08 – Tipos Abstractos de Datos (TADs)

DOCENTE – FEDERICO VILENSKY

Abstracción

- La abstracción es una operación mental destinada a aislar conceptualmente una propiedad o función concreta de un objeto, y pensar qué es, ignorando otras propiedades del objeto en cuestión. – Wikipedia
- 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 – DLE-RAE
- En ciencia, se observan fenómenos y se crear modelos abstractos que nos ayudan a explicar estos fenómenos
  - Por ejemplo las leyes de Newton
    - Las masas son puntuales, no tienen volumen
    - Solo nos interesan las masas que interactúan, no nos interesa el sistema entero
    - Usamos resortes y cuerdas "ideales"

Abstracción – para la programación

- En vez de programar en binario tenemos lenguajes de programación
- Nosotros escribimos en el lenguaje, y eso se transforma en código de maquina
- Tratamos de no escribir en lenguaje de maquina porque el nivel de detalle con el que tenemos que hacer todo es tanto que se hace muy complejo
  - Suma en Assembly
- Como funcionan los lenguajes que estamos acostumbrados, lo que se hace es se vamos refinando el nivel de detalle que requiere el código de maquina de a pasos
  - Codigo en lenguaje de programación (C++, C#, Go, Rust)
  - Lenguajes intermedios (paso opcional)
  - Assembly
  - Binario (lenguaje de maquina)

Lenguajes de programación

- Un lenguaje de programación de “alto nivel (de abstracción)” es un lenguaje que puede ser refinado hasta llegar a código de maquina
  - El programa que hace el refinamiento es lo que se llama **compilador**
- Se habla muchas veces que un programa es de mas alto nivel que otro, esto es cuando las abstracciones que vienen en el lenguaje son mayores
  - No tiene manejo de memoria alguno
  - Tiene una biblioteca estándar mas grande
  - La sintaxis es mas simple
- También podemos hablar de distintos niveles de abstracción adentro de un mismo programa

Ejemplo – Área de triángulo

- Nos piden programar un programa que recibe los vértices de un triángulo, y luego calcula el área del mismo
- Primero vamos a introducir al tipo de dato que va a representar al punto

```
struct Punto { float x; float y; };
```

- Luego definimos las variables, y definimos los pasos para ejecutar el programa

```
int main()
{
  Punto p1, p2, p3;
  leer(p1, p2, p3);
  float area = calcularAreaTriangulo(p1,p2,p3);
  imprimir(area);
}
```

- Estamos especificándolo de forma **abstracta**, ya que las funciones no las tenemos escritas en código, pero el efecto que tienen las funciones lo podemos especificar sin problema
  - Estas instrucciones, se llaman **instrucciones abstractas**

Ejemplo – Área de triángulo

- Lo que estamos haciendo hasta ahora es particionar el problema que nos dieron o bien
  - Mediante instrucciones concretas (que forman parte del lenguaje)
  - Mediante otros programas que debemos resolver a continuación
- La idea es ir acotando el nivel de detalle que tenemos que tener en cuenta
  - Desde lo mas abstracto hasta lo mas concreto posible
- O sea, vamos utilizando abstracciones para ir particionando el problema en pedacitos mas chicos y mas simples de resolver
- Repetimos este proceso la cantidad de veces que nos sea necesario

Ejemplo – Área de triángulo

- Vamos por ejemplo a resolver la función calcularAreaTriangulo

```
float calcularAreaTriangulo(Punto p1, Punto p2, Punto p3)
{
  float base = calcularBase(p1,p2);
  float altura = calcularAltura(p1,p2,p3);
  return base*altura / 2.0;
}
```

- Y seguimos resolviendo los distintos problemas hasta terminar

Abstracción Procedural

- Al uso de instrucciones abstractas se le llama **abstracción procedural**, es la forma mas básica que tenemos de abstracción en la programación
- Es un método de programación, independiente del lenguaje en el que estemos trabajando
- Algunas de estas instrucciones abstractas las sustituimos por subprogramas (procedimientos, funciones, métodos)
- Otras las sustituimos por instrucciones concretas

Abstracción de Datos

- Además de los procedimientos, nos puede interesar la idea de abstraer los Tipos de Datos
- O sea, que vamos a usar Tipos Abstractos de Datos (TADs), e instrucciones abstractas para armar nuestros programas
- Qué es un TAD?
  - Son tipos de datos que, si bien pueden estar especificados con precisión, no es un tipo concreto del lenguaje
    - Esto significa que no tenemos constructores del lenguaje para poder utilizarlos
- Tipos NO abstractos en C++:
  - Básicos: int, float, char, bool, * (punteros), etc.
  - Estructurales: [](array), struct, tipos de estructuras dinámicas

Abstracción de Datos

- Cómo introducimos un TAD?
  - En C++, le vamos a dar un nombre, y una cantidad de operaciones que podemos aplicar a los elementos de ese tipo (funciones, procedimientos, métodos)
- Cómo refinamos un TAD? (para poder utilizarlo)
  - Lo definimos como un tipo concreto del lenguaje y refinamos las operaciones asociadas al tipo
- Vamos a estar hablando de
  - **Especificación** del TAD
    - Definición: nombre del tipo, y nombre + especificación de las operaciones
  - **Implementación** del TAD
    - Refinamiento de las operaciones
  - **Representación** del TAD
    - El tipo concreto que nos da una implementación en particular

Uso de TADs

- Vamos a armar una calculadora, usando un tipo abstracto de datos
- La idea es escribir un programa que simule una calculadora que nos deje operar con números racionales, que tengan la forma m/n, siendo m y n enteros (con n != 0)
  - Pensemos en una calculadora común y corriente, con un display que nos muestra el ultimo número calculado
  - Con ese número podemos operar (suma, resta, multiplicación, división) con otro numero, lo que nos da un nuevo resultado
  - Podemos también borrar el numero del display o reemplazarlo por otro número
  - El programa tiene que aceptar **comandos**, con **argumentos** que se corresponden a cada una de las operaciones
  - Nos va a contestar a los comando con el resultado, que va a simular al display

Uso de TADs

- Vamos a necesitar introducir variables para almacenar el último resultado que calculamos (lo que vamos a usar para simular el display)
- Qué tipo va a tener?
  - Tenemos que guardar números racionales, los cuales no son un tipo primitivo del lenguaje
  - Vamos entonces a hacer de cuenta que tenemos un tipo de datos **Racional**, así podemos seguir con el resto del programa, el cual vamos a implementar mas tarde
  - Las operaciones que va a tener asociado este tipo las vamos a ir definiendo a medida que vayamos viendo que necesitamos al hacer el resto del programa

Definición de TADs

- Tenemos que ver como se definen los TADs en C++
  - Se escribe un modulo, donde:
    - Definimos al tipo (le damos un nombre)
    - Definimos sus operaciones
  - En este modulo NO hacemos la implementación
  - O sea, estamos especificando el tipo, pero no implementándolo
- Para qué hacemos esto?
  - Nos permite trabajar con un mayor nivel de abstracción
    - Pateamos el problema para mas adelante/para otro
  - El programa principal no va a depender de la implementación que hagamos, sino que solo va a depender de la especificación
    - Podemos modificar la implementación, o usar otra completamente distinta, sin que se vea afectado el resto del programa

Uso de TADs – Declaración de vars

- Vamos entonces a incluir el modulo que definimos en nuestro programa para poder utilizarlo

```
#include “moduloRacional”
```

- Ahora podemos utilizar variables de tipo Racional en el programa principal
  - Precisamos dos
    1. Display
    2. Operando, que va a ser combinado con el display por medio de operaciones aritméticas

```
int main( )
{
  Racional display, operando;
  inicializar( );
  finalizar( );
}
```

Uso de TADs - Comandos

- Ahora precisamos los comandos que vamos a aceptar en el programa
- Los vamos a introducir mediante una enumeración

```
enum Comando {suma, resta, producto, division, borrar, ingreso, fin};
```

- Y vamos a agregar una variable de tipo Comando en el main

```
int main(){
  Racional display, operando;
  Comando c;

  inicializar();

  finalizar();
}
```

Uso de TADs – Refinar main

- A nivel de pseudocódigo, lo que queremos hacer es:
  - Inicializar
  - Hacer
    - LeerComando(c);
    - Si c es
      - suma: proceso suma
      - resta: proceso resta
      - producto: proceso producto
      - division: proceso división
      - borrar: proceso borrado
      - Ingreso: proceso ingreso
  - Mientras c != fin
  - Finalizar

Uso de TADs – Refinar procesos

- Al refinar los procesos nos vamos a ir dando cuenta cuales son las operaciones requeridas para el TAD Racional
- Proceso suma:
  - ```
    LeerRacional(operando);
    SumaRacional(operando, display);
    ```
- Leer racional lee el numero y lo guarda en operando
- SumaRacional suma el primer parámetro al segundo y lo guarda en el segundo
- Resta, producto, y división funcionan muy parecido
    ```

Uso de TADs – Refinar procesos

- Proceso borrado:
  - `BorrarRacional(visor);`
- Proceso ingreso
  - `LeerRacional(visor);`
- Antes de cerrar el loop, debermos imprimir el resultado, salvo que el comando sea el de finalización
  - `if(c!=fin) EscribirRacional(Visor);`

Operaciones del TAD Racional

- La inicialización lo único que hace (cuando prendemos nuestra calculadora) es borrar el visor, e indicarnos algún mensaje que necesitemos, como por ejemplo instrucciones
- La finalización lo único que hace es borrar las variables
- El resto de las operaciones que necesitamos ya las vimos
  - SumaRacional, RestaRacional, ProductoRacional, DivisionRacional, BorrarRacional, LeerRacional, EscribirRacional

Especificación de operaciones

- Todos los procedimientos debemos declararlos en el modulo de Racionales, y debemos especificar sus efectos

```
/* pre: q es un racional valido Q y r es un racional valido R o borrado
   pos: r = Q+R, o r queda borrado si ya lo estaba
*/
void SumaRacional(Racional q, Racional & r);
```

- Especificamos el resto de las operaciones de forma similar

Operaciones básicas de Racionales

- Además nos interesa tener ciertas operaciones básicas
  - Poder formar nuevos racionales a partir de un numerador y denominador enteros (Constructor)
  - Poder obtener el numerador y denominador de un racional dado (Getter/Setter)
  - Las que nos permiten saber si una variable racional fue "borrada" (Predicado)
- Estas operaciones son útiles para cualquier TAD, aunque en este momento capaz no las necesitamos, son super comunes, y mas vale tenerlas ya definidas
- Es como vamos a interactuar con la implementación concreta, ya que no conocemos como es por dentro la implementación

Operaciones básicas de Racionales

- Vamos a definirnos las operaciones

```
/*pre: q es un racional valido
  pos: retorna el numerador de q;
int Numerador(Racional q);

int Denominador(Racional q);

int EsValido(Racional q);

int EstaBorrado(Racional q);
```

Uso de TADs

- El uso de TADs, nos permite desarrollar por un lado el programa principal y por otro lado el TAD
  - Incluso podemos compilar al programa principal sin necesidad de tener la implementación del TAD, solo con la especificación
- Podemos entonces tener un equipo independiente implementando al TAD en paralelo, una vez definida la especificación

(Clases)

- En vez de struct, vamos a estar utilizando clases, funcionan de una forma muy similar
- Pero las clases nos permiten asociar un comportamiento a la estructura, además de la información
- Esto va a hacer que sea un poco mas clara la notación, por ejemplo en la suma, en vez de pasar (Racional q, Racional &r) y tener que acordarnos que el segundo es el que modificamos y no el primero
- En cambio lo que vamos a hacer es, para llamar a la función

```
r->Suma(q);
```

- Y para modificar el r dentro de la función, vamos a utilizar la palabra clave `this`

Implementación del TAD Racional

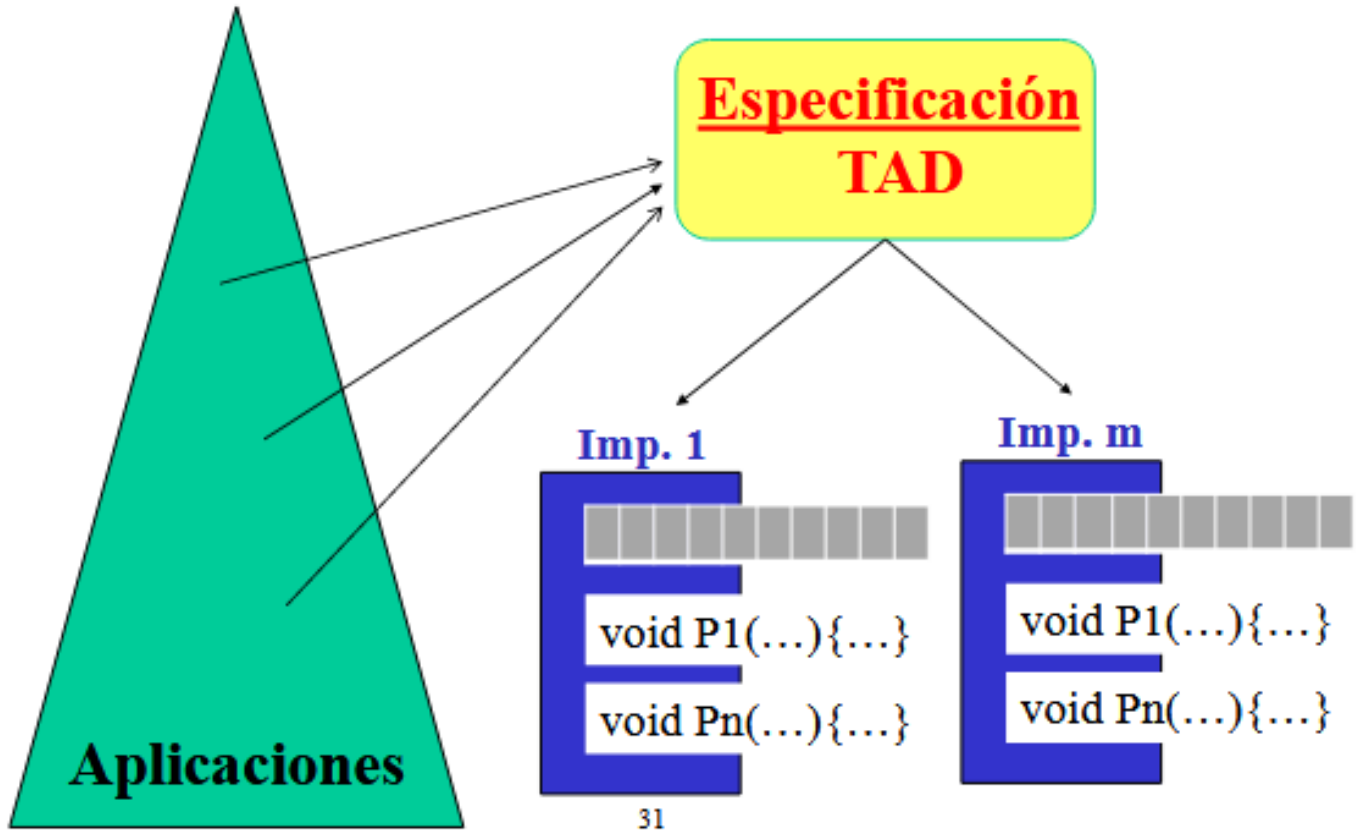
- Vamos a refinar al TAD Racional

```
class Racional{
  int num;
  int denom;

  int Numerador(){
    return this->num;
  }

  void Suma(Racional * q){
    Racional * sum = new Racional;
    sum.num = q->Numerador() * this->denom + this->num * q->Denominador();
    sum.denom = q->Denominador() * this->denom;
    this->Normalizar(sum);
  }
```

- Cumple con la especificación?
  - Si this esta borrado que pasa?
- El método Normalizar calcula la forma simplificada del racional dado como parámetro
  - Podemos tener mas métodos auxiliares en una implementación que no estén definidos en el TAD


Especificación
TAD
Aplicaciones
Imp. 1
void P1(…){…}
void Pn(…){…}
Imp. m
void P1(…){…}
void Pn(…){…}

Conclusiones

- Encontramos un método para ir especificando tipos abstractos durante el desarrollo de un programa
- Es una forma conveniente de ir abstrayendo y refinando
- Los programas quedan **desacoplados** de la implementación del TAD
  - Si modificamos al TAD, no vamos a tener que modificar nuestro programa
  - Solamente podemos modificar el estado del TAD mediante operaciones definidas, lo cual hace que sea difícil tener estados inconsistentes

Conclusiones

- Ventajas de uso de TADs
  - Modularidad – podemos elegir y cambiar la implementación (plug & play)
  - Ayudan a resolver sistemas no triviales
  - Separamos especificación de implementación
    - Programa más legible
    - Más fácil de mantener
    - Más fácil hacer testing (Robustez)
    - Más fácil reusar código
      - Menos chances de introducir bugs al no tener código repetido
    - Más extensible
    - Con el mismo programa, utilizando distintas representaciones del TAD podemos lograr distintas complejidades espaciales y temporales (modificamos los ordenes con cambiar la representación utilizada)

Conclusiones

- El uso de TADs nos permite prototipar de forma rápida
  - Podemos hacer una representación que sea poco eficiente, pero rápida de prototipar, y luego la cambiamos por otra mejor
- Algo que vamos a buscar en los TADs que vamos a ver es que sean lo mas genéricos posibles
  - Por ejemplo, no tener que redefinir la lista para cada tipo distinto

Observación

- Los ejemplos que vamos a estar haciendo en clase, los vamos a hacer en C++ con OOP, es decir clases, herencia y métodos asociados a objetos (instancias de la clase)
- Esto es porque es uno de los paradigmas mas comunes, y con lo que más se van a topar
- Sin embargo de mas esta decir que todo esto lo podemos llevar a cualquier otro paradigma
  - Procedural: C, Pascal, Go
  - Funcional: Haskell, Erlang, Elm, F#
  - Basado en Prototipos: JavaScript, TypeScript

Ejercicio

- Programar calculadora