Sesión 4

Implementación de módulos en C++

4.1 La clase `Estudiante`

En una sesión anterior introdujimos la clase `Estudiante`, como traducción a C++ del módulo del mismo nombre. Ello nos permitió codificar algunos programas en dicho lenguaje de manera bastante automática, ya que cada elemento del módulo tenía a su correspondiente en la clase: las variables de tipo Estudiante pasaron a ser objetos de la nueva clase y cada operación del módulo (crear, modificar, consultar ...) daba lugar a un método asociado.

Lo que no mostramos, sin embargo, fue la traducción de la implementación de dicho módulo. El usuario no podía saber cuál era la representación elegida para la clase `Estudiante` ni el código de sus métodos. La única información de la que se disponía era la especificación del módulo, y el compromiso por parte del codificador de respetar ésta en la traducción a C++.

En esta sesión realizaremos esta traducción nosotros mismos. Mostraremos una posible solución y aplicaremos algunas modificaciones sobre ella para obtener otra. Una vez terminada esta fase, volveremos a compilar y linkar los programas de la sesión anterior con la clase y los métodos resultantes para comprobar que su funcionamiento no se altera.

Un programador siempre ha de considerar que un tipo de datos puede tener más de una representación correcta. Generalmente, se comienza proponiendo una representación sencilla, aunque quizás no óptima, y luego se buscan maneras de mejorarla. La única condición es que los cambios que se apliquen no afecten al comportamiento de las operaciones de cara al exterior, es decir, que la nueva versión siga cumpliendo las especificaciones del tipo. Si por cualquier motivo dicho comportamiento se ve alterado, deberemos documentarlo en el fichero `.doc` correspondiente.

4.1.1 Elementos públicos y privados; elementos visibles y ocultos

Hemos comentado en clase de teoría que la especificación de un módulo ha de ser independiente de su implementación. La especificación de un módulo no debe contener elementos que permitan al usuario de la misma saber cómo está implementada o, en su defecto, no debe permitir al usuario acceder a dichos elementos para consultarlos o modificarlos.

El implementador de una clase en C++ dispone de varios mecanismos para conseguir dicha

independencia. En primer lugar, habréis visto que en el fichero `Estudiante2.hpp` hay dos secciones: la declaración de *elementos públicos* y la declaración de *elementos privados*. En la primera normalmente se encuentran las cabeceras de las operaciones públicas de la clase y en la segunda la declaración de los campos de datos de la misma. Eso permite utilizar las primeras e impide usar los segundos fuera de la clase.

En el caso particular del fichero `Estudiante2.hpp`, se ha tomado adicionalmente la medida de *ocultar* los elementos privados de la clase, es decir, sus campos. Esto supone una medida adicional de refuerzo de la independencia de la especificación respecto a la implementación. Para lograrlo, se han usado herramientas de lenguaje C++ que no forman parte del contenido de esta asignatura, pero sirve como ejemplo de lo que se puede llegar a hacer. En las clases que vostros diseñaréis no os pediremos la ocultación de los campos, nos conformaremos con *campos privados visibles*, de los que veremos un ejemplo a continuación.

4.1.2 Ejemplo: una primera implementación completa

La primera versión completa que veremos de la clase `Estudiante` en C++ está alojada en los ficheros `Estudiante.hpp` y `Estudiante.cpp`. El primero se compone de las dos partes mencionadas: primero se incluyen los campos de datos (visibles, pero privados) y después aparecen las cabeceras de la operaciones de la misma (públicas) con los cambios dados por el uso del parámetro implícito.

La representación elegida para el tipo Estudiante era una **tupla** que incluía el DNI, la nota y un booleano que indicaba si el estudiante tenía nota o no. Al traducirlo a C++, hay que tener en cuenta que logramos el mismo efecto simplemente listando los campos y sus tipos en la parte privada del fichero `Estudiante.hpp`, sin necesidad de usar `structs` u otros constructores del lenguaje.

El código de las operaciones aparece en el fichero `Estudiante.cpp`. Notad que las situaciones no permitidas por las precondiciones se controlan explícitamente mediante *excepciones*. No os pediremos que lo hagáis así en vuestros programas (bastará con tener disponibles maneras de comprobar las precondiciones) pero aquí lo incluimos a modo de ejemplo. En cualquier caso, notad que usamos mensajes de excepciones a la medida de cada error.

Observad también que en las cabeceras de las operaciones del fichero `Estudiante.cpp` va insertada la declaración `Estudiante::`, que indica que estas operaciones son las que se han declarado como públicas en `Estudiante.hpp`. Toda otra operación sería considerada como privada.

Podéis compilar el programa `red1.cpp` de la sesión anterior con esta clase y comprobaréis que sigue funcionando. Para que se vea que no hay trampa, hemos sustituido el `#include "Estudiante2.hpp"` por `#include "Estudiante.hpp"` y deberéis linkar el `red1.o` con el `Estudiante.o` del directorio de la sesión en lugar del `Estudiante2.o` de la ruta `$OBJECTS_CPP`.

La secuencia de comandos queda así:

```
g++ -c red1.cpp -I$INCLUDES_CPP
g++ -o red1.exe red1.o Estudiante.o
```

4.1.3 Ejercicio: una implementación alternativa

Suprimid el campo `tieneNota`[1] de la representación del módulo y dejad solamente `DNI` y `nota`, pero de forma que se mantengan todas las operaciones. Ello obliga a modificar el criterio para saber si un alumno tiene nota o no. Utilizad un valor especial del campo `nota` cuya presencia denote que la nota no ha sido añadida aún (ha de ser un valor que no se confunda con una nota válida, por ejemplo el -1).

Modificad la clase `Estudiante` de forma que se incorpore este cambio pero su funcionamiento no se vea afectado en absoluto. Comprobad que `red1` sigue funcionando con esta nueva versión sin modificar su código. Es conveniente recompilarlo, ya que la nueva clase tiene un campo menos, aunque las cabeceras de las operaciones no hayan cambiado. En cualquier caso, es obligatorio volver a linkarlo, ya que el `Estudiante.o` sí ha cambiado.

4.1.4 Ejercicio: otra modificación que el usuario no ve

Modificad los métodos de lectura y escritura para que no usen operaciones de la clase, sino que accedan directamente a los campos de la representación. Obviamente, no se trata de copiar tal cual el código de dichas operaciones sino de aplicar solamente las instrucciones necesarias. Notad que, como no cambia el fichero `Estudiante.hpp`, el usuario de la clase no necesita recompilar sus programas para seguir usándola. Tened especial cuidado con las situaciones anómalas.

4.1.5 Ejercicio: cambios que afectan a las cabeceras

Veamos una manera distinta de traducir los métodos del módulo `Estudiante`, que da lugar a unas nuevas cabeceras. Suprimid la operación modificadora `crear_estudiante(dni)` y considerad que la inicialización de un estudiante con un DNI se realiza mediante una única operación, también creadora, `Estudiante(int dni)`. Su código sería:

```
Estudiante (int dni)
  {
    if (dni<0) throw ....
    DNI = dni;
    ......  // inicializaciones de los demás campos
  }
```

Una vez completada, esta función se emplea así

```
// ..... se supone que hay un dni ya obtenido

Estudiante e(dni);
```

[1] Hay que decir que nos hubiera gustado llamar `tiene_nota` a este campo pero, por alguna razón, C++ no permite que los campos de una clase y las operaciones de la misma compartan nombre.

Notad que este cambio obliga a que los programas usuarios de esta clase también deban modificarse, lo que es bastante relevante, ya que se supone que éstos se diseñan basándose solamente en la especificación de dicha clase. Por eso, este tipo de cambios deben realizarse con mucho cuidado y documentándolos adecuadamente.

Aplicad el mencionado cambio al método de redondear la nota de un estudiante, versión función.

Una particularidad de C++ es que si una clase no tiene ninguna creadora explícita, C++ le dota de una creadora por defecto (es este caso sería `Estudiante()`), pero en caso contrario la creadora por defecto no existe. Por ejemplo, si eliminamos `Estudiante()` el compilador detectará un error, porque no está definida explícitamente.

4.1.6 Revisión del problema "Simplificación de un vector de estudiantes"

Comprobad si toda la secuencia de cambios realizados afecta al programa de simplificación de un vector de estudiantes (ejercicio 2.1.8).

4.2 La clase `Poli`

También podemos realizar algunos experimentos con la clase `Poli`. En primer lugar, estudiad detenidamente la implementación que os presentamos. Sobre ella, realizad y probad los siguientes cambios:

1. Modificad la operación `suma_poli` para que cada posición sólo se visite una vez si es posible.

2. Modificad los métodos de lectura y escritura para que no usen operaciones de la clase, sino que accedan directamente a los campos de la representación. Obviamente, no se trata de copiar tal cual el código de dichas operaciones sino de aplicar solamente las instrucciones necesarias. Intentad que los coeficientes del polinomio sólo se visiten una vez en cada operación

3. Añadid la operación `deriv` a la clase. Encontrad situaciones en las que sea interesante sustituir ciertas llamadas a operaciones de la clase por el acceso a los campos de la misma. Por ejemplo, notad el ahorro que conlleva el sustituir la operación `modif_coef` por instrucciones a medida de la situación.

4. Añadid la operación `div_poli` a la clase (y también su auxiliar `mult_poli_mon`), reescribiendo su código para aprovechar que conocemos la representación de la misma, como en el apartado anterior. Emplead el dividendo como parámetro implícito. Al acabar, comprobad si `div_poli` destruye información (escribid los polinomios dato *después* de dividir en el programa principal) y, si es el caso, buscad maneras de evitarlo.

5. Comprobad que si usáis la nueva `div_poli`, la función del cálculo de las raíces enteras sigue siendo correcta sin modificar nada más. Repetid el proceso con `div_ruffini`.