



UNIVERSIDADE DA CORUÑA

Programación II

Introducción a los Tipos Abstractos de Datos (TAD)

“Los estudiantes deben convencerse de que la programación *no es un arte* misterioso, sino una *disciplina ingenieril*...

...la *abstracción* y la *especificación* deben ser las piezas clave para conseguir una *programación efectiva*”

Barbara Liskov

Profesora en el Massachusetts Institute of Technology (MIT)

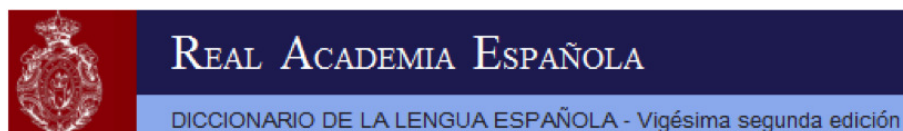
Medalla John von Neumann, 2004

Premio Turing, 2008

Computer Pioneer Award de la IEEE Computer Society, 2018

John Guttag

Profesor en el Massachusetts Institute of Technology (MIT)

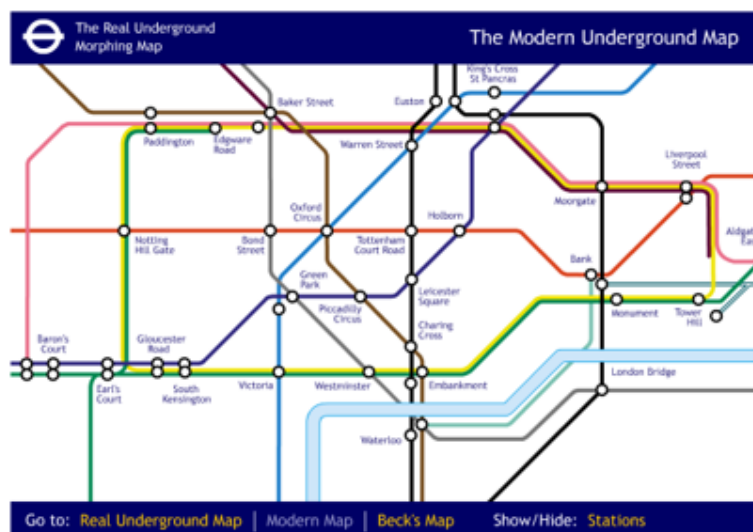
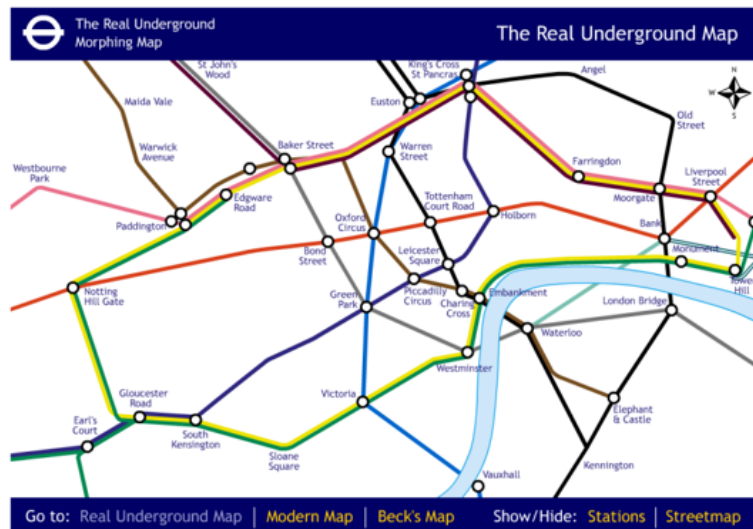
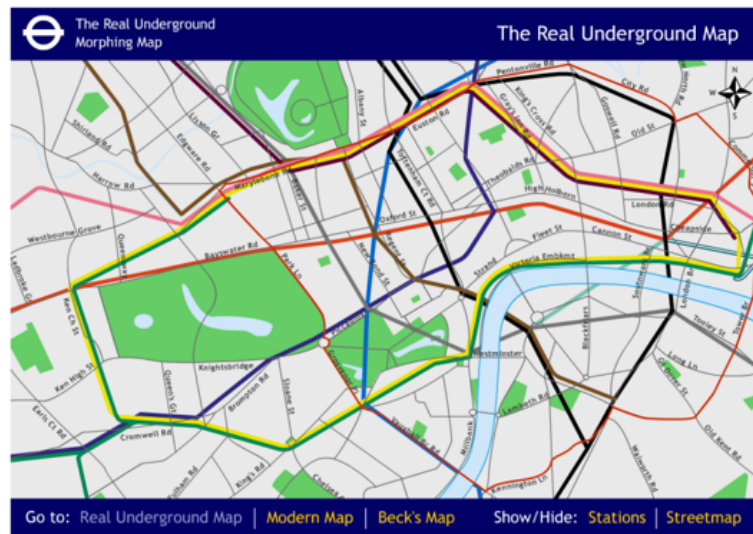


Del lat. *abstrahere*.

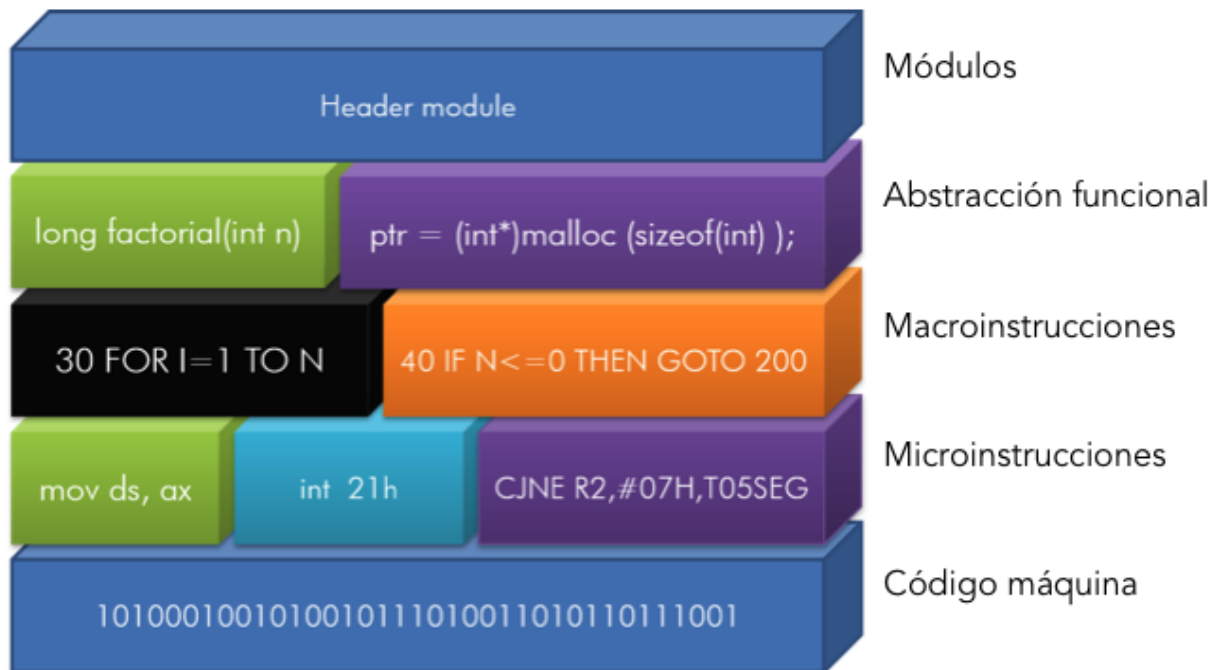
1. tr. Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción.

2. intr. Hacer caso omiso de algo, o dejarlo a un lado. Centremos la atención en lo esencial abstrayendo de consideraciones marginales.

Varios niveles de abstracción ¿Con cuál te quedas?



Niveles de abstracción funcional



Niveles de abstracción de datos

Objetivo: Conseguir que los tipos definidos por el programador se manejen como los incluidos en el propio lenguaje



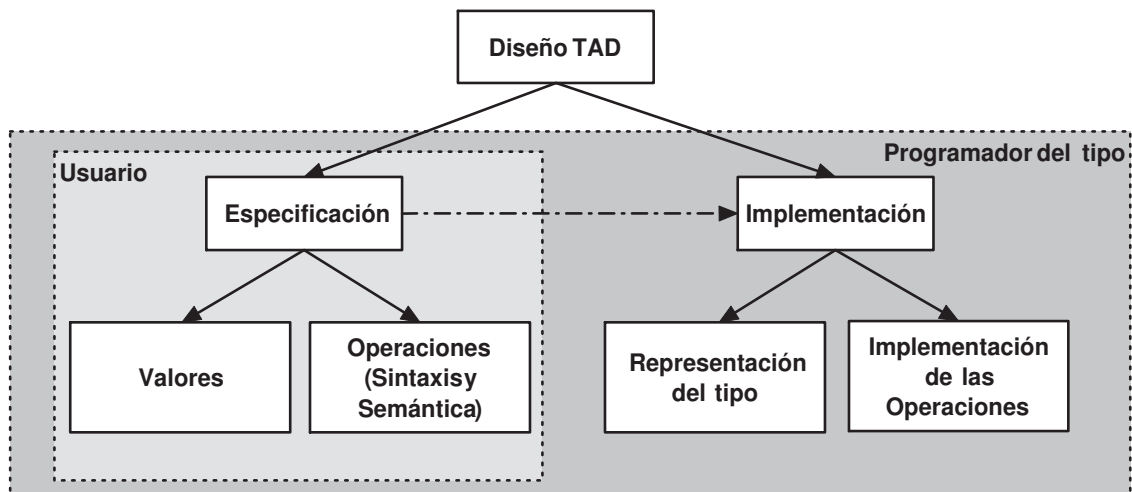
Definición de Tipo Abstracto de Datos (TAD)

John Guttag (1974):

Un tipo abstracto de datos es aquél **definido por el programador** que puede ser manipulado de forma **similar** a los **definidos por el sistema**.

Al igual que estos últimos, un tipo abstracto de datos corresponde a un **conjunto de valores** lícitos y de **operaciones** asociadas a los mismos, operaciones que se definen mediante una **especificación** que es **independiente de la implementación** de esos datos.

Niveles de especificación, implementación y uso de un TAD



Especificación formal de un TAD

Tipo	<i>Nombre del TAD</i>
Sintaxis	<i>Operaciones, tipo argumentos y resultado</i>
Semántica	<i>Comportamiento de las operaciones</i>

Hay que tener en cuenta que:

- Las reglas de comportamiento se intentarán en el orden indicado.
- Ciertos operadores no se definen. Pueden considerarse axiomas o *constructores* de los valores del tipo.
- La expresión del resultado puede ser recursiva, conteniendo referencias al mismo operador o a otros del TAD.
- Las expresiones pueden contener referencias a otros tipos ya definidos.

Ejemplo: definición formal del TAD correspondiente a los números naturales

Tipo	<i>Natural</i>
Sintaxis	<p>cero: $\rightarrow \text{Natural}$</p> <p>sucesor: $\text{Natural} \rightarrow \text{Natural}$</p> <p>escero: $\text{Natural} \rightarrow \text{Boolean}$</p> <p>igual: $\text{Natural} \times \text{Natural} \rightarrow \text{Boolean}$</p> <p>suma: $\text{Natural} \times \text{Natural} \rightarrow \text{Natural}$</p>
Semántica	<p>$\text{escero}(\text{cero}) \Rightarrow \text{true}$</p> <p>$\text{escero}(\text{sucesor}(n)) \Rightarrow \text{false}$</p> <p>$\text{igual}(\text{cero}, n) \Rightarrow \text{escero}(n)$</p> <p>$\text{igual}(\text{sucesor}(n), \text{cero}) \Rightarrow \text{false}$</p> <p>$\text{igual}(\text{sucesor}(n), \text{sucesor}(m)) \Rightarrow \text{igual}(n, m)$</p> <p>$\text{suma}(\text{cero}, n) \Rightarrow n$</p> <p>$\text{suma}(\text{sucesor}(n), m) \Rightarrow \text{sucesor}(\text{suma}(n, m))$</p>

donde “cero” y “sucesor” son los *constructores* del TAD.

Especificación informal de un TAD

TAD Nombre_del_tipo

VALORES: valores que pueden tomar los datos del tipo

OPERACIONES: nombre de las operaciones que los manipulan

Para cada operación (SINTAXIS y SEMÁNTICA):

Nombre_de_operación (tipo_de_argumento) \rightarrow tipo_de_resultado

{**Objetivo:** Descripción de la operación

Entrada: Descripción de los datos de entrada

Salida: Indica qué es lo que retorna la operación al invocarla

Precondiciones: Posibles excepciones. Características que tendrán que reunir los datos de entrada para que se realice bien la operación.

Poscondiciones: Indica un efecto lateral en la invocación a una función. Afirmaciones que podemos hacer sobre los datos después de que se ejecute la operación y que complete la información del objetivo y las salidas}

Pasos en la especificación de un TAD T

1. Seleccionar las operaciones, teniendo en cuenta para qué se va a utilizar dicho TAD T .
2. Clasificarlas:
 - *Constructoras*: Su resultado es de tipo T
 - *Generadoras*: Sólo con ellas es posible generar cualquier valor del tipo T y excluyendo cualquiera de ellas hay valores que no pueden ser generados.
 - *Modificadoras*: El resto
 - *Observadoras/Acceso*: Su resultado no es de tipo T .
 - *Destructoras*: Su resultado es de tipo T .
3. Comprobar la *consistencia* de las operaciones y su *completitud* (están las mínimas necesarias para manejar correctamente el nuevo tipo de datos)

Ejemplo: Especificación del Tipo Abstracto de Datos *Rational*

VALORES

- Concepto matemático de números racionales, es decir, un par de números enteros tal que el primero es el numerador y el segundo el denominador.

OPERACIONES (Sintáxis y Semántica)

- Constructoras Generadoras
 - **CreateRational** (**Entero**₁, **Entero**₂) → **Rational**
{*Objetivo*: Crea un número racional
Entrada:
Numerador del nuevo racional
Denominador del nuevo racional
Precondición: denominador distinto de 0
Salida: El número racional creado}
- Constructoras Modificadoras
 - **Sum** (**Rational**₁, **Rational**₂) → **Rational**₃
{*Objetivo*: Calcula la suma de dos número racionales
Entrada: Números racionales a sumar
Salida: Un nuevo racional suma de los números a la entrada}
- Observadoras
 - **Numerator** (**Rational**) → **Entero**
{*Objetivo*: Obtiene el numerador de un número racional
Entrada:
Número racional del que obtener el numerador
Salida: Numerador del número a la entrada}
 - **Denominator** (**Rational**) → **Entero**
{*Objetivo*: Obtiene el denominador de un número racional
Entrada:
Número racional del que obtener el denominador
Salida: Denominador del número a la entrada}

Definición Ampliada de Tipo Abstracto de Datos (TAD)

Ghezzi (1987):

Un nuevo tipo de dato se considera un TAD sólo si:

- El lenguaje proporciona algún método para permitir **asociar** la **representación** de los datos con las **operaciones** que los manipulan.
- La representación del nuevo tipo de dato así como la implementación de las operaciones pueden permanecer **ocultas** al resto de los módulos que los utilizan.

Es decir, para construir un tipo abstracto de datos, debemos ser capaces de:

- Exportar una definición de tipo.
- Proporcionar un conjunto de operaciones que puedan usarse para manipular los ejemplares de tipo.
- Proteger los datos asociados con el tipo de tal manera que se pueda operar con ellos sólo mediante las operaciones provistas.

Ejemplo de Tipo de dato como conjunto de valores

```
#include <stdio.h>

struct Rational {
    int num;
    int den;
};

int main () {
    struct Rational r1, r2, r3, r4, s;

    r1.num= 2; r1.den= 3;
    r2.num= 5; r2.den= 7;

    r3.num= 7; r3.den= 8;
    r4.num= 5; r4.den= 0;

    // s sum of r1 and r2
    s.num= r1.num * r2.den + r2.num * r1.den;
    s.den= r1.den * r2.den;
    printf("The sum is %d/%d\n", s.num, s.den);

    // s sum of r3 and r4
    s.num= r3.num * r4.den + r4.num * r3.den;
    s.den= r3.den * r4.den;
    printf("The sum is %d/%d\n", s.num, s.den);
}
```

Ejemplo de Tipo de dato como conjunto de valores y operaciones

```
#include <stdio.h>

struct Rational {
    int num;
    int den;
};

// Operacion para crear un racional
struct Rational createRational(int n, int d) {
    struct Rational temp;

    temp.num= n;
    temp.den= d;
    return temp;
}

// Operacion que retorna el numerador de un racional
int numerator(struct Rational r) {
    return r.num;
}

// Operacion que retorna el denominador de un racional
int denominator(struct Rational r) {
    return r.den;
}

// Operacion que retorna la suma de dos racionales
struct Rational sum(struct Rational r1, struct Rational r2) {
    struct Rational s;

    s.num= r1.num * r2.den + r2.num * r1.den;
    s.den= r1.den * r2.den;
    return s;
}

int main () {
    struct Rational r1, r2, r3, r4, s;

    r1= createRational(2,3); r2= createRational(5,7);
    r3= createRational(7,3); r4= createRational(5,4);

    s=sum(r1,r2); printf("The sum is %d/%d\n", numerator(s), denominator(s));
    s=sum(r3,r4); printf("The sum is %d/%d\n", numerator(s), denominator(s));
}
```

Encapsulando y compilando de forma independiente...utilizando como estructura de datos un registro (struct)...

rational_struct.h

```
#ifndef RATIONAL_STRUCT_H
#define RATIONAL_STRUCT_H

typedef struct Data Rational;
struct Data {
    int num;
    int den;
};

Rational createRational(int n, int d);
int numerator(Rational r);
int denominator(Rational r);
Rational sum(Rational r1, Rational r2);

#endif // RATIONAL_STRUCT_H
```

rational_struct.c

```
#include "rational_struct.h"

// Operacion para crear un racional
Rational createRational(int n, int d) {
    Rational temp;

    temp.num= n;
    temp.den= d;
    return temp;
}

// Operacion que retorna el numerador de un racional
int numerator(Rational r) {
    return r.num;
}

// Operacion que retorna el denominador de un racional
int denominator(Rational r) {
    return r.den;
}

// Operacion que retorna la suma de dos racionales
Rational sum(Rational r1, Rational r2) {
    Rational s;

    s.num= r1.num * r2.den + r2.num * r1.den;
    s.den= r1.den * r2.den;
    return s;
}
```

...utilizando como estructura de datos un puntero a un registro...

rational_pointer.h

```
#ifndef RATIONAL_POINTER_H
#define RATIONAL_POINTER_H

typedef struct Data* Rational;
struct Data {
    int num;
    int den;
};

Rational createRational(int n, int d);
int numerator(Rational r);
int denominator(Rational r);
Rational sum(Rational r1, Rational r2);

#endif // RATIONAL_POINTER_H
```


rational_pointer.c

```
#include <stdlib.h>
#include "rational_pointer.h"

// Operacion para crear un racional
Rational createRational(int n, int d) {
    Rational temp;

    temp= malloc(sizeof(*temp));
    temp->num= n;
    temp->den= d;
    return temp;
}

// Operacion que retorna el numerador de un racional
int numerator(Rational r) {
    return r->num;
}

// Operacion que retorna el denominador de un racional
int denominator(Rational r) {
    return r->den;
}

// Operacion que retorna la suma de dos racionales
Rational sum(Rational r1, Rational r2) {
    Rational s;

    s= malloc(sizeof(*s));
    s->num= r1->num * r2->den + r2->num * r1->den;
    s->den= r1->den * r2->den;
    return s;
}
```

...el programa que utiliza el tipo de datos es independiente de la representación del tipo de dato

```
#include <stdio.h>
#include "rational_pointer.h"

int main () {
    Rational r1, r2, r3, r4, s;

    r1 = createRational(2, 3);
    r2 = createRational(5, 7);

    r3 = createRational(7, 3);
    r4 = createRational(5, 4);

    s = sum(r1, r2);
    printf("The sum is %d/%d\n", numerator(s), denominator(s));
    s = sum(r3, r4);
    printf("The sum is %d/%d\n", numerator(s), denominator(s));
}
```

Ventajas del uso de un Tipo Abstracto de Datos (TAD)

1. Recogen mejor la semántica de los tipos. Al agrupar la representación junto a las operaciones que definen su comportamiento, y forzar a utilizar el TAD a través de estas operaciones se evitan errores en el manejo del tipo de datos (Por ejemplo, la división por 0 en el caso del TAD *Rational*).
2. Abstracción: Separa la especificación (**qué hace**) de la implementación (**cómo lo hace**). Los usuarios de un TAD no necesitan conocer sus detalles de implementación. Como consecuencia:
 - a) Se favorece la extensibilidad del código: Es posible *modificar* y *mejorar* la implementación del TAD sin afectar a los demás módulos que lo utilizan.
 - b) Aumenta la facilidad de uso.
 - c) Aumenta la legibilidad del código que usa el TAD.
3. Produce código reutilizable.
4. Favorece la ausencia de errores, al reutilizar código ya probado y forzar a utilizar la estructura de datos correctamente.