text{si } S = \emptyset \Rightarrow \text{longitud}(s) = 0$$

- Habrá una longitud máxima de cadena permitida CC_MAX y un carácter especial de fin de cadena

p.e. con CC_MAX=8 y carácter de fin de cadena @.

La cadena “PROG2” de longitud 5 sería:

P	R	O	G	2	@			
---	---	---	---	---	---	--	--	--

- **Funciones primitivas**

- **cc_crear, cc_liberar**
 - **cc_vacia, cc_llena**
 - **cc_insertarCaracter, cc_extraerCaracter**

- **CCCrear**

- Prototipo: `CC cc_crear ()`
- Descripción: crea y devuelve una cadena de caracteres “vacía”, es decir, sólo almacenando en ella el carácter especial de fin de cadena
- Entrada: ninguna
- Salida: cadena creada, vacía
(habría que especificar qué devolver en caso de error)
- Modifica: nada

- **CCLiberar**

- Prototipo: `void cc_liberar(CC s)`
- Descripción: libera una cadena dejándola “vacía”
- Entrada: cadena s
- Salida: ninguna
- Modifica: cadena s, dejándola vacía

- **CCVacía**

- Prototipo: `boolean cc_vacia(CC s)`
- Descripción: comprueba si una cadena de caracteres está o no vacía
- Entrada: cadena s
- Salida: verdadero si $S = \emptyset$, falso en caso contrario
- Modifica: nada

- **CCLlena**

- Prototipo: `boolean cc_llena (CC s)`
- Descripción: comprueba si una cadena de caracteres está o no llena, es decir si su longitud es igual o menor a CC_MAX
- Entrada: cadena s
- Salida: verdadero si la longitud de s es CC_MAX, falso en caso contrario
- Modifica: nada

- **CCInsertarCaracter**

- Prototipo: `status cc_insertar(CC s, car c, pos p)`
- Descripción: inserta un carácter dado en una posición determinada de una cadena (que queda modificada)
- Entrada: cadena `s`, carácter `c` a insertar, posición `p` en la que insertar `c`
- Salida: OK si la inserción se realiza con éxito, error en caso contrario
- Modifica: si devuelve OK, la cadena `s` contiene `c`; si devuelve error, nada.

$$s = [c_1 \ c_2 \ \dots \ c_p \ \dots \ c_n] \rightarrow s = [c_1 \ c_2 \ \dots \ \textcolor{red}{c} \ c_{p+1} \ \dots \ c_{n+1}]$$

- **CCExtraerCaracter**

- Prototipo: `status cc_extraer(CC s, car c, pos p)`
- Descripción: extrae el carácter situado en una posición determinada de una cadena (que queda modificada)
- Entrada: cadena `s`, variable `c` donde guardar el carácter extraído, posición `p` del carácter a extraer
- Salida: OK si la extracción se realiza con éxito, error en caso contrario
- Modifica: si devuelve OK, `c` contiene el carácter extraído y `s` ya no lo contiene; si devuelve error, nada

$$s = [c_1 \ c_2 \ \dots \ c_p \ \dots \ c_n] \rightarrow s = [c_1 \ c_2 \ \dots \ c_{n-1}]$$

- Funciones primitivas
 - **CCCrear:** `CC cc_crear()`
 - **CCVacía:** `boolean cc_vacia(CC s)`
 - **CCLlena:** `boolean cc_llena(CC s)`
 - **CCInsertarCaracter:** `status cc_insertar(CC s, car c, pos p)`
 - **CCExtraerCaracter:** `status cc_extraer(CC s, car c, pos p)`
- Funciones adicionales, que se pueden obtener usando las anteriores
 - **CCReemplazarCaracter:**
`status cc_reemplazar(CC s, car c, pos p)`
 - **CCLongitud:**
`entero cc_longitud(CC s)`



```
/* Versión 1: sin control de errores */  
status cc_reemplazar(CC s, car c, pos p)  
    cc_extraer(s, aux, p)  
    cc_insertar(s, c, p)  
    devolver OK
```

```
/* Versión 2: con control de errores, pero sin recuperación
   de la cadena original en caso de error (al insertar) */
status cc_reemplazar(CC s, car c, pos p)
    si cc_extraer(s, aux, p) = OK:
        si cc_insertar(s, c, p) = OK:
            devolver OK
        devolver ERROR
```

```
/* Versión 3: asumir que en el TAD, tras K extracciones de S
   se pueden realizar hasta K inserciones en S sin error */
status cc_reemplazar(CC s, car c, pos p)
    si cc_extraer(s, aux, p) = OK:
        cc_insertar(s, c, p)
        devolver OK
    devolver ERROR
```


- Funciones primitivas
 - **CCCrear:** `CC cc_crear()`
 - **CCVacía:** `boolean cc_vacia(CC s)`
 - **CCLlena:** `boolean cc_llena(CC s)`
 - **CCInsertarCaracter:** `status cc_insertar(CC s, car c, pos p)`
 - **CCExtraerCaracter:** `status cc_extraer(CC s, car c, pos p)`
- Funciones adicionales, que se pueden obtener usando las anteriores
 - **CCReemplazarCaracter:**
`status cc_reemplazar(CC s, car c, pos p)`
 - **CCLongitud:**
`entero cc_longitud(CC s)`



```
/* Versión 1: sin control de errores */
entero cc_longitud(CC s)
    longitud = 0
    s2 = cc_crear()

    mientras cc_vacia(s) = FALSO:
        cc_extraer(s, c, 0)
        cc_insertar(s2, c, 0)
        longitud = longitud + 1

    mientras cc_vacia(s2) = FALSO:           /* Restauramos s */
        cc_extraer(s2, c, 0)
        cc_insertar(s, c, 0)

    devolver longitud
```

```
/* Versión 2: con control de errores */
entero cc_longitud(CC s)
    longitud = 0
    s2 = cc_crear()
    si s2 = NULL:
        devolver -1

    mientras cc_vacia(s) = FALSO:
        cc_extraer(s, c, 0)
        si cc_insertar(s2, c, 0) = ERROR: /* Restauramos s si error */
            cc_insertar(s, c, 0)
            mientras cc_vacia(s2) = FALSO:
                cc_extraer(s2, c, 0)
                cc_insertar(s, c, 0)
            devolver -1
        longitud = longitud + 1

    mientras cc_vacia(s2) = FALSO: /* Restauramos s */
        cc_extraer(s2, c, 0)
        cc_insertar(s, c, 0)

    devolver longitud
```

- Tipos Abstractos de Datos (TAD)
- **Ejemplos de TAD**
 - TAD Cadena de caracteres
 - **TAD Número complejo**
 - TAD Conjunto
- Implicaciones de los TAD
- Programación Orientada a Objetos

- Datos

- Dos números reales 're' e 'im' asociados respectivamente a la parte real y a la parte imaginaria del número complejo

- **Funciones primitivas**

- **crear**: `Complejo complejo_crear(float r, float i)`
- **liberar**: `void complejo_liberar(Complejo c)`
- **parteReal**: `float complejo_getReal(Complejo c)`
- **parteImaginaria**: `float complejo_getImaginaria(Complejo c)`
- **actualizar**: `Complejo complejo_actualizar(Complejo c, float r, float i)`



- **Funciones primitivas**

- Complejo `complejo_crear`(float r, float i)
- void `complejo_liberar`(Complejo c)
- float `complejo_getReal`(Complejo c)
- float `complejo_getImaginaria`(Complejo c)
- status `complejo_actualizar`(Complejo c, float r, float i)

- **Funciones derivadas a partir de las primitivas**

- **conjugado:**

status `complejo_conjugado`(Complejo c, Complejo conj)

$$z = a + bi$$
$$\bar{z} = a - bi$$

- **suma:**

status `complejo_sumar`(Complejo c1, Complejo c2, Complejo res)

$$w = c + di$$
$$z + w = (a + c) + (b + d)i$$

- **resta:**

status `complejo_restar`(Complejo c1, Complejo c2, Complejo res)

$$z - w = (a - c) + (b - d)i$$

- **producto:**

status `complejo_multiplicar`(Complejo c1, Complejo c2, Complejo res)

$$z \cdot w = (a + bi) \cdot (c + di)$$
$$z \cdot w = (ac - bd) + (ad + bc)i$$

```
/* conj es la variable donde guardar el resultado de la función */
status complejo_conjugado(Complejo c, Complejo conj)
    r = complejo_getReal(c)
    i = complejo_getImaginaria(c)

    si r = NAN OR i = NAN:          /* NAN = not a number (valor no válido) */
        devolver ERROR

    si complejo_actualizar(conj, r, -i) = ERR:
        devolver ERROR
    devolver OK
```

$$z = a + bi$$
$$\bar{z} = a - bi$$

```
/* res es la variable donde guardar el resultado de la función */
status complejo_sumar(Complejo c1, Complejo c2, Complejo res)
    r1 = complejo_getReal(c1)
    i1 = complejo_getImaginaria(c1)
    r2 = complejo_getReal(c2)
    i2 = complejo_getImaginaria(c2)

    si r1 = NAN O i1 = NAN O r2 = NAN O i2 = NAN:
        devolver ERROR

    r = r1 + r2
    i = i1 + i2

    si complejo_actualizar(res, r, i) = ERR:
        devolver ERROR
    devolver OK
```

$$\begin{aligned}z &= a + bi \\ w &= c + di \\ z + w &= (a + c) + (b + d)i\end{aligned}$$


```
/* res es la variable donde guardar el resultado de la función */
status complejo_restar(Complejo c1, Complejo c2, Complejo res)
    r1 = complejo_getReal(c1)
    i1 = complejo_getImaginaria(c1)
    r2 = complejo_getReal(c2)
    i2 = complejo_getImaginaria(c2)

    si r1 = NAN O i1 = NAN O r2 = NAN O i2 = NAN:
        devolver ERROR

    r = r1 - r2
    i = i1 - i2

    si complejo_actualizar(res, r, i) = ERR:
        devolver ERROR
    devolver OK
```

$$\begin{aligned}z &= a + bi \\ w &= c + di \\ z - w &= (a - c) + (b - d)i\end{aligned}$$

```
/* res es la variable donde guardar el resultado de la función */
status complejo_multiplicar(Complejo c1, Complejo c2, Complejo res)
    r1 = complejo_getReal(c1)
    i1 = complejo_getImaginaria(c1)
    r2 = complejo_getReal(c2)
    i2 = complejo_getImaginaria(c2)

    si r1 = NAN O i1 = NAN O r2 = NAN O i2 = NAN:
        devolver ERROR

    r = r1*r2 - i1*i2
    i = r1*i2 + i1*r2

    si complejo_actualizar(res, r, i) = ERR:
        devolver ERROR
    devolver OK
```

$$\begin{aligned}z &= a + bi \\ w &= c + di \\ z \cdot w &= (a + bi) \cdot (c + di) \\ z \cdot w &= (ac - bd) + (ad + bc)i\end{aligned}$$

TAD Número complejo

- Posible implementación en C (sin control de errores)

```

/* En complejo.h */
typedef struct _Complejo Complejo;
-----
/* En complejo.c */
#include complejo.h
struct _Complejo {
    float re, im;
};

Complejo *complejo_crear(float r, float i) {
    Complejo *pc = (Complejo *) malloc(sizeof(Complejo));
    if (pc){          //if (pc!=NULL)
        pc->re = r;
        pc->im = i;
    }
    return pc;  //Devuelve NULL si error en el malloc
}

void complejo_liberar(Complejo *pc) {
    free(pc);
}

float complejo_getReal(const Complejo *pc) {
    return pc->re;
}

float complejo_getImaginaria(const Complejo *pc) {
    return pc->im;
}

```

• Funciones primitivas

- `Complejo complejo_crear(float r, float i)`
- `void complejo_liberar(Complejo c)`
- `float complejo_getReal(Complejo c)`
- `status complejo_actualizar(Complejo c, float r, float i)`

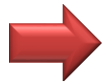


• Funciones derivadas a partir de las primitivas

- `status complejo_conjugado(Complejo c, Complejo res)`
- `status complejo_sumar(Complejo c1, Complejo c2 , Complejo res)`
- `status complejo_restar(Complejo c1, Complejo c2 , Complejo res)`
- `status complejo_multiplicar(Complejo c1, Complejo c2, Complejo res)`

• Si elegimos una EdD distinta para implementar el TAD:

```
struct _Complejo {  
    float re, im;  
};
```



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

¿Qué funciones habría que volver a implementar?

- Tipos Abstractos de Datos (TAD)
- **Ejemplos de TAD**
 - TAD Cadena de caracteres
 - TAD Número complejo
 - **TAD Conjunto**
- Implicaciones de los TAD
- Programación Orientada a Objetos

- Datos
 - Colección no ordenada y sin repeticiones de objetos de tipo *Elemento* (se asume que existe TAD *Elemento*)
- Funciones primitivas
 - `Conjunto conjunto_crear()`
 - `void conjunto_liberar(Conjunto c)`
 - `entero conjunto_cardinalidad(Conjunto c)`
 - `boolean conjunto_pertenece(Conjunto c, Elemento e)`
 - `status conjunto_insertar(Conjunto c, Elemento e)`
 - `status conjunto_union(Conjunto a, Conjunto b, Conjunto r)`
 - `status conjunto_interseccion(Conjunto a, Conjunto b, Conjunto r)`



```
boolean conjunto_pertenece(Conjunto c, Elemento e)
    para cada Elemento x de c:
        si x = e:           // Habría una función elemento_comparar(x,e)
            devolver TRUE
    devolver FALSE

status conjunto_insertar(Conjunto c, Elemento e)
    si conjunto_pertenece(c, e) = FALSE:
        c = c  $\cup$  {e}
        devolver OK
    devolver ERROR
```

- Datos
 - Colección no ordenada y sin repeticiones de objetos de tipo *Elemento* (se asume que existe TAD *Elemento*)
- Funciones primitivas
 - `Conjunto conjunto_crear()`
 - `void conjunto_liberar(Conjunto c)`
 - `entero conjunto_cardinalidad(Conjunto c)`
 - `boolean conjunto_pertenece(Conjunto c, Elemento e)`
 - `status conjunto_insertar(Conjunto c, Elemento e)`
 - `status conjunto_union(Conjunto a, Conjunto b, Conjunto r)`
 - `status conjunto_interseccion(Conjunto a, Conjunto b, Conjunto r)`




```
/* r es la variable donde guardar el resultado de la función */
status conjunto_union(Conjunto a, Conjunto b, Conjunto r)
    r= conjunto_crear ();
    Para cada Elemento e de a:
        si conjunto_insertar(r, e) = ERROR:
            conjunto_liberar(r)
            devolver ERROR
    Para cada Elemento e de b:
        si conjunto_pertenece (a, e) = FALSE:
            si conjunto_insertar(r, e) = ERROR:
                conjunto_liberar(r)
                devolver ERROR
    devolver OK
```

- Datos
 - Colección no ordenada y sin repeticiones de objetos de tipo *Elemento* (TAD que se asume existe)
- Funciones primitivas
 - `Conjunto conjunto_crear()`
 - `void conjunto_liberar(Conjunto c)`
 - `entero conjunto_cardinalidad(Conjunto c)`
 - `boolean conjunto_pertenece(Conjunto c, Elemento e)`
 - `status conjunto_insertar(Conjunto c, Elemento e)`
 - `status conjunto_union(Conjunto a, Conjunto b, Conjunto r)`
 - `status conjunto_interseccion(Conjunto a, Conjunto b, Conjunto r)`



```
/* r es la variable donde guardar el resultado de la función */  
status conjunto_interseccion(Conjunto a, Conjunto b, Conjunto r)  
    r = conjunto_crear()  
    Para cada Elemento e de a:  
        si conjunto_pertenece(b, e) = TRUE:  
            si conjunto_insertar(r, e) = ERROR:  
                conjunto_liberar(r)  
            devolver ERROR  
    devolver OK
```

- Tipos Abstractos de Datos (TAD)
- Ejemplos de TAD
 - TAD Cadena de caracteres
 - TAD Número complejo
 - TAD Conjunto
- **Implicaciones de los TAD**
- Programación Orientada a Objetos

- **Objetivos de los TAD**

- **Encapsulamiento (aislamiento) de los datos** internos, permitiendo su acceso y manejo sólo a través de sus primitivas
- **Ocultación de la implementación** interna de las funcionalidades, y uso de ellas a través de las primitivas

- **Consideraciones sobre los TAD**

- Desde el punto de vista del **desarrollador** del TAD:

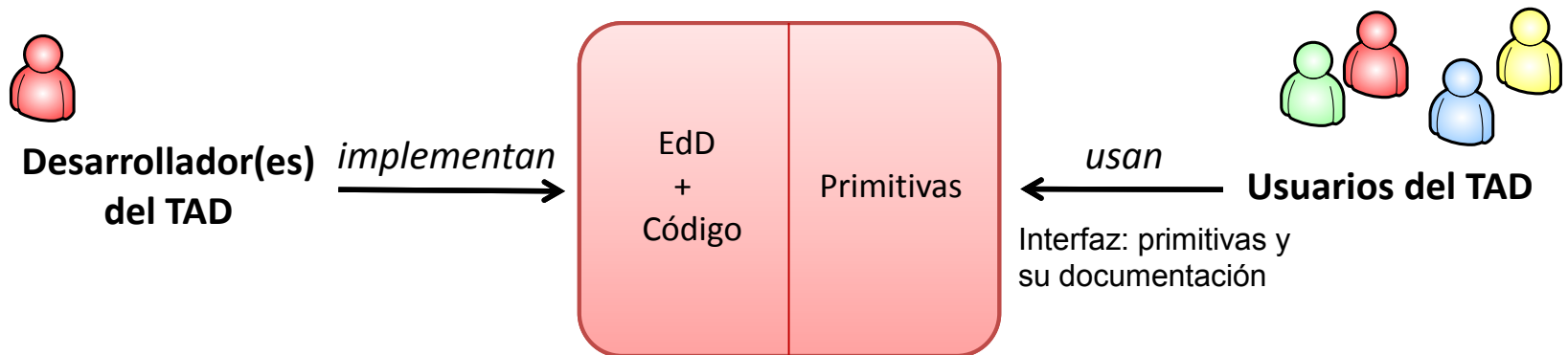
- ¿cómo implementar el TAD?

- Definir la EdD
 - Codificar las funciones primitivas

- Desde el punto de vista del **usuario** del TAD:

- ¿cómo usar el TAD?

- Trabajar sobre los prototipos y documentación de las primitivas



- **Implementación de un TAD en C**
 - Compuesta de un .c y un .h
 - En el .h
 - Definición (`#define`) de constantes
 - Declaración de la EdD (`typedef struct`) definida en el .c
 - Declaración de las primitivas públicas
 - En el .c
 - Inclusión (`#include`) del .h
 - Definición de la EdD (`struct`)
 - Implementación de las primitivas públicas y funciones privadas
 - En otros .c
 - Inclusión del .h
 - Uso del TAD sólo a través de las primitivas

• Implementación de un TAD en C

```
// complejo.h (a falta de incluir cabeceras y comentarios)
#ifndef COMPLEJO_H
#define COMPLEJO_H
#include "tipos.h"

typedef struct _Complejo Complejo;

Complejo *complejo_crear(float r, float i);
void complejo_liberar(Complejo *pc);
status complejo_actualizar(Complejo *pc, float r, float i);
float complejo_getReal(const Complejo *pc);
float complejo_geImaginaria(const Complejo *pc);
status complejo_conjugado(const Complejo *pc, Complejo *pcConj);
status complejo_sumar(const Complejo *pc1, const Complejo *pc2, Complejo *pcRes);
status complejo_restar(const Complejo *pc1, const Complejo *pc2, Complejo *pcRes);
#endif

// Comienzo de complejo.c (a falta de cabeceras y comentarios)
#include "complejo.h"
#include "tipos.h"

struct _Complejo {
    float re;
    float im;
};
```


- **Ventajas de usar TAD**

- **Facilidad de implementación**, permitiendo el desarrollo paralelo de las diferentes componentes de una aplicación
- **Facilidad de depuración de errores**, pues se pueden aislar las pruebas de componentes concretas de una aplicación
- **Facilidad de cambios**, debido a que hay menos dependencias fuertes entre las componentes de una aplicación
- **Facilidad de reutilización**, ya que componentes de una aplicación proporcionan funcionalidades independientes

- **Inconveniente de usar TAD**

- **Esfuerzo adicional en la etapa de diseño**, en la que el proceso de abstracción del TAD puede ser complicado

- **Consecuencia de usar TAD**

- Facilidad de reutilización y extensión del código al poseer mayor
 - **modularidad** (primitivas = “ladrillos”)
 - **portabilidad**
 - Abstracción de los datos y de las funcionalidades
 - Programación Orientada a Objetos (POO)

- Tipos Abstractos de Datos (TAD)
- Ejemplos de TAD
 - TAD Cadena de caracteres
 - TAD Número complejo
 - TAD Conjunto
- Implicaciones de los TAD
- **Programación Orientada a Objetos**

- La evolución de los lenguajes de programación tiende a introducir más abstracciones

**Lenguajes
de bajo nivel**

**Lenguajes
procedurales**

**Lenguajes
orientados a objetos**

(Lenguajes máquina,
ensamblador)

(Fortran, COBOL,
Pascal, C, ...)

(Smalltalk, C++,
Java, ...)

- La evolución de los lenguajes de programación tiende a introducir más abstracciones
 - Soporte para uso de TAD
 - **Lenguajes procedurales:** *tipos de datos abstractos han de ser definidos por el usuario*
 - **Lenguajes orientados a objetos:** *clases, datos y operaciones constituyen una entidad.*