UNIVERSIDAD COMPLUTENSE MADRID

Tema 4: Diseño e implementación de TADs

Miguel Gómez-Zamalloa
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Fac. Informática

Contexto

- Análisis de eficiencia
  1. Análisis de eficiencia de los algoritmos
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer*
  3. Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
  4. **Diseño e implementación de TADs**
  5. Tipos de datos lineales
  6. Tipos de datos arborescentes
  7. Diccionarios
  8. Aplicación de TADs

Contenidos

Tema 4: Diseño e implementación de TADs

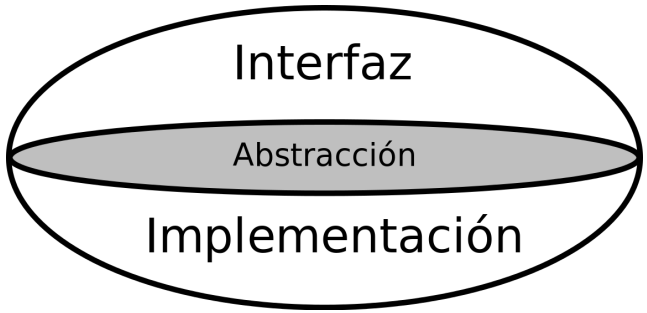
- Introducción
- Especificación de TADs
- Implementación de TADs en C++
- Ejemplo: El TAD Set

- Te plantean el siguiente problema: Dado un número x, se cogen sus dígitos y se suman sus cuadrados, para dar x_1. Se realiza la misma operación, para dar x_2, y así mucho rato hasta que ocurra una de las dos cosas siguientes:
  - se llega a 1, y se dice entonces que el número es "feliz"
  - nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es "infeliz"
- Ejemplos:
  - el 7 es feliz: $7 \to 49 \to 97 \to 130 \to 10 \to 1$
  - el 38 es infeliz:
    $38 \to 73 \to 58 \to 89 \to 145 \to 42 \to 20 \to 4 \to 16 \to 37 \to 58$

- La *abstracción* es un método de resolución de problemas
  - Los problemas tienden a ser cada vez más complejos
  - Es imposible considerar todos los detalles al mismo tiempo
- Una abstracción es un modelo simplificado de un problema donde:
  - Se consideran los aspectos de un determinado nivel
  - y se ignoran los restantes
- Ejemplos: construcción de edificio, receta albóndigas, internet, etc.
- Ventajas al razonar con abstracciones:
  - Se simplifica la resolución del problema
  - Soluciones claras $\Rightarrow$ Razonamientos de corrección
  - Soluciones (o fragmentos de ellas) reutilizables
  - Permite la división de tareas

La Abstracción como Metodología de Programación

- La programación se beneficia del uso de abstracciones
- Evolución de la programación:
  - Código máquina
  - Ensamblador
  - Lenguajes de alto nivel (funciones, módulos, TADS, POO, prog declarativa, etc)
  - Gestores de aplicaciones (hojas de cálculo, gestores de BBDD, generadores de interfaces, etc)
- En cada nivel tenemos:



Abstracción Funcional

- Una función o procedimiento es un conjunto de sentencias que realizan una determinada acción sobre unos datos
- Una vez implementada se puede usar como si fuese una operación del lenguaje (abstracción)
- Es importante disponer de una especificación (formal o informal)

Abstracción de Datos

- El nivel de abstracción de datos ha ido aumentado:
  - Bits y bytes
  - Tipos predefinidos: `int`, `char`, `float`, etc $\Rightarrow$ No debemos preocuparnos por su representación binaria
  - Tipos definidos por el programador. E.g. Tipo `Date`
  - TADS definidos por el programador: Entidades donde se definen conjuntamente el tipo de datos + las operaciones que los manipulan
- Observar la diferencia entre los tipos predefinidos y los definidos por el programador:
  - Un tipo predefinido es, en cierto modo, un TAD
    - Conjunto de valores + conjunto de operaciones
    - El compilador chequea que no se asignen valores erróneos
    - El programador solo necesita conocer el comportamiento de las operaciones
  - En los tipos definidos por el programador, al tener acceso a la representación interna es posible realizar operaciones incorrectas. E.g. `Tdate date; date.day = 31;`

- Un ejemplo clásico de TAD predefinido de C++ es la cadena de caracteres de su librería estándar: *std::string*
- Dada una cadena *s*, es posible saber su longitud (*s.length()* ó *s.size()*), concatenarle otras cadenas ($s \mathrel{+}= t$), sacar copias ($t = s$), o mostrarla por pantalla ($std::cout << s$), entre otras muchas operaciones
- Todo ello sin necesidad de saber cómo reserva la memoria necesaria ni cómo codifica sus caracteres

Tipos Abstractos de Datos

Un TAD se compone de:

1. *Interfaz* (*visible*) $\Rightarrow$ Especificación
   - Nombre
   - Dominio del tipo
   - Operaciones y su comportamiento
2. Implementación (*oculto*)
   - Representación interna del tipo
     - Tipos predefs
     - Tipos definidos
     - Otros TADS
   - Algoritmos $\Rightarrow$ Funciones y procedimientos

Ejemplo: El TAD `Date`

1) Especificación

- Nombre: `Date`
- Dominio: Las fechas desde el 1/1/1
- Operaciones + especificación informal:

```
Date newDate(int day,int month,int year);// throws domain_exception
  // Se devuelve un Date representando la fecha day/month/year
  // day/month/year deben representar una fecha válida,
  // sino se lanzaría una excepción

Date incr(Date date);
  // Se devuelve la fecha resultante al incrementar date en un día

int diff(Date date, Date date');
  // Se devuelve la distancia en número de días entre date y date'.

void print(Date date);
  // Imprime por consola la fecha en el formato DD/MM/YYYY
```

Ejemplo: El TAD `Date`

2) Implementación

- En general un TAD admite varias representaciones posibles
- Normalmente cada representación facilita unas operaciones y dificulta otras
- Alternativas para `Date`:
  1. Registro con tres `int` → `day`, `month` y `year`
  2. Un solo `int` → Número de días transcurridos desde el 1/1/1
- Con 1) se facilita `newDate` y `print`, y se dificulta `incr` y `diff`
- Con 2) se facilita `incr` y `diff`, y se dificulta `newDate` y `print`
- Otra alternativa sería llevar 1) + 2) ...

Soporte para TADS en Lenguajes de Programación

- Un lenguaje ofrece soporte para la implementación de TADs si incluye mecanismos para separar la interfaz de la implementación
  - Privacidad: La representación interna está oculta
  - Protección: El tipo solo puede usarse a través de sus operaciones
- **Convención**: En ocasiones se hace un "acuerdo entre caballeros" entre usuario e implementador para no tocar la parte privada
  - Ejemplo: Marca _ en identificadores privados, escribiendo, e.g., '*_dia*'
  - Esto ya no es necesario en C++ pero sí lo era en C
- Un lenguaje soporta el uso de TADs si permite elevar su uso al rango del de los tipos predefinidos del lenguaje
- El grado de soporte varía mucho $\Rightarrow$ Uso de módulos, POO
- En este curso usaremos la POO de C++
- Ejemplo: El TAD `Date`

Los TADs como Apoyo a la Programación Modular

- El desarrollo de programas grandes y complejos se realiza descomponiéndolos en unidades menores $\rightarrow$ *Divide y vencerás*
- Los TADs facilitan la división de tareas
- Un ejemplo:
  - Dado un vector $\mathtt{v}$ de N enteros y un número k, $0 \leq k \leq N$, determinar los k números mayores que aparecen en el vector
  - Usaremos una estructura auxiliar donde almacenamos los k números mayores encontrados hasta ahora. Dos opciones:
    1. Usar un vector (por ej) y incluir su gestión dentro del propio algoritmo
    2. Posponer la elección de la estructura y suponer que existe
  - Ventajas de 2):
    1. Descomponemos la tarea
    2. Reutilización del TAD resultante
    3. Claridad de la lógica del algoritmo

TADs y Estructuras de datos

- Un TAD está formado por una colección de valores y un conjunto de operaciones sobre dichos valores
- Una **estructura de datos** es una estrategia de almacenamiento en memoria de la información que se desea guardar
- Muchos TADs se implementan utilizando estructuras de datos
  - E.g., los TADs Pila y Cola pueden implementarse usando la estructura de datos de las listas enlazadas
- Algunos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación. E.g.: vectores, tablas, grafos, etc.

- Las especificaciones informales son, en general, imprecisas
  - E.g. Qué ocurre si hacemos `Date date; ... date += -5;`?
- Se podrían tratar de escribir todos los detalles textualmente ...
- Pero siempre habría ambigüedades e imprecisiones al no ser un lenguaje formal
- Aún así, las especificaciones informales son el escenario habitual
  - E.g. La *STL* de C++, la *API* de Java o la misma *JVM*

Especificaciones Formales

- Una especificación formal permite:
  1. Verificación formal $\Rightarrow$ Demostrar la corrección de los programas (automáticamente)
  2. Testing formal $\Rightarrow$ Generación automática de tests, testing automático, etc.
- Mas del 50 % de los costes en el desarrollo de software se dedica al testing!
- La verificación formal y el testing necesitan especificaciones formales
- Aspectos básicos de especificaciones formales:
  1. Dominio del TAD $\Rightarrow$ conjunto de posibles valores
  2. Invariante de la representación
  3. Relación de equivalencia
  4. Especificaciones formales pre/post de cada operación

Implementación de TADs en C++

- C++ es un lenguaje tremendamente flexible y potente (incluso demasiado ...)
- Hay más de una forma de hacer las cosas
- Hay que entender las ventajas y desventajas de cada alternativa
- Veremos las "mejores prácticas" reconocidas en la industria $\Rightarrow$ Referencia para hacer tus propias implementaciones

Clasificación de Operaciones

Constructor Crea una nueva instancia del tipo. En C++, un constructor se llama siempre como el tipo que construye. Se llaman automáticamente cuando se declara una nueva instancia del tipo.

Mutador Modifica la instancia actual del tipo. En C++, *no pueden* llevar el modificador *const* al final de su definición.

Observador No modifican la instancia actual del tipo. En C++, *deben* llevar el modificador *const* (aunque el compilador no genera errores si se omite).

Destructor Destruye una instancia del tipo, liberando cualquier recurso que se haya reservado en el momento de crearla (por ejemplo, cerrando ficheros, liberando memoria, o cerrando conexiones de red). Los destructores se invocan automáticamente cuando las instancias a las que se refieren salen de ámbito.

Mutadores vs. observadores

- *Se debe usar el modificador* ***const*** *siempre que sea posible* (i.e., para todas las operaciones observadoras)
- Hacer esta distinción tiene múltiples ventajas. Por ejemplo, una instancia de un tipo *inmutable* (sin mutadores) se puede compartir sin que haya riesgo alguno de que se modifique el original.
- Es frecuente, durante el diseño de un TAD, poder elegir entre suministrar una misma operación como mutadora o como observadora.
- E.g., en el TAD *Fecha*, podríamos elegir entre suministrar una operación *suma(int dias)* que modifique la fecha actual sumándola *delta* días (mutadora), o que devuelva una *nueva* fecha *delta* días en el futuro (observadora, y por tanto *const*).

Operaciones parciales

- Ciertas operaciones pueden ser erróneas; bien por definición o bien por limitaciones de la representación
- Ejemplos:
  - Acceso a la cima de una pila vacía
  - Acceso a un índice no válido en un vector
- Estas operaciones se denominan *parciales*
- Sus precondiciones para garantizar un comportamiento predecible deben especificarse debidamente
- Utilizaremos *excepciones* como mecanismo de tratamiento de errores

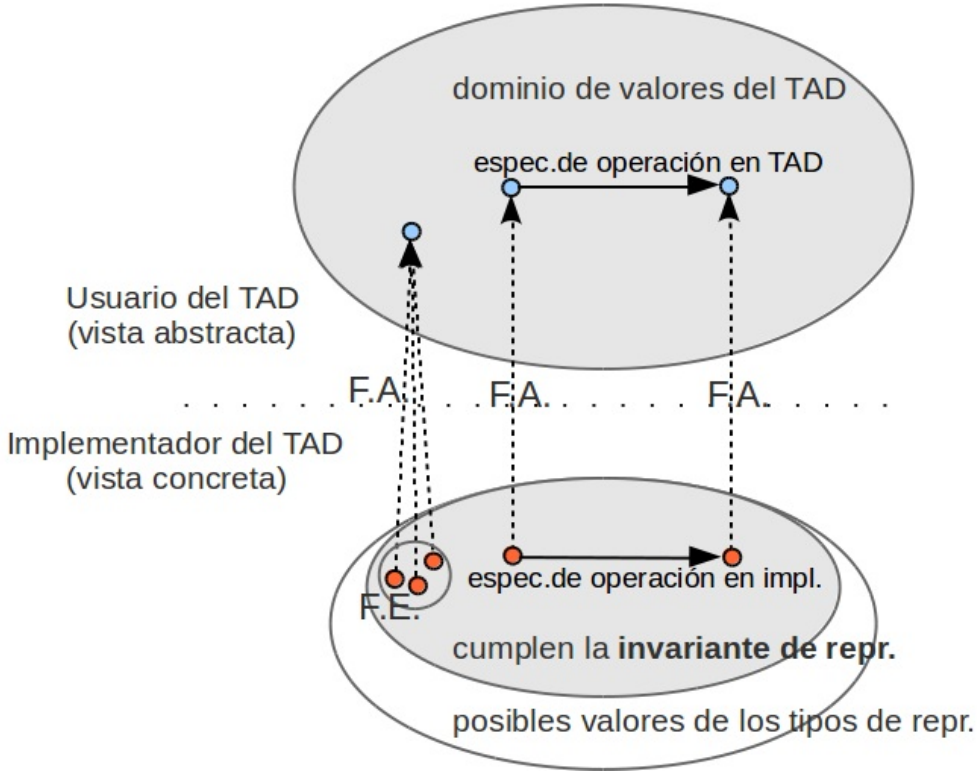
Invariante de la Representación

- Consiste en el *conjunto de condiciones que se tienen que cumplir para que una representación se considere como válida*
- Ejemplos:
  - TAD *Fecha* representado mediante tres enteros $\Rightarrow$ Los tres enteros forman una fecha válida
  - TAD *Rectangulo* representado mediante punto origen y par de enteros para ancho y alto $\Rightarrow 0 \leq \_ancho \wedge 0 \leq \_alto$
- Proporcionaremos el invariante de la representación mediante lógica de primer orden (como comentario)

Relación de Equivalencia

- La **relación de equivalencia** indica cuándo dos valores del tipo implementador representan el *mismo valor* del TAD
- En C++, una forma de implementarla es sobrecargando el operador '==='

```
1 class Rectangulo {
2     // permite ver equivalencia de rectangulos mediante r1 == r2
3     bool operator==(const Rectangulo& r) const {
4         return (esVacio() && r.esVacio())
5             || (_alto == r._alto && _ancho == r._ancho
6                 && _origen == r._origen);
7     };
8 };
```



- El invariante de representación delimita los valores posibles del tipo representante *válidos*
- La función de abstracción (F.A) los pone en correspondencia con términos del TAD
- Varios valores del tipo representante pueden describir al mismo término del TAD – Función de equivalencia (F.E)

TADs Genéricos

- Un TAD genérico es aquel en el que uno o más de los tipos que se usan se dejan sin identificar, permitiendo usar las mismas operaciones y estructuras con distintos tipos concretos
- Ejemplo: TAD `Conjunto`
  - No debería haber grandes diferencias entre almacenar enteros, punteros arbitrarios, o `Rectangulos`
- Hay varias formas de conseguir esta genericidad
  1. Plantillas
  2. Herencia
  3. Lenguajes dinámicos

1) Plantillas

- Permite declarar tipos como “de plantilla” (*templates*), que se resuelven en tiempo de compilación para producir todas las variantes concretas que se usan realmente
- Mantienen un tipado fuerte y transparente al programador

```
1 // requiere <stack>, un TAD generico 'pila' via plantillas
2     stack<bool> s;
3     s.push(true);       // s.push("ey") falla
4     bool b = s.top(); // b == true
```

2) Herencia

- Muy usado en Java y disponible en cualquier lenguaje con soporte OO.
- Requiere que todos los tipos concretos usados desciendan de un tipo base que implemente las operaciones básicas que se le van a pedir

```
1 // java.util.Stack (TAD generico 'pila' via herencia)
2     Stack s = new Stack(); // conteniene Object
3     s.push(Boolean.FALSE); // s.top() devuelve Object
4     s.push("ey");          // funciona sin problemas
5 // Java SE 5 en adelante: version semi-generica
6     Stack<Boolean> t = new Stack<Boolean>();
7     s.push(Boolean.FALSE); // s.top() devuelve Boolean
8     s.push("ey");          // error de compilacion
```

3) Lenguajes dinámicos

- JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución
- En estos casos, basta con que los objetos de los tipos introducidos soporten las operaciones requeridas en tiempo de ejecución (pero se pierde la comprobación de tipos en compilación)

Plantillas en C++

- En C++, se pueden definir TADs genéricos usando la sintaxis

```
1 template <class T1, ... class Tn> contexto
```

  y refiriéndose a los T_i igual que se haría con cualquier otro tipo a partir de este momento
- Generalmente se escogen mayúsculas que hacen referencia a su uso
- Por ejemplo, para un tipo cualquiera se usaría `T`, para un elemento `E`; etc.

Ejemplo de TAD genérico: Pair

```
1  // en el .h
2  template <class A, class B>
3  class Pair {
4      // una pareja inmutable generica
5      A _a; B _b;
6  public:
7      // Generador (un constructor sencillo)
8      Pair(A a, B b) { _a=a; _b=b; } // cuerpos en el .h
9      // Observadores
10     //    Pair(a, b).first() = a
11     A first() const { return _a; }
12     //    Pair(a, b).second() = b
13     B second() const { return _b; }
14 };
```

Plantillas en C++

```
1 // en el main.cpp
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 int main() {
6     Pair<int,string> p(4, "hola");
7     cout << p.first() << " " << p.second() << "\n";
8     return 0;
9 }
```

Plantillas en C++

- En C++ se pueden definir los cuerpos de funciones en el .h en lugar de en el prototipo (tal y como se hace en el ejemplo)
- En el caso de TADs genéricos, esto es **obligatorio** (en caso contrario se producirán errores de enlazado)
- Aunque para implementaciones más grandes es preferible usar esta versión alternativa, que deja el tipo más despejado, a costa de repetir la declaración de los tipos de la plantilla para cada contexto en el que se usa:

Plantillas en C++

```
1  // en el .h
2      ...
3      // Generador (un constructor sencillo)
4      Pair(A a, B b);
5      // Observadores
6      A first() const;
7      B second() const;
8  };
9  template <class A, class B>
10 Pair<A,B>::Pair(A a, B b) { _a = a; _b = b; }
11 template <class A, class B>
12 A Pair<A,B>::first() const { return _a; }
13 template <class A, class B>
14 B Pair<A,B>::second() const { return _b; }
```

Ejemplo: El TAD Set

```
1  template <class E>
2  class Set {
3      ... // Tipo representante
4  public:
5      // Constructor
6      Set();
7
8      // Inserta un elemento (mutadora)
9      // Podria dar error si no hay espacio
10     void add(const E &e);
11
12     // Elimina un elemento (mutadora)
13     void remove(const E &e);
14
15     // Si contiene e, devuelve 'true' (observadora)
16     bool contains(const E &e) const;
17 };
```