

PRÁCTICA 2: ABSTRACCIÓN

DOXYGEN

Gustavo Rivas Gervilla



UNIVERSIDAD
DE GRANADA

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

OBJETIVOS

- Asimilar los conceptos fundamentales de abstracción, aplicado al desarrollo de programas.
- **Documentar** un T.D.A.
- Practicar con el uso de **Doxygen**.
- Profundizar en:
 - **Especificación** del T.D.A.
 - **Representación** del T.D.A.
 - **Función de abstracción**.
 - **Invariante de representación**.

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

INTRODUCCIÓN

La **abstracción** estará presente en toda nuestra vida como ingenieros:

- Las matemáticas son pura abstracción.
- Esta presente en cualquier **algoritmo** por simple que sea.
- En el diseño de una **base de datos**.
- Al elaborar una aplicación **Android**.
- ...

INTRODUCCIÓN

La **abstracción** estará presente en toda nuestra vida como ingenieros:

- Las matemáticas son pura abstracción.
- Esta presente en cualquier **algoritmo** por simple que sea.
- En el diseño de una **base de datos**.
- Al elaborar una aplicación **Android**.
- ...



Figura 1: Cada estudiante de la UGR se representa en forma de datos en filas de varias tablas.

INTRODUCCIÓN

La **abstracción** estará presente en toda nuestra vida como ingenieros:

- Las matemáticas son pura abstracción.
- Esta presente en cualquier **algoritmo** por simple que sea.
- En el diseño de una **base de datos**.
- Al elaborar una aplicación **Android**.
- ...



Figura 2: El modelo-vista-controlador usado normalmente en las aplicaciones móviles es un claro ejemplo de abstracción. Nosotros sólo vemos botones.

INTRODUCCIÓN

Nosotros nos vamos a centrar en dos tipos de abstracciones:

- Abstracción **procedimental**.
- Abstracción de **datos**: **T.D.A.**

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

T.D.A.

Un T.D.A. es un conjunto de datos (**información**) y un conjunto de **operaciones** que se aplican sobre esos datos.

datos + operaciones

Esto tiene las mismas ventajas¹ que cualquier modularización que hagamos en un programa:

- Facilidad de uso.
- Desarrollo y mantenimiento más sencillo.
- Reusabilidad.
- Fiabilidad: es más fácil realizar pruebas sobre los módulos de forma independiente (*test unitarios*).

¹ Antonio Garrido Carrillo y Joaquín Fernández Valdivia. *Abstracción y estructuras de datos en C++*. Delta Publicaciones, 2006.

T.D.A.

Se conoce las operaciones que permiten manejar el TDA pero **no su implementación**, de ahí tenemos que:

T.D.A. \neq Estructura de Datos

La estructura de datos que elijamos será sólo para una implementación concreta, y una forma concreta de almacenar esa información.

T.D.A.

Por ejemplo, en una biblioteca podemos tener los libros ordenados en estantes por orden alfabético y etiquetados con códigos. O podemos tenerlos apilados en cajas de cartón, sin ningún tipo de sistema de etiquetado.

- El **T.D.A. es el mismo**:
 - Tenemos los mismos datos (la misma información): los libros de la biblioteca.
 - Tenemos las mismas operaciones a realizar sobre esos libros: pedir un libro, buscar un libro, devolverlo...
 - Pero según qué implementaciónelijamos estas operaciones serán más o menos **eficientes**.



T.D.A.

La separación entre **especificación** e implementación, cuando estamos trabajando en C++ la hacemos separándolas en dos archivos:

En el archivo .h encontraremos la especificación y la representación, mientras que en el .cpp tenemos la implementación. Con lo que la separación no es totalmente perfecta, ya que no separamos visión externa de interna. Pero sí que **ocultamos** la máxima información posible técnicamente.



Esto hace que la *re-compilación* sea más rápida, ya que si no cambiamos la interfaz el resto de módulos que hacen uso de ella no cambia.

T.D.A.

Cuando escogemos una representación concreta para almacenar la información, estamos escogiendo el **tipo *rep*** para nuestro T.D.A.:

- Cajas de cartón.
- Estanterías ordenadas por orden alfabético de título.
- Secciones por temática y orden alfabético.
- Sistema automatizado con interfaz web.

El **invariante de representación** son las condiciones que cumplen los elementos del tipo *rep* para representar un objeto real del T.D.A. Por ejemplo, para los libros tener un ISBN válido.

Finalmente, la **función de abstracción** nos indica cómo se relacionan o se *traducen* los elementos del tipo *rep* con los objetos del “*mundo real*”.

OBJETIVOS

INTRODUCCIÓN

T.D.A.

Ejemplos

DOXYGEN

LA PRÁCTICA

T.D.A

EJEMPLOS: T.D.A. RACIONAL

Para cada $r \in \mathbb{Q}$, tenemos que $r = \frac{a}{b}$ con $a, b \in \mathbb{Z}$ y $b \neq 0$.

Podemos elegir diversos tipos *rep*, y en base a eso, la función de abstracción y el invariante de representación serán ligeramente distintos:

tipo <i>rep</i>	abstracción	invariante
<code>int a, b;</code>	a/b	$b \neq 0$
<code>int v[2];</code>	$v[0]/v[1]$	$v[0] \neq 0$

T.D.A.

EJEMPLOS: ESTUDIANTE

Para un estudiante vamos a considerar los siguientes campos:

- Nombre.
- Fecha de nacimiento.
- DNI.

tipo <i>rep</i>	abstracción	invariante
<code>int</code> dia, mes, anio; <code>string</code> nombre, DNI;	El estudiante con ese DNI.	El DNI está en la BD. Y las restricciones usuales de fecha.
<code>int</code> fecha; <code>string</code> nombre, DNI;	El estudiante con ese DNI.	El DNI está en la BD. Y $\text{fecha} \geq 0$.

T.D.A.

EJEMPLOS: CARTA MAGIC

Para una carta Magic vamos a representar:

- Nombre.
- Coste de maná.
- Poder y resistencia.

tipo <i>rep</i>	abstracción	invariante
<pre>string nombre; int mana[6]; int poder, resistencia;</pre>	La carta con esas características.	$\text{mana}[i] \geq 0, i \in 0, \dots, 5.$ $\text{poder} \geq 0$ y $\text{resistencia} \geq 0.$
<pre>string nombre; unsigned int mana[6]; unsigned int poder, resistencia;</pre>	La carta con esas características.	

Observemos que en la segunda opción no incluimos ninguna condición en el invariante de representación. Nuestra representación **ya incorpora** esa restricción.

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

DOXYGEN

Documentar es importante para:

- El mantenimiento del código.
- Poder reutilizar nuestro código.
- Compartirlo con otras personas.
- Depurarlo si hemos realizado una buena **especificación**.

Existen diversas herramientas para documentar código:

- Sphinx lo podemos usar para documentar código Python.
- Javadoc se usa para documentar código Java, en particular para documentar un proyecto Android (Android Studio integra la herramienta para generar la documentación).

DOXYGEN

En esta práctica utilizaremos Doxygen una herramienta estándar para documentar código C++ (aunque también lo podemos usar con otros lenguajes como Java o Python).

Vamos a usar el código del T.D.A. Racional que podemos encontrar en el material asociado a la práctica a modo de ejemplo para ver cómo documentar código con Doxygen.

DOXYGEN

Los bloques de documentación Doxygen se engloban entre `/**` y `*/`, aunque hay otras opciones.

```
1  /**
2      * @file Racional.h
3      * @brief Fichero cabecera del TDA Racional
4      *
5      */
6
7  #ifndef __RACIONAL
8  #define __RACIONAL
9
10 #include <iostream>
11
12 using namespace std;
13
14 /**
15     * @brief T.D.A. Racional
16     *
17     * Una instancia @e c del tipo de datos abstracto @c Racional es un objeto
18     * del conjunto de los números racionales, compuestos por dos valores enteros
19     * que representan, respectivamente, numerador y denominador. Lo representamos
```

DOXYGEN

```
20      *
21      * num/den
22      *
23      * Un ejemplo de su uso:
24      * @include usoRacional.cpp
25      *
26      * @author
27      * @date Octubre 2017
28      */
```


DOXYGEN

A continuación vemos el tipo *rep* escogido para este T.D.A., así como cómo documentamos la función de abstracción y el invariante de representación asociado.

```
29  [
30  class Racional {
31
32  private:
33  /**
34   * @page repConjunto Rep del TDA Racional
35   *
36   * @section invConjunto Invariante de la [representacin
37   *
38   * El invariante es \e rep.den!=0
39   *
40   * @section faConjunto [Funcin de [abstraccin
41   *
42   * Un objeto [vlido @e rep del TDA Racional representa al valor
43   *
44   * (rep.num,rep.den)
45   *
46   */
```

DOXYGEN

```
48     int num; /**< numerador */  
49     int den; /**< denominador */
```

Observemos cómo se documentan los atributos de la clase con un comentarios después del atributo, añadiendo < al comentario. Esto está también recogido como tal en la documentación de Doxygen.

DOXYGEN

Dentro de cada bloque de documentación podemos emplear etiquetas especiales para marcar distintos elementos. Cada una de estas etiquetas pueden comenzar por @ o por \:

@brief	Añadimos una descripción general del método o clase.
@param	Marcamos cada uno de los parámetros del método.
@pre	Marcamos cada pre-condición del método.
@post	Marcamos cada post-condición del método.
@return	Describimos el valor devuelto por el método.

DOXYGEN

Hay otras muchas palabras clave que podemos utilizar para marcar distintos elementos de la documentación a generar:

<code>@author</code>	<code>@example</code>	<code>@authors</code>
<code>@bug</code>	<code>@date</code>	<code>@copyright</code>
<code>@deprecated</code>	<code>@exception</code>	<code>@sa</code>
<code>@warning</code>	<code>@code</code>	<code>@em</code>

DOXYGEN

```
50  ????
51  public:

53  /**
54   * @brief Constructor por defecto de la clase. Crea el numero racional 0/1
55   */
56   Racional();

58  /**
59   * @brief Constructor de la clase
60   * @param n numerador del racional a construir
61   * @param d denominador del racional a construir
62   * @return Crea el numero racional n/d
63   * @pre d debe ser distinto de cero
64   */
65   Racional(int n, int d);
66   // Racional(int n=0, int d=1): num(n),den(d) {}

68  /**
69   * @brief Constructor de copias de la clase
70   * @param c.num numerador del racional a construir
71   * @param c.den denominador del racional a construir
```

DOXYGEN

```
72     */
73     Racional (const Racional& c);

75 /**
76  * @brief Numerador
77  * @return Devuelve el numerador del racional
78  */
79     int numerador ();

81 /**
82  * @brief Denominador
83  * @return Devuelve el denominador del racional
84  */
85     int denominador();

87 /**
88  * @brief ㉔Asignacin de un racional
89  * @param n numerador del racional a asignar
90  * @param d denominador del racional a asignar
91  * @return Asigna al objeto ㉔implcito el numero racional n/d
92  * @pre d debe ser distinto de cero
93  */
94     void asignar(int n, int d);
```

DOXYGEN

```
96  /**
97   * @brief Compara dos racionales
98   * @param r racional a comparar
99   * @return Devuelve 0 si este objeto es igual a r,
100   *          <0 si este objeto es menor que r,
101   *          >0 si este objeto es mayor que r
102   */
103   bool comparar(Racional r);

105  /**
106   * @brief Imprime un racional en el formato "(n/d)";
107   */
108   void print();

110  /**
111   * @brief Suma dos racionales
112   * @param r racional a sumar con el objeto implicito
113   */
114   Racional operator+(const Racional & r);

116  /**
117   * @brief Multiplicacin de dos racionales
```

DOXYGEN

```
118     * @param r racional a multiplicar con el objeto implicito
119     */
120     Racional operator*(const Racional & r);

123 /**
124     * @brief Sobrecarga del operador +=
125     * @param r racional a sumar con el objeto  implicito
126     */
127     void operator+=(const Racional &r);

129 /**
130     * @brief Sobrecarga del operador ==
131     * @param r racional a comparar con el objeto  implicito
132     * @return Devuelve 0 si este objeto es igual a r
133     */
134     bool operator ==(const Racional &r);

136 /**
137     * @brief Salida de un racional a ostream
138     * @param os stream de salida
139     * @param r Racional a escribir
140     * @post Se obtiene en \a os la cadena (num/den) con \e num,den los valores
```


DOXYGEN

```
141     *   del numerador y denominador de \a r
142     */
143     friend ostream& operator<< (ostream& os, const Racional& r);

145 /**
146  * @brief Entrada de un Racional desde istream
147  * @param is stream de entrada
148  * @param r Racional que recibe el valor
149  * @retval El Racional 2 ledo en r
150  * @pre La entrada tiene el formato (num/den) con \e num,\e den los valores
151  *   del numerador y denominador
152  */
153     friend istream& operator>> (istream& is, Racional& r);

155 /**
156  * @brief Convierte un racional en irreducible;
157  */
158     Racional  simplifica();
```

DOXYGEN

Doxygen, además de proporcionar una sintaxis para documentar código, nos permite, a partir de esos comentarios, generar documentación en forma de un recurso externo al código.

Para ello no tenemos más que ejecutar la siguiente orden en el directorio donde tengamos nuestro código:

```
doxygen <config-file>
```

Donde <config-file> es el archivo de configuración de Doxygen que creemos.

DOXYGEN

Doxygen es muy potente y tiene un gran número de opciones de configuración:

OUTPUT_LANGUAGE	Seleccionar el idioma para la generación de la documentación.
INPUT	Especificar los ficheros documentados y directorios que contienen ficheros documentados.
INPUT_ENCODING	Para especificar la condificación de los ficheros a parsear (por defecto es UTF-8).
EXCLUDE	Ficheros o directorios a ignorar de los recogidos en INPUT.
EXPERIMENT_PATH	Especificar ficheros o directorios que contienen trozo de código que usamos como ejemplos.

DOXYGEN

EXTRACT_ALL	Si lo activamos Doxygen asumirá que todo está documentado y por tanto incluirá todo documentado aunque no lo esté.
EXTRACT_PRIVATE	Si lo activamos todos los miembros privados de una clase se incluirán en la documentación.
EXTRACT_STATIC	Lo análogo para los miembros estáticos.
SORT_MEMBER_DOCS	Si lo activamos (está activado por defecto) ordenará los métodos de una clase por orden alfabético.
OUTPUT_DIRECTORY	El camino (relativo o absoluto) del directorio donde alojar los ficheros de documentación generados.

DOXYGEN

- Podemos generar la documentación en distintos formatos: HTML (GENERATE_HTML), \LaTeX (GENERATE_LATEX) o incluso man page (GENERATE_MAN).
- Podemos incluir fórmulas escritas en código \LaTeX y tablas.
- Puede generar automáticamente diagramas de herencia entre clases (HAVE_DOT) o de llamadas entre funciones (CALL_GRAPH).
- Con el comando @ [LanguageId] podemos incluir documentación en distintos lenguajes en el mismo bloque. Luego con la opción OUTPUT_LANGUAGE filtraremos sólo el lenguaje deseado.

```
/** \~english This is English \~dutch Dit is Nederlands \~german Dies ist  
    Deutsch. \~ output for all languages.  
*/
```

DOXYGEN

Con tantas opciones, un archivo de configuración puede ser muy complejo, para generar una plantilla del fichero no tenemos más que ejecutar:

```
doxygen -g <config-file>
```

OBJETIVOS

INTRODUCCIÓN

T.D.A.

DOXYGEN

LA PRÁCTICA

LA PRÁCTICA

- Es voluntaria.
- No puntúa.
- Completar la clase con alguna función más.
 - Especificación en el .h con su correspondiente documentación Doxygen.
 - Implementación en el .cpp.
 - Y comprobar que funciona en el usoRacional.cpp.

LA PRÁCTICA

Observemos cómo se indica en el .cpp que estamos implementando los métodos de la clase Racional, a través del operador `::`. Evidentemente la cabecera de los métodos deberá coincidir con la que hemos puesto en la especificación.

```
1  #include<iostream>
2  #include "Racional.h"

4  using namespace std;

6  //Constructor por defecto
7  Racional::Racional(){
8      num = 0;
9      den = 1;
10 }

12 //Constructor
13 Racional::Racional(int n, int d ){
14     num = n;
15     den = d;
16 }
```

LA PRÁCTICA

```
18 //Constructor de copia
19 Racional::Racional (const Racional& c){
20     num = c.num;
21     den = c.den;
22 }

24 //Asignacin
25 void Racional::asignar(int n, int d){
26     num = n;
27     den = d;
28 }
```

LA PRÁCTICA

En cambio los operadores de E/S se declaran como funciones `friend` para poder tener acceso directo a la parte privada de la clase, pero no son parte de la clase en sí.

De hecho se podrían declarar, y sería una buena práctica, fuera de la clase, sin ser `friend`. Simplemente necesitaríamos métodos públicos que nos permitiesen acceder a esa parte privada (*getters* y *setters*).

```
60  ???
61  //Operador <<
62  ostream& operator<< (ostream & os, const Racional & r){
63      return os << '(' << r.num << ',' << r.den << ')';
64  }

66  //Operador >>
67  istream& operator>> (istream& is, Racional& r){
68      char caracter;
69      int numerador, denominador;
70      is >> caracter >> numerador >> caracter >> denominador >> caracter;
```

LA PRÁCTICA

```
71     r= Racional(numerador,denominador);  
72     return is;  
73 }
```

¿Alguna pregunta?
Buena semana.