Diseño Modular II

Josefina Sierra Santibáñez

18 de febrero de 2017

Orientación a Objetos en C++

En C++ los **módulos de datos** se representan mediante unidades denominadas **clases**.

Una clase define

- una estructura de **atributos**, que se denomina **la representación del tipo**, y
- unas operaciones denominadas **métodos**.

La definición de una **clase** contiene varios miembros. Estos pueden ser datos o funciones. Los miembros de una clase que representan datos se denominan **atributos** y las funciones de una clase **métodos**.

Dada la definición de una clase podemos definir **objetos** de la clase.

```
class Racional { ... }; // definicion de clase
Racional e1; // definicion de un objeto
```

Un **objeto** es una instancia de una clase. Cada objeto contiene **datos para los atributos** definidos en la clase y es **propietario de los métodos** de la clase, que actuan sobre dicho objeto.

Orientación a Objetos en C++

El objeto propietario de un método se denomina **el parámetro implícito** de dicho método (abreviado **p.i.**).

- El **p.i. no aparece explícitamente en la cabecera** de los métodos especificados en la definición de una clase.
- El **p.i.** funciona a todos los efectos como un **parámetro** pasado **por referencia** en C++.
- Se puede usar la palabra **const** para evitar que el **p.i.** sea modificado por un método.

Utilizaremos la definición de la clase **Racional** (contenida en el archivo **Racional_main.cc**) para ilustrar algunos conceptos básicos de la orientación a objetos en C++.

```
class Racional {
 ...
 public:
  int numerador() const; // numerador del p.i.
  void modifica_numerador(int n); // del p.i.
};
```

Orientación a Objetos en C++

En una **llamada a un método** de una clase, **el objeto sobre el que se aplica** (i.e. el propietario del método) **precede al nombre del método separado por un punto**.

```
#include <iostream>
using namespace std;

class Racional { ... };

int main() {
  Racional r(4,7);
  int d = r.denominador();
  r.modificar_denominador(d*2);
};
```

Los **métodos** que forman parte de un objeto pueden

- **crearlo**, e.g. `Racional(int n, int d);`
- **consultarlo**, e.g. `int denominador() const;`
- **modificarlo**, e.g. `void modifica_denominador(int d);`

Orientación a Objetos en C++

En la definición de una clase hay dos etiquetas **public** y **private**. Estas etiquetas determinan la visibilidad de los miembros de la clase.

Un **miembro público** de una clase **puede ser utilizado por cualquier función de otro programa o por un método de otra clase**.

El **acceso a los miembros privados** de una clase está **restringido a los métodos de dicha clase**.

Normalmente **los atributos se declaran privados**, para restringir el acceso a los detalles de la representación del tipo de datos. Esto se conoce como **ocultación de la información**.

Al utilizar atributos privados, es posible modificar la representación del tipo de la clase sin que esto afecte a otras partes del programa que utilizan la clase. Ya que **los objetos de la clase sólo pueden ser manipulados a través de los métodos públicos**.

Ejemplo 1: Definición de una Clase en C++

Un objeto de la clase `Racional` contiene **dos datos**, almacenados en los atributos **num** y **den**, que corresponden al numerador y al denominador del número racional representado.

La clase Racional tiene **dos métodos privados**:

- **normaliza** se utiliza para mantener la representación (i.e. **num** y **den**) de un objeto de la clase **Racional** normalizada
- **mcd** se ha declarado **static**, porque **no utiliza el p.i.**, simplemente calcula el máximo común divisor de sus argumentos

Tiene también los siguientes **métodos públicos**:

- dos métodos **constructores**, que permiten crear objetos nuevos
- dos métodos **consultores**, que permiten conocer el numerador y el denominador de un objeto de tipo Racional
- dos métodos **modificadores**, que permiten modificar el numerador o el denominador de un objeto de la clase Racional
- un método de **lectura** y otro de **escritura**

Orientación a Objetos en C++

En una clase todos los miembros son **privados por defecto**, de modo que la etiqueta **public no es opcional**.

Los métodos diseñados para **uso general se declaran públicos**.

- Un **constructor** es un método que describe cómo se construye una instancia de una clase.
- Un **consultor** es un método que examina el estado de un objeto, pero no lo modifica. Permiten obtener información sobre el p.i.
- Un **modificador** es un método que altera el estado del p.i.
- Los métodos de **lectura** se comunican con el canal de entrada (estándar) para leer objetos de la clase.
- Los métodos de **escritura** se comunican con el canal de salida (estándar) para escribir objetos de la clase.

Orientación a Objetos en C++: Constructores

Los constructores son métodos que **sirven para crear objetos nuevos**. El lenguaje C++ proporciona un tipo particular de constructores que **tienen el mismo nombre que la clase**.

- Estos métodos **se ejecutan automáticamente cuando se declara un objeto nuevo**. Por ejemplo, la declaración

  ```
  Racional r_1;
  ```

  realiza una llamada al constructor **Racional()** y construye un objeto `r_1` de tipo `Racional` que representa el número 0.
- Los **constructores** de una misma clase **se diferencian** entre sí **por su lista de parámetros**. La siguiente declaración construye un objeto `s` que representa el número racional $\frac{n}{d}$.

  ```
  Racional s(n,d);
  ```
- Si no se define ningún constructor para una clase, C++ **genera uno sin parámetros**, que inicializa los atributos utilizando el comportamiento por defecto del lenguaje de programación.
- Si en la definición de una clase figura un constructor con algún parámetro, se ha de definir explícitamente uno sin parámetros.

Orientación a Objetos en C++

En C++ se puede indicar si un método es un consultor o un modificador. **Por defecto, todos los métodos son modificadores**.

Para **especificar que un método es un consultor**, **se añade** la palabra **const detras** del paréntesis final **de la lista de parámetros**.

- Especificar si un método es un consultor **es una parte esencial del proceso de diseño**.
- Tiene importantes **consecuencias semánticas** (e.g. los métodos modificadores no se pueden aplicar a objetos constantes).
- No debe considerarse como un simple comentario.

Por ejemplo, en la definición de la clase **Racional** se especifica que

- el método **modifica_numerador** es un modificador, ya que su cabecera es `void modifica_numerador(int n);`
- el método **numerador** es un consultor, ya que su cabecera es `int numerador() const;`

Ejemplo 2: Separación de **Interfaz** e Implementación

La clase definida en el programa **Racional_main.cc** es sintácticamente correcta en C++. Podéis generar el archivo ejecutable **Racional_main.exe** mediante la siguiente instrucción

```
> g++ -o Rational_main.exe Rational_main.cc
```

Sin embargo en C++ se suele **separar** el **interfaz** de la clase de su **implementación**, escribiéndolos en archivos diferentes.

El **interfaz de la clase** contiene la declaración de la clase, de sus atributos y de sus métodos. Normalmente se guarda en un archivo con extensión **.hh** (e.g. **Racional.hh**).

Cualquier **programa que use una clase necesita conocer** el contenido de **su interfaz**, i.e. debe incluir (`#include`) dicho interfaz.

En nuestro ejemplo, el archivo de interfaz **Racional.hh** se incluye

- en el archivo de implementación, i.e. **Racional.cc**
- en el archivo que contiene la función `main`, i.e. **program.cc**

Ejemplo 2: Separación de **Interfaz** e Implementación

En programas complicados puede ocurrir que varios archivos incluyan el mismo archivo, y durante el proceso de compilación el archivo incluído **se intente copiar dos o más veces**.

Para evitar este problema, el archivo que contiene el interfaz **utiliza el preprocesador para definir un símbolo** cuando es procesado por primera vez. Esto es lo que hace la segunda línea del archivo `Racional.hh`: **#define RACIONAL_HH**

- El símbolo que se define en el archivo del interfaz de una clase (e.g. **RACIONAL_HH**) no debe aparecer en ningun otro archivo. Normalmente se construye a partir del nombre del archivo.
- La primera línea comprueba si el símbolo está definido: **#ifndef RACIONAL_HH**
- Si el símbolo no está definido, el archivo `Racional.hh` se procesa; en otro caso no se procesa, porque sabemos que ya ha sido procesado por el compilador anteriormente.

Ejemplo 2: Separación de **Interfaz** e Implementación

En una instrucción de tipo **#include <nombre>** el preprocesador busca el archivo **nombre** primero entre los componentes estándar de C++, y después en los directorios introducidos por la **opción -I** (que usaréis en las sesiones de laboratorio).

En una instrucción de tipo **#include "nombre.hh"** el preprocesador busca el archivo **nombre.hh** primero en el directorio local, y después en los componentes mencionados en el caso anterior.

Por este motivo usaremos

- **comillas** para incluir **módulos definidos por nosotros mismos**

```
#include "Racional.hh"
```

- **ángulos** para inclusiones de **elementos estándar**

```
#include <iostream>
```

Ejemplo 2: Separación de Interfaz e **Implementación**

La **implementación de una clase** es un archivo con extensión **.cc** (e.g. Racional.cc) que contiene las implementaciones de sus métodos.

- El archivo de implementación de una clase (Racional.cc) debe incluir el archivo de interfaz de la clase (#include "Racional.hh")
- En el archivo de implementación, **cada método debe indicar la clase a la que pertenece**. Si no, C++ asumiría que pertenece al ámbito global y se producirían errores de compilación.

  La sintáxis es **NombreClase::NombreMetodo**. El operador **::** se denomina **operador de resolución de alcance** o de ámbito.
- La **signatura de un método** en la **implementación** de una clase (e.g. `Racional.cc`) **debe coincidir excatamente** con su signatura en el **interfaz** de la clase (e.g. `Racional.hh`).

  El hecho de que un método sea consultor (i.e. el **const** final) o modificador forma parte de la signatura de dicho método.

Ejemplo 2: Compilación Separada

Para obtener el archivo ejecutable cuando la definición de la clase está separa en dos archivos (e.g. `Racional.hh` y `Racional.cc`), **se compila la clase** mediante la instrucción

```
> g++ -c Racional.cc
```

Esto genera **el archivo objeto de la clase** (`Racional.o`), que se enlaza con los archivos objeto de los programas que usan la clase.

Por ejemplo, para obtener el archivo ejecutable correspondiente al programa fuente **program.cc** se utilizan las siguientes instrucciones

```
> g++ -c program.cc
> g++ -o program.exe program.o Racional.o
```

La primera instrucción **compila** el **archivo fuente** `program.cc` y genera el **archivo objeto** (`program.o`). La segunda **enlaza** (en Inglés **link**) **los archivos objeto** `program.o` y `Racional.o`, para generar el **archivo ejecutable** `program.exe`.

Ejemplos de Diseño Modular

Presentamos algunos ejemplos de programas C++ construídos utilizando la metodología de **diseño modular**.

Estos programas utilizan dos **módulos de datos**, que corresponden a las siguientes **clases** C++:

- una clase denominada **Estudiant**, que permite representar y manipular información acerca de un estudiante
- una clase denominada **Cjt_estudiants**, que permite representar y manipular información acerca de un conjunto de estudiantes

Especificación de una Clase

Durante la **fase de especificación** de una clase

- **No se escribe nada en la parte** de la definición de la clase etiquetada con la palabra **private**, para asegurar la independencia entre los distintos módulos.
- Los **atributos** de la clase **no deben mencionarse en** la **especificación de la clase, ni** en las **especificaciones Pre/Post de sus métodos públicos**, por el mismo motivo
- Sí debe aparecer la **especificación Pre/Post de** cada uno de los **métodos públicos de la clase**.
- El resultado de la especificación es un **documento distinto del interfaz** de la clase, no es un archivo C++.
- La **especificación de una clase** debe contener la **información necesaria para que un "programador" pueda usar la clase**.

Especificación de la clase Estudiant: Constructores

La especificación de la clase **Estudiant** incluye dos constructores.

```
Estudiant();
/* Pre: cert */
/* Post: el resultat es un estudiant amb DNI=0
   i sense nota */
Estudiant(int d);
/* Pre: d >= 0 */
/* Post: el resultat es un estudiant amb DNI=d
   i sense nota */
```

- **Estudiant()** crea un estudiante con DNI igual a cero y sin nota. Se invoca al declarar una variable `est_1` del modo siguiente

  ```
  Estudiant est_1;
  ```

- **Estudiant(int d)** crea un estudiante con DNI igual a `d` y sin nota. Se invoca al declarar una variable `est_2` del modo siguiente

  ```
  Estudiant est_2(12345678);
  ```

Métodos Consultores de la clase Estudiant

La especificación de la clase **Estudiant** contiene cuatro consultores

```
int consultar_DNI() const;
/* Pre: cert  */
/* Post: el resultat es el DNI del p.i. */
double consultar_nota() const;
/* Pre: el p.i. te nota */
/* Post: el resultat es la nota del p.i. */
static double nota_maxima();
/* Pre: cert */
/* Post: el resultat es la nota maxima permitida
bool te_nota() const;
/* Pre: cert  */
/* Post: el resultat indica si el p.i. te nota */
```

Los consultores **te_nota** y **nota_maxima** son necesarios, porque hay operaciones que incluyen en su precondición

- el requisito de que un estudiante tenga nota o no, y
- que el valor de uno de sus argumentos sea una nota válida.

Métodos Modificadores de la clase Estudiant

La especificación de la clase contiene dos métodos modificadores

```
void afegir_nota(double n);
/* Pre: el p.i. no te nota, 0<=n<=nota_maxima() *
/* Post: la nota del p.i. passa a ser n */

void modificar_nota(double n);
/* Pre: el p.i. te nota, 0<=n<=nota_maxima() */
/* Post: la nota del p.i. passa a ser n */
```

Métodos de Lectura y de Escritura de la clase Estudiant

La clase también incluye un método de lectura y otro de escritura.

```
void llegir();
/* Pre: hi ha preparats al canal estandard
   d'entrada un enter no negatiu i un double */
/* Post: el parametre implicit passa a tenir
   el DNI i nota llegits del canal d'entrada;
   si el double no pertany a [0..nota_maxima()],
   el p.i. es queda sense nota */

void escriure() const;
/* Pre: cert */
/* Post: s'han escrit el DNI y la nota del p.i.
     al canal estandard de sortida; si el p.i.
     no te nota escriu "NP" */
```

Durante la fase de especificación se supone que los datos que recibe un método de lectura tienen el formato correcto. Ya que en esta fase no se conocen los detalles de la representación del tipo de datos.

Resumen de Compilación Separada en Diseños Modulares

Los programas con los que trabajaremos contendrán **varios módulos que compilaremos** en general **de forma separada**.

La **implementación de cada módulo** estará dividida a su vez en **dos o más archivos**. Por ejemplo, la implementación de la clase **Estudiant** consta de los archivos **Estudiant.hh** y **Estudiant.cc**.

Concretamente compilaremos un programa distribuido en *n* módulos (`modulo_1, ..., modulo_n`) en **dos fases**.

1. Obtendremos el fichero objeto (con extensión .o) de cada módulo y del programa que contiene la función `main`.

```
> g++ -c modulo_1.cc
...
> g++ -c modulo_n.cc
> g++ -c main.cc
```

   Esta fase genera los archivos `modulo_1.o,..., modulo_n.o, main.o`. Es posible que la compilación de cada módulo se realice en distintos momentos o por programadores diferentes.

Resumen de Compilación Separada en Diseños Modulares

2. Se enlazan los ficheros objeto de los distintos módulos para construir el programa ejecutable (con extensión .exe).

   `> g++ -o main.exe main.o modulo_1.o ... modulo_n.o`

   Esta fase genera el archivo **main.exe** enlazando los archivos `main.o`, `modulo_1.o`,..., `modulo_n.o`, además de todos los elementos estándar del lenguaje.

En los ejemplos de uso de las clases **Estudiant** y **Cjt_estudiants** veremos ejemplos de **programas que utilizan código contenido en archivos fuente a los que no tendremos acceso**. Por tanto, no los podremos compilar nosotros mismos.

Pero **tendremos acceso a**

- los **archivos objeto** (i.e. `Estudiant.o` y `Cjt_estudiants.o`)
- las **especificaciones** de las clases Estudiant y Cjt_estudiants

Ejemplos de Uso de la clase Estudiant

Para usar la clase `Estudiant` **basta** con disponer de **su especificación**, no es necesario conocer los detalles de la representación del tipo de datos, ni la implementación de sus métodos.

- Ejemplo 1: acción que modifica un objeto de tipo `Estudiant` redondeando su nota a la décima más próxima.
- Ejemplo 2: cálculo del porcentaje de estudiantes presentados a un examen a partir de un vector de objetos de tipo *Estudiant* con la información de cada estudiante.
- Ejemplo 3: búsqueda de un estudiante con un DNI dado en un vector de `Estudiant`.
- Ejercicio: redondear las notas de los estudiantes almacenados en un vector de `Estudiant`.

Diagramas Modulares

En un diseño modular se pueden **definir nuevos módulos** funcionales o de datos **a partir de** tipos de datos ya definidos. Estos últimos pueden ser **tipos de datos del lenguaje** de programación u **otros módulos** definidos por nosotros mismos o por otro programador.

Las **relaciones entre** los distintos **módulos de un programa** suelen representarse gráficamente mediante **diagramas modulares**.

Concretamente, en esta asignatura utilizaremos diagramas modulares para representar **un tipo particular de relación entre módulos a nivel de especificación**, que denominamos **uso** y definimos del modo siguiente: **un módulo A usa otro módulo B si en la especificación los métodos públicos del módulo A aparecen parámetros o resultados cuyo tipo es B**.

En asignaturas posteriores veréis **otros tipos de relaciones entre módulos** como *herencia/generalización* o *asociación/composición*.

Diagramas Modulares

En un **diagrama modular** representaremos la relación de uso entre dos módulos (i.e. A **usa** B) mediante una **flecha** $A \longrightarrow B$.

- El módulo correspondiente al programa principal se dibujará en la parte superior del diagrama.
- Debajo del módulo del programa principal se colocarán los módulos que son usados directamente por éste.
- En el siguiente nivel aparecerán los módulos que son usados por estos últimos, y así sucesivamente.
- Un diagrama modular no debe ser necesariamente un árbol, sino un **grafo dirigido acíclico**.

Especificación del módulo Cjt_estudiants

Cjt_estudiants es un nuevo tipo de datos que permite **representar y manipular** información acerca de un **conjunto de estudiantes**.

Su especificación se presenta en las páginas 20 a 22 de los apuntes de *Introducción al Diseño Modular*.

En la especificación de la clase **Cjt_estudiants se usa** la clase **Estudiant**. En C++ esto se indica utilizando la palabra **#include** seguida del nombre del interfaz de la clase **Estudiant.hh**.

Como ya hemos comentado anteriormente, para usar una clase no es preciso conocer su implementación.

Este ejemplo demuestra que **se puede especificar** la clase **Cjt_estudiants conociendo sólo la especificación** de la clase **Estudiant**.

Ampliación de una Clase

Supongamos que deseamos **añadir nuevas funcionalidades** a la clase **Cjt_estudiants** para **borrar un estudiante** y obtener el **estudiante con nota máxima** del conjunto.

Para hacer ampliaciones tenemos básicamente **tres opciones**.

1. **Modificar la clase añadiendo nuevos métodos**

   Implica **modificar la especificación** de la clase (ver páginas 22 y 23 de los apuntes de *Introducción al Diseño Modular*), y por tanto la forma en que es usada por otros módulos. Lo cual **afecta a la independencia entre módulos**.

   Podemos hacerlo **si tenemos acceso a la implementación** de la clase (i.e. **Cjt_estudiants.hh** y **Cjt_estudiants.cc**). Porque **es preciso modificar** su implementación.

Modificación de la Clase Estudiant

```
class Cjt_estudiants {
...
// Modificadores
...
void esborrar_estudiant(int d);
/* Pre: existeix un estudiant al p.i. amb DNI=d *
/* Post: el p.i. conte els mateixos estudiants
que l'original menys l'estudiant amb DNI = d */

// Consultores
...
Estudiant estudiant_nota_max() const;
/*Pre:el p.i. conte almenys un estudiant amb nota
/* Post: el resultat es l'estudiant del p.i. amb
nota maxima; si en te mes d'un, es el de DNI mes
petit */
};
```

Ampliación de una Clase

2. **Definir nuevas operaciones fuera de la clase**. Estas operaciones se pueden especificar donde se usen o en **un módulo funcional** (un **enriquecimiento** del original).

   La desventaja es que las nuevas operaciones **no serán orientadas a objetos**. Serán funciones convencionales de C++.

   Para especificar un módulo funcional escribiremos las **especificaciones Pre/Post** de las **nuevas operaciones** en un **archivo con extension .hh** (e.g. **E_Cjt_estudiants.hh**).

   Para **separar la especificación del módulo funcional de su implementación**, escribiremos la implementación de la nuevas operaciones en un archivo **E_Cjt_estudiants.cc**, que no será visible por los módulos que usen el enriquecimiento.

   Definir un **módulo funcional** sólo tiene sentido si las nuevas operaciones son **suficientemente generales** como para **ser usadas** por **otros módulos** o en **diferentes problemas**.

Ampliación de una Clase

3. Definir **una nueva clase que herede** los atributos y métodos **de Cjt_estudiants, e incorpore los métodos nuevos**.

   La **herencia** permite definir nuevos tipos de datos que heredan los atributos y métodos de una clase existente.

   Es un **concepto fundamental de la programación orientada a objetos**, que se estudiará en cursos posteriores.

   La desventaja en este caso es que definiríamos un nuevo tipo de datos. Los **nuevos métodos serían propiedad** de los **objetos de la nueva clase**.

Sesión 2 de Laboratorio

Es muy importante ampliar la información de esta sesión de teoría leyendo el guión de la sesión 2 de laboratorio

http://www.cs.upc.edu/pro2/data/uploads/sesion2.pdf

y realizando todos los ejercicios propuestos en este guión. Algunos de estos ejercicios, pero no todos, se pueden resolver utilizando la aplicación **jutge** (www.jutge.org).

Asimismo es conveniente revisar el material de la carpeta de la **asignatura PRO2** en los ordenarores de la **FIB**. Se puede acceder a dicha carpeta desde los ordenarores de las aulas de laboratorio (utilizando la ruta **/assig/pro2**) y desde el Racó de la FIB (pestaña 'Serveis', apartado 'Accés al disc').

Concretamente las especificaciones de clases y programas de la subcarpeta correspondiente a la sesión 2 de **/assig/pro2** deben usarse para resolver los ejercicios del guión de la sesión 2 de laboratorio.