Diseño Modular III

Josefina Sierra Santibáñez

25 de febrero de 2017

Implementación de Clases en C++

La implementación una clase requiere **elegir una representación para el tipo de datos**, i.e. una estructura de atributos definidos en términos de tipos existentes, e **implementar sus métodos**.

- **public** y **private** determinan respectivamente los miembros de una clase que se pueden usar fuera de la clase y los que no.
- Los atributos se declaran **privados**, para que no puedan ser manipulados directamente fuera de la clase.
- Para implementar los atributos se puede usar cualquier combinación de tipos del lenguaje o tipos definidos por nosotros.
- En el cuerpo de un método de una clase se puede acceder al contenido de los atributos del parámetro implícito directamente (i.e. utilizando sólo el nombre del atributo).
- También se puede acceder al contenido de los atributos de otros objetos de dicha clase (distintos del p.i.). Pero en este caso es preciso utilizar la notación `nombre_objeto.nombre_atributo`.

Implementación de Clases en C++

La implementación de una clase en C++ normalmente consta de dos archivos.

- El archivo **.hh**, denominado **interfaz**, debe contener las cabeceras de todos los métodos y la representación del tipo.
- El archivo **.cc**, denominado **implementación**, debe contener la implementación de todos los métodos.
- En los apuntes de *Introducción al Diseño Modular* (páginas 25 a 29) se describe la **implementación de la clase Estudiant**.

Representación del Tipo Estudiant

La representación del tipo en el archivo `Estudiant.hh` consta de:

- Tres **atributos**: **dni** entero, **nota** double, y **amb_nota** de tipo booleano para especificar si el estudiante tiene nota.
- **MAX_NOTA** es una constante que limita el rango de notas válidas. Se declara **static**, para que sea **compartida por todos los objetos de la clase** `Estudiant`.
- Las **operaciones static**, como `nota_maxima()`, **sólo tienen acceso a los atributos static** de la clase.
- La **invariante de la representación** es un **conjunto de propiedades que deben cumplir los valores de los atributos** para representar objetos válidos de la clase. La invariante **se debe de mantener en todo momento**, y se considera **implícita a las precondiciones y postcondiciones** de los métodos.
- La invariante de `Estudiant` sólo requiere que **dni** sea **un natural**, y que si el estudiante tiene **nota** ésta sea **válida**.

Implementación de la clase Estudiant

En el archivo de implementación (e.g. **Estudiant.cc**) 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 ámbito.

En **Estudiant.cc** la cabecera de cada método está precedida por **Estudiant::** Esto permite identificar cada método con su correspondiente cabecera en el archivo **Estudiant.hh**, dándole permiso para acceder a los atributos de los objetos de la clase.

- En la implementación de una clase se puede usar la palabra **this** para referirnos al **parámetro implícito**.
- Será necesario usar **this** cuando

  – exista en un bloque de código una variable con el mismo nombre que un atributo, o

  – necesitemos referirnos al parámetro implícito en su conjunto (e.g. se pase como argumento en una llamada a otro método).

Implementación de la clase Estudiant

- Si en un bloque de código existe una variable con el mismo nombre que un atributo de la clase es preciso usar **this** para distinguir a cuál de los dos nos referimos en cada momento.

```
void Estudiant::afegir_nota(double nota) {
/*Pre: el p.i. no tiene nota y
        0 <= "nota" <= nota_maxima() */
/*Post: la nota del p.i. pasa a ser "nota"*/
    this->nota = nota;
    amb_nota = true;
}
```

- Si no hay problemas de ambiguedad, se utiliza el nombre del atributo (`amb_nota`) para referirnos al valor del atributo del p.i.
- Los nombres de los atributos no deben mencionarse en las especificaciones Pre/Post de los métodos públicos.

Implementación de la clase Cjt_Estudiants

En primer lugar comentamos la **representación del tipo**, contenida en el archivo **Cjt_Estudiants.hh**

- La constante **MAX_NEST** se introduce para limitar la capacidad de los conjuntos. Este atributo se declara **static** para que sea compartido por todos los objetos de la clase `Cjt_Estudiants`.
- Los estudiantes del conjunto se almacenan en el atributo **vest**, un vector de objetos de tipo `Estudiant`.
- El atributo **nest** indica el número de estudiantes que tiene el conjunto. Los estudiantes del conjunto están almacenados en la parte del vector correspondiente a los índices `[0 ...nest-1]`.
- La invariante establece que los estudiantes deben estar **ordenados crecientemente por DNI**.
- Esta decisión de implementación permite usar el método de **búsqueda dicotómica** en operaciones de consulta y modificación. Sin embargo requiere mantener el orden en inserciones y en la operación de lectura.

Implementación de la clase Cjt_Estudiants

Comentamos los **métodos privados** declarados en **Cjt_Estudiants.hh**

- Se introducen **métodos privados**, `ordenar_cjt_estudiants` y `cerca_dicot`, que se utilizan como operaciones auxiliares en distintos métodos públicos.
- En la **especificación de un método privado** pueden aparecer **elementos privados** de la clase como sus atributos.
- Los **métodos privados** no se suelen detectar durante la fase de especificación. Se añaden durante la fase de **implementación**.
- El método `cerca_dicot` se declara **static**, porque **no utiliza el parámetro implícito**. Ya que le pasamos el vector de estudiantes como argumento.

Implementación de la clase Cjt_Estudiants

En las páginas 31 a 34 de los apuntes de *Introducción al Diseño Modular* se presenta el contenido del archivo **Cjt_Estudiants.cc**, en el que se implementan todos los métodos.

- Recordad que para ordenar los n primeros elementos de un vector v, suponiendo que $n \leq v.size()$, se puede usar

```
sort(vest.begin(), vest.begin()+n, comp);
```

- En las páginas 34 y 35 de los apuntes se presenta la implementación de la segunda opción para la ampliación de la clase `Cjt_estudiants`, i.e. el módulo funcional `E_Cjt_estudiants` que supone un enriquecimiento de la clase.

  La implementación de `esborrar_estudiant` es ineficiente, porque no podemos acceder a los atributos de la clase.
- La primera opción consiste en modificar `Cjt_estudiants.hh` añadiendo las especificaciones Pre/Post de los nuevos métodos, y el archivo `Cjt_estudiants.cc` con sus implementaciones.

Ampliación de la clase Cjt_Estudiants

También se puede **modificar la representación del tipo**, para implementar de forma eficiente los nuevos métodos.

Concretamente se añade un nuevo atributo **imax** para recordar la posición del estudiante con nota máxima.

En la página 36 de los apuntes se presenta la implementación de los nuevos métodos, y se proponen los siguientes ejercicios que forman parte de la sesión 3 de laboratorio:

- Implementar el método **recalcular_posicio_imax**.
- Mejorar la implementación de **esborrar_estudiant** utilizando el método de **búsqueda dicotómica**.
- Aplicar todas las **modificaciones** necesarias a la implementación de los **métodos de Cjt_estudiants** para que **inicialicen y mantengan actualizado** el atributo **imax**.