Abstracción de datos

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Contenidos

- Abstracción en programación
  - Mecanismos de abstracción: parametrización y especificación
- Tipos de abstracción en programación
  - Abstracción procedimental. Especificación de una AP
  - Abstracción de datos. TDA. Visiones de un TDA. Especificación e Implementación de un TDA. Función de Abstracción e Invariante de Representación
  - Abstracción en iteración
  - TDAs en C++
  - Abstracción por generalización. TDAs genéricos. Parametrización de tipos en C++

Abstracción en programación

- **Abstracción:** operación intelectual por la que se separa un rasgo o cualidad para analizarlo aisladamente y mejorar su comprensión, ignorando otros detalles.

- **Abstracción en la resolución de problemas:** ignorar detalles específicos buscando el esquema general del problema o de su solución, para obtener un perspectiva distinta.

- **Abstracción:** descomposición en la que se varía el nivel de detalle. Para que sea útil:
  - Todas las partes deben estar al mismo nivel
  - Cada parte debe poder abordarse por separado
  - Las soluciones de las partes deben poder unirse para obtener la solución final del problema

Mecanismos de abstracción en programación

- **Abstracción por parametrización:** se introducen parámetros para abstraer un número infinito de computaciones.

  Ej: sqrt(valor)

- **Abstracción por especificación:** permite abstraerse de la implementación de un subprograma asociándole una descripción precisa de su comportamiento

Ej: double sqrt(double a);

Requisitos: $a \geq 0$

Efecto: devuelve una aproximación de $\sqrt{a}$

La especificación es un comentario lo suficientemente definido y explícito como para poder usar el subprograma sin tener que conocer otros elementos

Abstracción por especificación

Suele expresarse en términos de:

- **Precondiciones**: condiciones necesarias y suficientes para que el subprograma se comporte como se ha previsto
- **Postcondiciones**: enunciados que se suponen ciertos tras la ejecución del subprograma si se cumplieron las precondiciones

```
int busca_minimo(int *vector, int tam)
/*
  precondición:
    tam>0
    vector es un vector con tam componentes
  postcondición:
    devuelve la posición del mínimo del vector
*/
```

Tipos de abstracción en programación

- **Abstracción procedimental:** definimos un conjunto de operaciones (procedimiento, subprograma, método) que se comporta como una operación simple

- **Abstracción de datos (TDA):** definimos un conjunto de datos y una serie de operaciones que caracterizan el comportamiento del conjunto. Las operaciones están vinculadas a los datos del tipo

- **Abstracción de iteración:** nos permite trabajar sobre colecciones de objetos sin preocuparnos de la forma concreta en la que se organizan

Abstracción procedimental

Permite abstraer un conjunto de operaciones de cómputo como una operación simple. Realiza la aplicación de un conjunto de entradas en las salidas, con posible modificación de las entradas.

- La identidad de los datos no es relevante para el diseño . Sólo nos interesa su número y tipo

- Con la abstracción por especificación, la implementación resulta irrelevante. Lo que importa es qué hace, no cómo lo hace.

  - **Localidad:** Para implementar una abstracción procedimental no es necesario conocer la implementación de otras abstracciones de las que se haga uso

  - **Modificabilidad:** Podemos cambiar la implementación de la abstracción procedimental sin que afecte a otras que la usen, siempre y cuando no modifiquemos la especificación

Especificación de una abstracción procedimental

- Cabecera (parte sintáctica): indica el nombre del subprograma y el número, orden y tipo de las entradas y salidas.

  Se suele adoptar la sintaxis de un lenguaje de programación. Ej: los prototipos de funciones en C++

- Cuerpo (parte semántica): compuesto por

1. Argumentos (parámetros)] Precondiciones
2. Requiere
3. Valores de retorno] Postcondiciones
4. Efecto
5. Excepciones

Especificación de una abstracción procedimental

1. **Argumentos:** explica el significado de cada parámetro de la abstracción procedimental (no el tipo, ya especificado en la cabecera)
   - Indicar restricciones sobre el conjunto de datos sobre los que puede operar el procedimiento
   - Indicar si cada argumento es modificado o no. Cuando un parámetro se modifique se indicará con "ES MODIFICADO"

2. **Requiere:** restricciones derivadas del uso del procedimiento no recogidas en Argumentos. P.ej.: la necesidad de que previamente se haya ejecutado otra abstracción procedimental

Especificación de una abstracción procedimental

3. **Valores de retorno:** descripción de qué valores devuelve la abstracción procedimental y qué significan

4. **Efecto:** Describe el comportamiento para las entradas válidas (según los requisitos). Deben indicarse las salidas generadas y las modificaciones producidas sobre los argumentos marcados como "ES MODIFICADO"
   - No se indica nada sobre el comportamiento del procedimiento cuando la entrada no cumple los requisitos
   - No se indica nada sobre cómo se realiza el efecto

5. **Excepciones:** Describe (si es necesario) el comportamiento cuando se da una circunstancia que no permita su finalización con éxito

Ejemplo

```
int busca_minimo(int *vector, int tam)
/*
  Argumentos:
    vector: array 1-D en el que hacer la búsqueda
    tam: número de elementos de vector
  Devuelve:
    índice del mínimo en el vector
  Precondiciones:
    tam > 0
    vector es un vector con tam componentes
  Efecto:
    Busca el elemento más pequeño en el array vector
    y devuelve su posición, es decir, el valor
      i / v[i] <= v[j] para 0 <= j < tam
*/
```

Especificación junto al código fuente

- Falta de herramientas para soportar y mantener el uso de especificaciones ➠ responsabilidad exclusiva del programador
- Necesidad de vincular código y especificación
- Incluirla entre comentarios en la parte de interfaz del código
- Usar una herramienta:
  - doc++ (http://docpp.sourceforge.net) Permite generar documentación de forma automática en distintos formatos (html, LaTeX,…) a partir del código fuente
  - doxygen (http://www.doxygen.org)

Especificación usando doxygen

- Toda la especificación se encierra entre /** ... */

```
/**
 ... texto ...
*/
```

- Se incluye una frase que describa toda la función: @brief
- Cada argumento va precedido de @param
- Los requisitos adicionales van precedidos de @pre
- Los valores de retorno van precedidos de @return
- La descripción del efecto sigue a @post

Especificación usando doxygen

```
/**
 @brief Calcula el índice del elemento máximo de un vector
 @param vector array 1-D en el que hacer la búsqueda
 @param tam número de elementos del array \a vector

 @pre  \a vector es un array 1-D con al menos \a tam componentes
 @pre  \a tam > 0
 @return el elemento más pequeño en el array \a vector, es decir,
   el valor i tal que v[i] <= v[j] para 0 <= j <  \a tam
*/

int IndiceMaximo(int *vector, int tam);
```

Especificación usando doxygen

Function Documentation

◆ IndiceMaximo()

```
int IndiceMaximo ( int *  vector,
                   int    tam
                 )
```

Calcula el índice del elemento máximo de un vector.

Parameters

vector array 1-D en el que hacer la búsqueda

tam número de elementos del array *vector*

Precondition

vector es un array 1-D con al menos *tam* componentes

tam > 0

Returns

el elemento más pequeño en el array *vector*, es decir, el valor i tal que v[i] <= v[j] para 0 <= j < *tam*

Abstracción de datos

- **Tipo de Dato Abstracto (TDA)**: Entidad abstracta formada por un conjunto de datos y una serie de operaciones asociadas

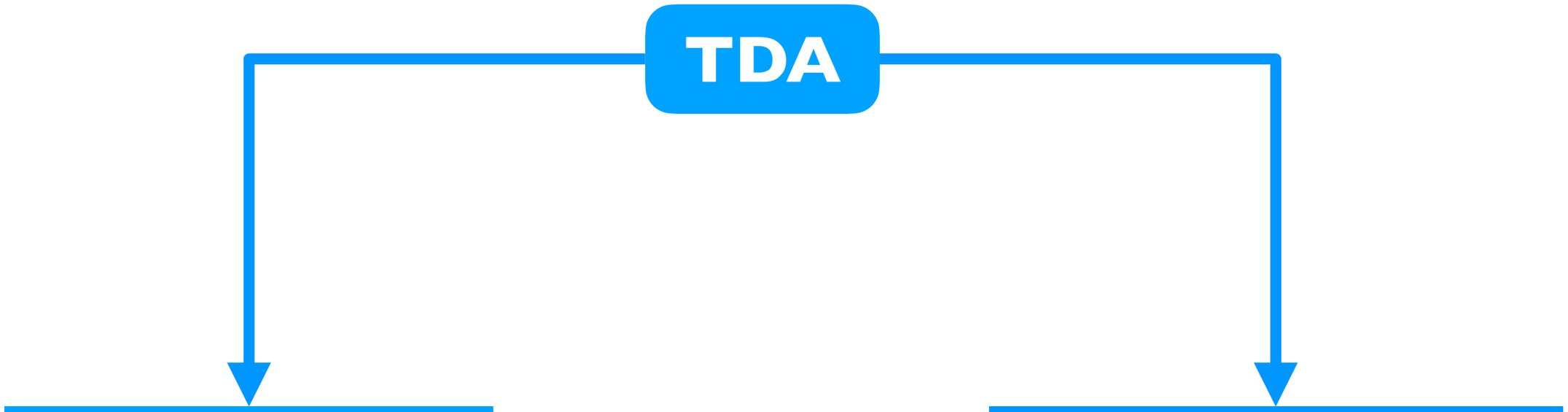
  Ej: los tipos de datos de C++

Conceptos:

- **Especificación**: descripción del comportamiento del TDA
- **Representación**: forma concreta en que se representan los datos en un lenguaje de programación para poder manipularlos
- **Implementación**: forma específica en que se desarrollan las operaciones

Visiones de un TDA

- Hay dos visiones de un TDA:
  - Visión externa: especificación
  - Visión interna: representación e implementación
- Ventajas de la distinción de visiones:
  - Se puede cambiar la visión interna sin afectar a la visión externa
  - Facilita la labor del desarrollador, permitiéndole centrarse en cada fase por separado



- Sintaxis
- Semántica

Accesible

- Representación
- Algoritmos

No accesible

Especificación de un TDA

- La especificación es ***esencial***. Define su comportamiento, pero no dice nada sobre su implementación
- Indica el tipo de entidades que modeliza, qué operaciones se les puede aplicar, cómo se usan y qué hacen
- Estructura de la especificación:

A. **Cabecera**: nombre del tipo y listado de las operaciones

B. **Definición**: descripción del comportamiento sin indicar la representación. Se debe indicar si es mutable o no. También se indica dónde residen los objetos

C. **Operaciones**: especificar las operaciones una a una como abstracciones procedimentales

TDA Fecha

```
/*
 TDA Fecha
 Fecha: constructor, dia, mes, año, siguiente, anterior,
 escribe, lee, menor, menor o igual

 Definición:
 Fecha representa fechas según el calendario occidental
 Son objetos mutables
 Residen en memoria estática

 Operaciones:
*/
```

TDA Fecha

```
/**
 @brief Constructor primitivo.
 @param f Objeto creado. Debe ser nulo.
 @param dia dia de la fecha. 0 < dia <= 31.
 @param mes mes de la fecha. 0 < mes <= 12.
 @param anio año de la fecha.
 @pre Los tres argumentos deben representar una fecha válida
 según el calendario occidental.
 @return Objeto Fecha correspondiente a la fecha dada por los argumentos.
 @doc
 Crea un objeto Fecha a partir de los argumentos.
 Devuelve el objeto creado sobre f.
 */
  void constructor (Fecha & f, int dia, int mes, int anio);

/**
 @brief Lee una fecha de una cadena.
 @param f  Objeto creado. Es MODIFICADO.
 @param cadena Cadena de caracteres que representa
 una fecha en formato dd/mm/aaaa.
 @return Objeto Fecha que representa la fecha leída en cadena.
 */
  void lee(Fecha & f, char cadena[]);
```

TDA Fecha

```
/**
 @brief Obtiene el día del objeto receptor.
 @param f Objeto receptor.
 @return Día del objeto f.
 */
  int dia(Fecha f);

/**
 @brief Obtiene el mes del objeto receptor.
 @param f Objeto receptor.
 @return  Mes del objeto f.
 */
  int mes(Fecha f);

/**
 @brief Obtiene el año del objeto receptor.
 @param f Objeto receptor.
 @return Año del objeto f.
 */
  int anio(Fecha f);
```

TDA Fecha

```
/**
 @brief Escribe el objeto receptor en una cadena.
 @param f Objeto receptor.
 @param cadena Cadena que recibe la expresión de f.
 Debe tener suficiente espacio.
 Es MODIFICADO.
 @return Cadena escrita.
 @doc
 Sobre 'cadena' se escribe una representación en formato 'dd/mm/aaaa'
 del objeto f. Devuelve la dirección de la cadena escrita.
 */
  char * Escribe (Fecha f, char * cadena);

/**
 Cambia f por la fecha siguiente a la que representa.
 @param f Objeto receptor. Es MODIFICADO.
 */
  void Siguiente(Fecha f, Fecha &g);

/**
 @brief Cambia f por la fecha anterior a la que representa.
 @param f Objeto receptor. Es MODIFICADO.
 */
  void Anterior(Fecha & f);
```

TDA Fecha

```
/**
 @brief Decide si f1 es anterior a f2.
 @param f1
 @param f2 Fechas que se comparan.
 @return
 true, si f1 es una fecha estrictamente anterior a f2.
 false, en otro caso.
 */
  bool menor(Fecha f1, Fecha f2);

/**
@brief Decide si f1 es anterior o igual que f2.
@param f1
@param f2 Fechas que se comparan.
@return
true, si f1 es una fecha anterior o igual a f2.
false, en otro caso.
*/
 bool menor_o_igual(Fecha f1, Fecha f2);
```

TDA Fecha. Ejemplo de uso

```
#include <iostream>
#include "fecha.h"
using namespace std;
int main() {
  Fecha f, g;
  int dia, mes, anio;
  char c1[100], c2[100];

  std::cout << "Introduzca el día del mes que es hoy: ";
  cin >> dia;
  cout << "Introduzca el mes en el que estamos: ";
  cin >> mes;
  cout << "Introduzca el año en el que estamos: ";
  cin >> anio;

  constructor(f, dia, mes, anio);
  Siguiente(f, g);
  if (menor(f, g)){
    Escribe(f, c1);
    Escribe(g, c2);
    cout << "El día " << c1 << " es anterior a " << c2 << endl;
  }
  return 0;
}
```

Implementación de un TDA

- Implica dos tareas:
  1. Diseñar la representación que se dará a los objetos
  2. Basándose en la representación, implementar cada operación
- Dentro de la implementación habrá dos tipos de datos:
  - **Tipo Abstracto**: definido en la especificación con operaciones y comportamiento definido
  - **Tipo *rep***: tipo a usar para representar los objetos del tipo abstracto y sobre el que implementar las operaciones

Representación del TDA

- Tipo abstracto: TDA Fecha
- Tipo *rep*:

```
struct Fecha {
  int dia;
  int mes;
  int anio;
};
```

El tipo *rep* no está definido unívocamente por el tipo abstracto.
P.ej.: typedef int Fecha[3];

TDA Polinomio ($a_n x^n + \ldots + a_1 x + a_0$)

Tipo *rep*: typedef float polinomio[n+1];

Implementación del TDA

En toda implementación hay dos elementos importantes:

- **Función de Abstracción**: conecta el tipo abstracto y el tipo *rep*
- **Invariante de representación**: condiciones que caracterizan los objetos del tipo *rep* que representan objetos abstractos válidos

Siempre existen, aunque habitualmente no se es consciente de su existencia

Función de abstracción

- Define el significado de un objeto *rep* de cara a representar un objeto abstracto. Establece una relación formal entre un objeto *rep* y un objeto abstracto

$$f_A : rep \longrightarrow A$$

Es una aplicación sobreyectiva

Ejemplos:

- TDA Racional $\{\mathbf{num}, \mathbf{den}\} \longrightarrow \dfrac{\mathbf{num}}{\mathbf{den}}$
- TDA Fecha $\mathbf{dia}, \mathbf{mes}, \mathbf{anio} \longrightarrow \mathbf{dia/mes/anio}$
- TDA Polinomio $r[0..n] \longrightarrow r[0] + r[1]x + \cdots + r[n]x^n$

Invariante de representación (IR)

- Invariante de representación: expresión lógica que indica si un objeto del tipo *rep* es un objeto del tipo abstracto o no

Ejemplos:

- TDA Racional: dado el objeto rep r={num,den}, debe cumplir den≠0
- TDA Fecha: dado el objeto rep f={dia,mes,anio} debe cumplir
  - 1 ≤ dia ≤ 31
  - 1 ≤ mes ≤ 12
  - Si mes es 4, 6, 9 u 11, entonces dia ≤30
  - Si mes es 2 y bisiesto(anio), entonces dia≤29
  - Si mes es 2 y !bisiesto(anio), entonces dia≤28

Indicando la FA y el IR

- Tanto la Función de Abstracción como el Invariante de Representación **deben aparecer escritos** en la documentación

```
#include "racional.h"
/*
    ** Función de abstracción:
       ----------------------
        fA : tipo_rep  ----> Q
             {num, den} ----> q

    La estructura {num, den} representa el racional q = num/den.

    **  Invariante de Representación:

      ---------------------------
    Cualquier objeto del tipo_rep, {num, den}, debe cumplir:
      - den!=0
*/
```

Preservación del IR

- La conservación del Invariante de Representación es fundamental para todos los objetos modificados por las operaciones que los manipulan. Su conservación se puede establecer demostrando que:

  1. Los objetos creados por los constructores lo verifican
  2. Las operaciones que modifican los objetos los dejan en un estado que verifica el IR antes de finalizar

¡Ojo! Esto sólo se puede garantizar si hay ocultamiento de información

TDA = Colección de valores + Operaciones sobre ellos

→ ESPECIFICACIÓN + IMPLEMENTACIÓN

- Tipo: Definición → Tipo Abstracto
- Operaciones

- Tipo → Tipo *rep* → FA, IR
- Operaciones

TDA Racional

- Tipo abstracto

  Pareja de valores (num,den) que representa un número racional num/den, con den≠0

- Tipo *rep*

```
typedef int Racional[2];
```

```
typedef struct{ int numerador;
                int denominador;
              } Racional;
```

FA $\{\textbf{num}, \textbf{den}\} \longrightarrow \dfrac{\textbf{num}}{\textbf{den}}$

IR `r.denominador ≠ 0`

TDA Fecha

- Tipo abstracto: Representa fechas según el calendario gregoriano en el formato dd/mm/aaaa
- Tipo *rep*

```
typedef int Fecha[3];
```

```
typedef struct{ int dia;
                int mes;
                int anio;
              } Fecha;
```

FA r → r.dia/r.mes/r.anio

IR un objeto f={dia,mes,anio} debe cumplir

- 1 ≤ dia ≤ 31
- 1 ≤ mes ≤ 12
- Si mes es 4, 6, 9 u 11, entonces dia ≤ 30
- Si mes es 2 y bisiesto(anio), entonces dia ≤ 29
- Si mes es 2 y !bisiesto(anio), entonces dia ≤ 28
  donde bisiesto(i) =((i%4==0) && (i%100!=0)) || (i%400==0)

TDA Polinomio

- Tipo abstracto

  Sucesión de números reales $\{a_0, a_1, \ldots, a_n\}$ representando el polinomio $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ (grado n)

- Tipo *rep* p $\longrightarrow$ p.coef[0] + p.coef[1] * x + ⋯ + p.coef[p.grado] * $x^{p.grado}$

```
typedef struct{ float * coef;
                int MaxGrado;
                int grado;
              } Polinomio;
```

FA p $\longrightarrow$ p.coef[0] + p.coef[1] * x + ⋯ + p.coef[p.grado] * $x^{p.grado}$

IR Condición para que el objeto representado sea válido:

$\forall p \neq 0$, p.coef[p.grado] $\neq 0$

&&

p[i]=0, $\forall i$ / p.grado < i $\leq$ p.MaxGrado

TDA Conjunto

- Tipo abstracto

  Colección de elementos de un tipo determinado (TipoBase) que no pueden repetirse

- Tipo *rep*

```
typedef int TipoBase;
typedef struct{ TipoBase * elementos;
                int nelem;
              } Conjunto;
```

FA r → {r.elementos[i]/ 0 ≤ i < r.nelem}

IR

r.elementos[i] ≠ r.elementos[j], ∀i,j, 0 ≤ i,j < r.nelem

TDA: clases de operaciones

- **Constructores primitivos**: crean objetos del tipo de dato sin requerir un objeto del mismo tipo como entrada
- **Constructores**: crean objetos del tipo de dato a partir de otro objeto del tipo de dato que reciben como entrada
- **Modificadores**: modifican los objetos del tipo de dato
- **Observadores**: reciben como entrada objetos del tipo de dato y devuelven resultados de otros tipos
- **Destructores**: permiten eliminar un objeto del tipo de dato, recuperando los recursos consumidos para su representación

Abstracción en iteración

- Una de las aplicaciones más importantes de la abstracción por especificación es la definición de **abstracciones iterativas**

- El **iterador** facilita el acceso a los elementos de colecciones (estructuras de datos con varios elementos) ignorando los detalles de su implementación y de las operaciones propias de la colección

Abstracción en iteración

- Supongamos que queremos sumar una colección de enteros

```
int suma(int v[], int n){
  int s=0;
  for(int i=0; i<n; i++){
   s += v[i];
  }
  return s;
}
```

La colección se almacena en un vector

```
int suma(lista l){
  int n;
  if (lista_vacia(l))
    return 0;
  else
    n = primer_elemento(l);
    return n + suma (resto_elementos(l))
}
```

La colección se almacena en una lista

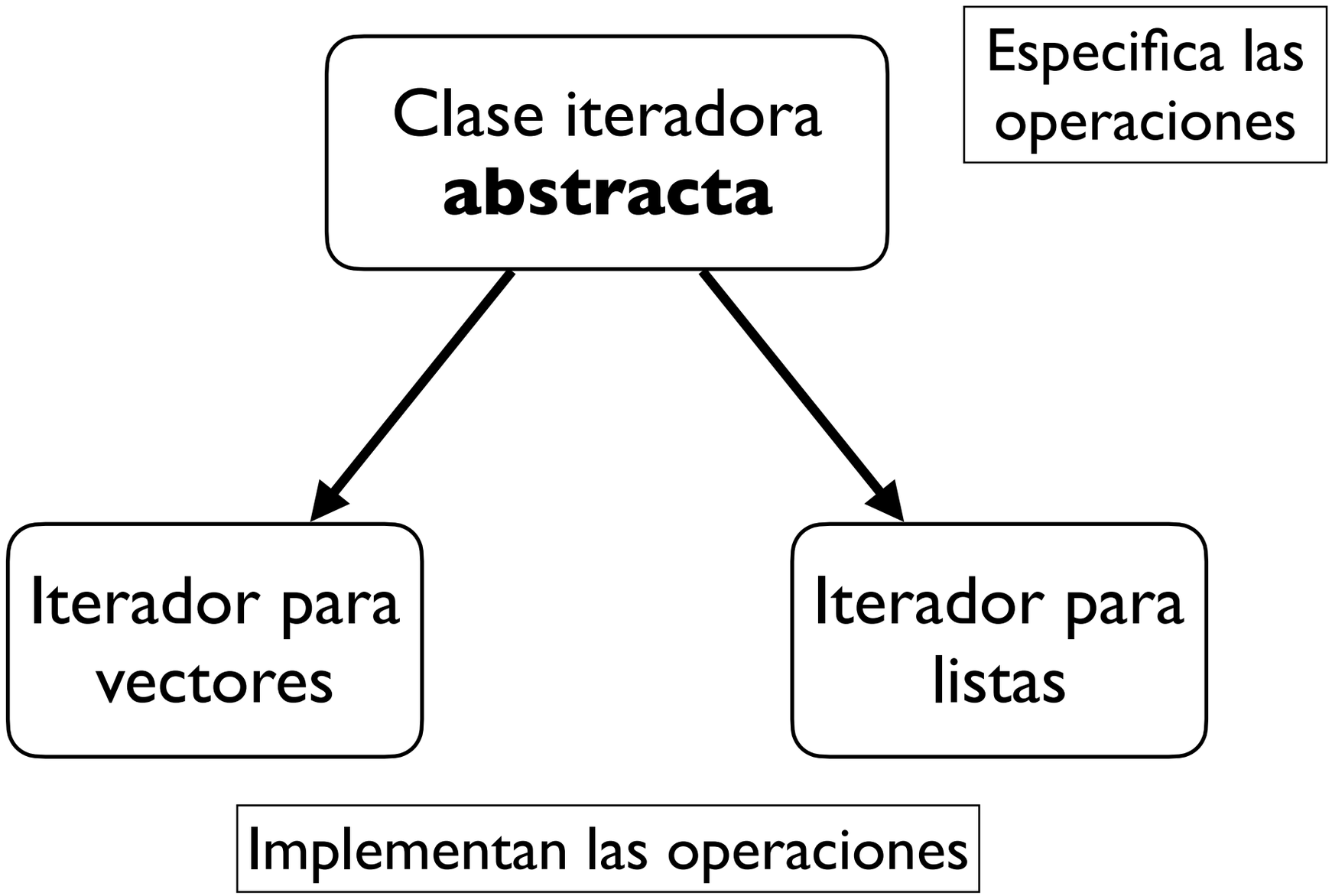
Abstracción en iteración

- Esquema general para el tratamiento de los elementos de una colección

```
obtener_primer_elemento();
mientras (hay_elementos()){
  procesar (elemento_actual());
  obtener_siguiente_elemento();
}
```

- Donde:
  - obtener_primer_elemento() inicializa el recorrido de la secuencia y, si no es vacía, deja disponible el primer elemento
  - hay_elementos() indica si hay elementos no tratados (recorridos) en la secuencia
  - elemento_actual() proporciona el elemento actual en la secuencia
  - obtener_siguiente_elemento() deja disponible el siguiente elemento de la secuencia

Abstracción en iteración



Abstracción en iteración

- Podemos reescribir ahora el algoritmo así:

```
int suma(iterador itr){
  int s=0;
  obtener_primer_elemento(itr);
  while(hay_elementos(itr)){
    s += elemento_actual(itr);
    obtener_siguiente_elemento(itr);
  }
  return s;
}
```

- Vemos que ahora no se accede directamente a las estructuras de datos.
- El algoritmo es independiente de las operaciones propias de la colección

TDAs en C++

- Los tipos predefinidos de C++ en realidad son TDAs.
- La idea es integrar en el lenguaje los nuevos tipos de datos que definamos de forma análoga a los predefinidos, de forma que códigos como éste sean válidos:

```
#include <iostream>
#include <polinomio.h>
using namespace std;
int main(){
  Polinomio p1, p2, resultado;
  cout << "Introduzca el primer polinomio" << endl;
  cin >> p1;
  cout << "Introduzca el segundo polinomio" << endl;
  cin >> p2;
  resultado = p1 + p2;
  cout << "La suma de los polinomios es " << resultado << endl;
  return 0;
}
```

TDAs en C++

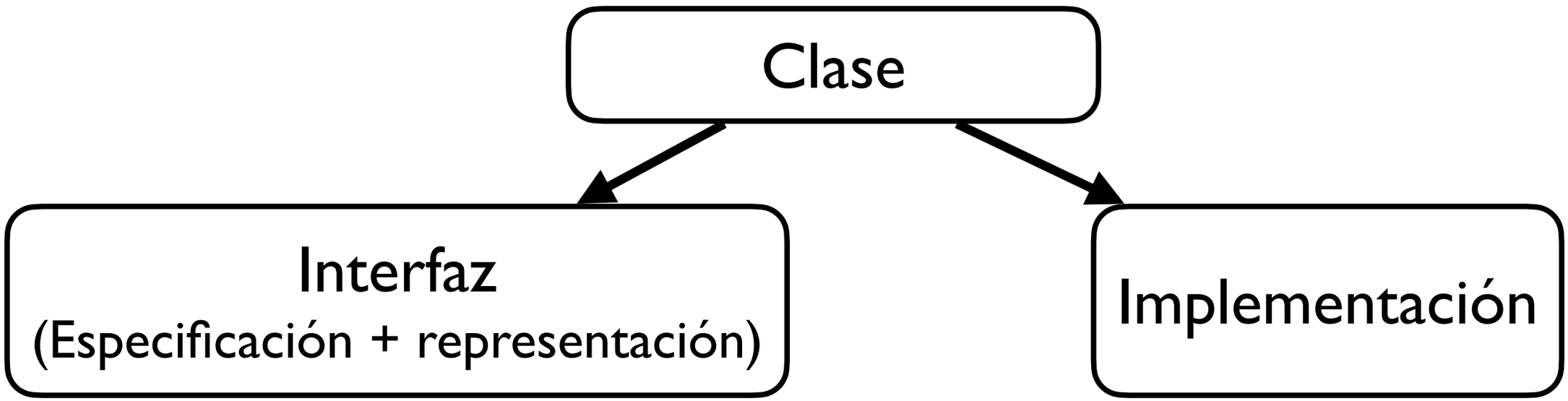
- Recordemos, TDA = datos + operaciones
- Podemos implementar la unión de datos y operaciones mediante estructuras (struct)

```
struct Polinomio{
  float * coef;        //Array de coeficientes
  int grado;           //grado actual del polinomio
  int MaxGrado;        //Máximo grado (espacio reservado)

  void AsignarCoeficiente(int i, float c);
  int Grado();
  float Coeficiente(int i)
  ..........
};
```

- Tiene inconvenientes, como la falta de ocultamiento. Por defecto, en las estructuras los miembros son públicos. También faltan otros mecanismos como la herencia...

TDAs en C++




```
class <nombre del tipo>{
  public:
    //sintaxis de las operaciones de la clase
    //(declaración de métodos)
  private:
    //área de datos (representación)
    //operaciones internas de la clase
};
```

- No podemos separar completamente la especificación de la implementación de la clase (la representación se incluye en la interfaz

TDAs en C++

- Una reflexión: ¿recuerdo lo estudiado en MP?
  - Control de acceso: private y public. Ámbito
  - Implementación de métodos miembro
  - Acceso a los datos miembro. El puntero this
  - Constructores. Destructor
  - Métodos inline
  - Métodos friend
  - Sobrecarga de operadores. Operadores como métodos miembro o métodos externos a la clase
    - El operador de asignación
    - Operadores de entrada/salida
    - Operadores unarios y binarios

Abstracción por generalización

- Avanzando en el nivel de abstracción, podemos crear una abstracción procedimental que se pueda usar para más de un tipo de dato.

```
int minimo (const int a, const int b){
  return(a < b ? a : b);
}
```

- Esta **función patrón** devuelve el mínimo de sus dos argumentos enteros. Pero su forma de calcular el mínimo es válida para cualquier tipo que soporte el operador <

```
template <class T>
T minimo (const T &a, const T &b){
  return(a < b ? a : b);
}
```

Abstracción por generalización

- Ejemplo: intercambio de valores

```
template <class T>
void intercambiar(T & a, T & b){
  T aux = a;
  a = b;
  b = aux;
}
```

- Otro ejemplo: ordenación de un vector

```
template <class T>
void ordenar_seleccion(T * vector, int tam){
  int pos_minimo;
  for(int i=0; i<tam; i++){
    pos_minimo = i;
    for(int j=i+1; j<tam; j++)
      if(vector[j] < vector[pos_minimo])
        pos_minimo = j;
    intercambiar(vector[i], vector[pos_minimo]);
  }
}
```

Parametrización/generalización de tipos

- La parametrización de un tipo de dato consiste en introducir un parámetro en su definición para poder usarlo con diferentes tipos base

  P.ej.: VectorDinamico T, donde T puede ser int, complejo, polinomio…

- Las especificaciones de VectorDinamico de diferentes tipos base son iguales, salvo por la naturaleza específica de los elementos que contendrá

- En lugar de escribir cada especificación, representación e implementación, se puede escribir una sola, incluyendo parámetros que representan tipos de datos

TDAs genéricos

- La especificación de un TDA genérico es igual que la de un TDA normal, salvo que se añaden requisitos adicionales para los tipos sobre los que se desee instanciar el TDA, normalmente la existencia de ciertas operaciones.

```
/**
   VectorDinamico::VectorDinamico, ~VectorDinamico, redimensionar,
   dimension, componente, asignar_componente

   Este TDA representa vectores de objetos de la clase T cuyo
   tamaño puede cambiar en tiempo de ejecución. Son mutables.

   Residen en memoria dinámica.

   Requisitos para la instanciación:
   La clase T debe tener definidas las siguientes operaciones:
    - Constructor por defecto
    - Constructor de copia
    - Operador de asignación
*/
```

Parametrización de tipos en C++

- C++ ofrece el mecanismo de los *templates* de clases para parametrizar tipos
- Declaración de un *template*:

```
template <parámetros> declaración
```

- Los parámetros de la declaración genérica pueden ser:
  - class identificador. Se instancia por un tipo de dato
  - tipo-de-dato identificador. Se instancia por una constante

Parametrización de tipos en C++

- Definición de funciones miembro:

```
template <class T>
class VectorDinamico{
  private:
    T * datos;
    ...
}

template <class T>
T VectorDinamico<T>::componente(int i) const{
  return datos[i]
}
```

Parametrización de tipos en C++

- Para usar un tipo genérico hay que instanciarlo, indicando los tipos base concretos que queremos utilizar.

```
VectorDinamico <int> vi;
VectorDinamico <float> vf;
```

- El compilador genera las definiciones de clases y los métodos correspondientes para cada instanciación que encuentre de la clase genérica. Por ello la definición completa de la clase genérica debe estar disponible (definición y métodos)

- Primera solución: el código se organiza como siempre (interfaz en el fichero .h e implementación en el .cpp), **pero la inclusión se hace al revés**. No se incluye el .h en el .cpp, sino el .cpp al final del .h

Parametrización de tipos en C++

Vector_Dinamico.h

```
#ifndef VectorDinamico_h
#define VectorDinamico_h

template <class T>
class Vector_Dinamico{
...
};

#include "Vector_Dinamico.tpp"
#endif // VectorDinamico_h
```

Vector_Dinamico.tpp

```
#include <cassert>

template <class T>
VectorDinamico<T>::VectorDinamico (int n){
...
}
...
```

Parametrización de tipos en C++

- Segunda solución: separar declaración y definición de la clase en dos ficheros (.h y .tpp), como siempre, y forzar la instanciación con un nuevo fichero

Vector_Dinamico_float.h

```
#include "Vector_Dinamico.h"
#include "Vector_Dinamico.tpp"


//Forzamos la instanciación de la clase Vector_Dinamico
//con el tipo base float, que se usa en el programa
template class Vector_Dinamico<float>;
```

compilando por separado main.c, Vector_Dinamico_float.h y enlazándolos

- Como podemos ver, cualquier solución vulnera el principio de ocultamiento de información 😫

Consultar *C++ Templates. The Complete Guide*