Diseño Modular

Josefina Sierra Santibáñez

18 de febrero de 2017

Introducción al Diseño Modular ⇒ Orientación a Objetos

En esta asignatura utilizaremos la **orientación a objetos** como mecanismo de C++ para codificar **diseños modulares**, es decir, diseños de programas basados en el concepto de *módulo*.

1. Introduciremos conceptos asociados a la modularidad como **mecanismo general de estructuración de programas**.
2. Aplicaremos estos conceptos al **desarrollo de programas** C++.

Importancia del diseño modular en la evaluación de PRO2
La calificación global de la asignatura se calcula de acuerdo con la siguiente fórmula

$$0{,}10 \cdot N_{ControlLab} + 0{,}25 \cdot N_{Practica} + 0{,}15 \cdot N_{ControlPractica} + 0{,}25 \cdot N_{ParcialTeor1} + 0{,}25 \cdot N_{ParcialTeor2}$$

Los **contenidos** del capítulo de **Diseño Modular** se **evalúan en** el **Control Lab**, la **Práctica** y el **Control Práctica**. Intervienen, por tanto, en un **50 % de la evaluación de la asignatura**.

Necesidad de Controlar la Complejidad

El objetivo es producir programas **fiables, fáciles de entender, mantener, modificar** y **reutilizar**.

Estas propiedades afectan significativamente

- al **coste de diseño** de los programas, y
- a su **vida útil**.

En **programas pequeños** esto se puede conseguir utilizando unas cuantas acciones o funciones agrupadas en una unidad monolítica.

Pero **si el tamaño del programa crece considerablemente** la utilización de una estructura monolítica nos puede hacer perder todas estas propiedades.

Necesidad de Controlar la Complejidad

Problemas de la utilización de una estructura monolítica para codificar **programas de tamaño mediano o grande**.

- Modificar o mantener fiablemente una unidad monolítica de miles de líneas es sumamente complejo.
- Reutilizar partes de dicha unidad monolítica presenta muchas dificultades.
- Garantizar la fiabilidad de un programa de estas dimensiones es prácticamente imposible.

Necesidad de Controlar la Complejidad

La forma de evitar estos problemas en programas de tamaño medio es **dividir el programa en módulos**, que juntos realicen la misma tarea que el programa monolítico.

Requisitos de una buena descomposición modular.

- Los **módulos** deben ser **independientes**: *debe ser posible realizar cambios en un módulo sin que esto obligue a modificar los demás módulos*.
  La independencia entre los módulos facilita el mantenimiento de los programas, su modificación y el trabajo en equipo.
- Los **módulos** deben tener **una entidad propia**: deben realizar unas tareas determinadas.
  Esta propiedad es importante para construir programas fáciles de entender y módulos reutilizables en otros programas.
- Los módulos **han de interactuar con otros módulos de una forma simple y bien definida**.
  Esto facilita el trabajo en equipo y la reutilización de módulos.

Abstracción

En las descomposiciones modulares que veremos se utilizan técnicas de **abstracción funcional** y de **abstracción de datos**.

Aplicar **abstración** a un problema implica ignorar ciertos detalles para transformarlo en otro más sencillo y a la vez más general.

- **Abstracción mediante parametrización**: sustituir datos particulares por parámetros en acciones y funciones.
- **Abstracción mediante especificación Pre/Post:** permite conocer qué hace una función o acción sin necesidad de saber cómo lo hace. **Para utilizar una operación especificada con Pre/Post sólo es necesario conocer su especificación**: cabecera, precondición y postcondición.
- **Abstracción de datos:** definir un nuevo tipo de datos de manera que **pueda utilizarse conociendo únicamente su especificación** (no los detalles de su implementación) y que **las modificaciones a su implementación no afecten a su uso**.

Abstracción mediante Parametrización

Sustituir datos particulares por parámetros en acciones y funciones.

```
void base2(int n) {
    if (n < 2) cout << n;
    else {
        base2(n/2);
        cout << n%2;
    }
}
```

Sustituyendo la constante 2 por el parámetro b obtenemos

```
void base_b(int n, int b) {
    if (n < b) cout << n;
    else {
        base_b(n/b,b);
        cout << n%b;
    }
}
```

Abstracción mediante Especificación Pre/Post

Una **especificación Pre/Post** de una acción o función **consta de** los siguientes elementos:

- **cabecera** de la operación, indica el nombre de la operación, el tipo de su resultado y los tipos de sus parámetros;
- **precondición**, describe las propiedades que deben cumplir los argumentos que contienen datos;
- **postcondición**, especifica las propiedades que deben cumplir los resultados y los argumentos pasados por referencia no constante después de ejecutar la operación.

Una especificación Pre/Post debe interpretarse del modo siguiente: **si los datos de la operación satisfacen la precondición, entonces los resultados cumplirán la postcondición**; en otro caso, la operación podría generar errores de ejecución o almacenar valores absurdos en resultados y argumentos pasados por referencia no constante.

Abstracción mediante Especificación Pre/Post

Una especificación Pre/Post **permite conocer qué hace** una función o acción **sin necesidad de conocer cómo lo hace**. Esto significa que

- Para utilizar una operación especificada con Pre/Post sólo se necesita **conocer su cabecera**, qué deben cumplir los datos (**Pre**) y qué satisfarán los resultados (**Post**).
- Para utilizar dicha operación correctamente es preciso disponer además de los **medios** necesarios **para comprobar su precondición** antes de invocarla.
- Una consecuencia importante de lo anterior es que **cualquier modificación de la implementación** de una acción o función **que no afecte a su especificación Pre/Post** $\Rightarrow$ **no debe afectar a su uso**.

Abstracción mediante Especificación Pre/Post

```
bool is_prime(int n) {
/* Pre: n > 0 */
/* Post: Devuelve "true" si n es primo, y
   "false" en otro caso. */
   int div = 2;
   bool is_prime = n != 1;
   while (is_prime and div < n) {
      if (n%div == 0) is_prime = false;
      else div = div + 1;
   }
   return is_prime;
}
```

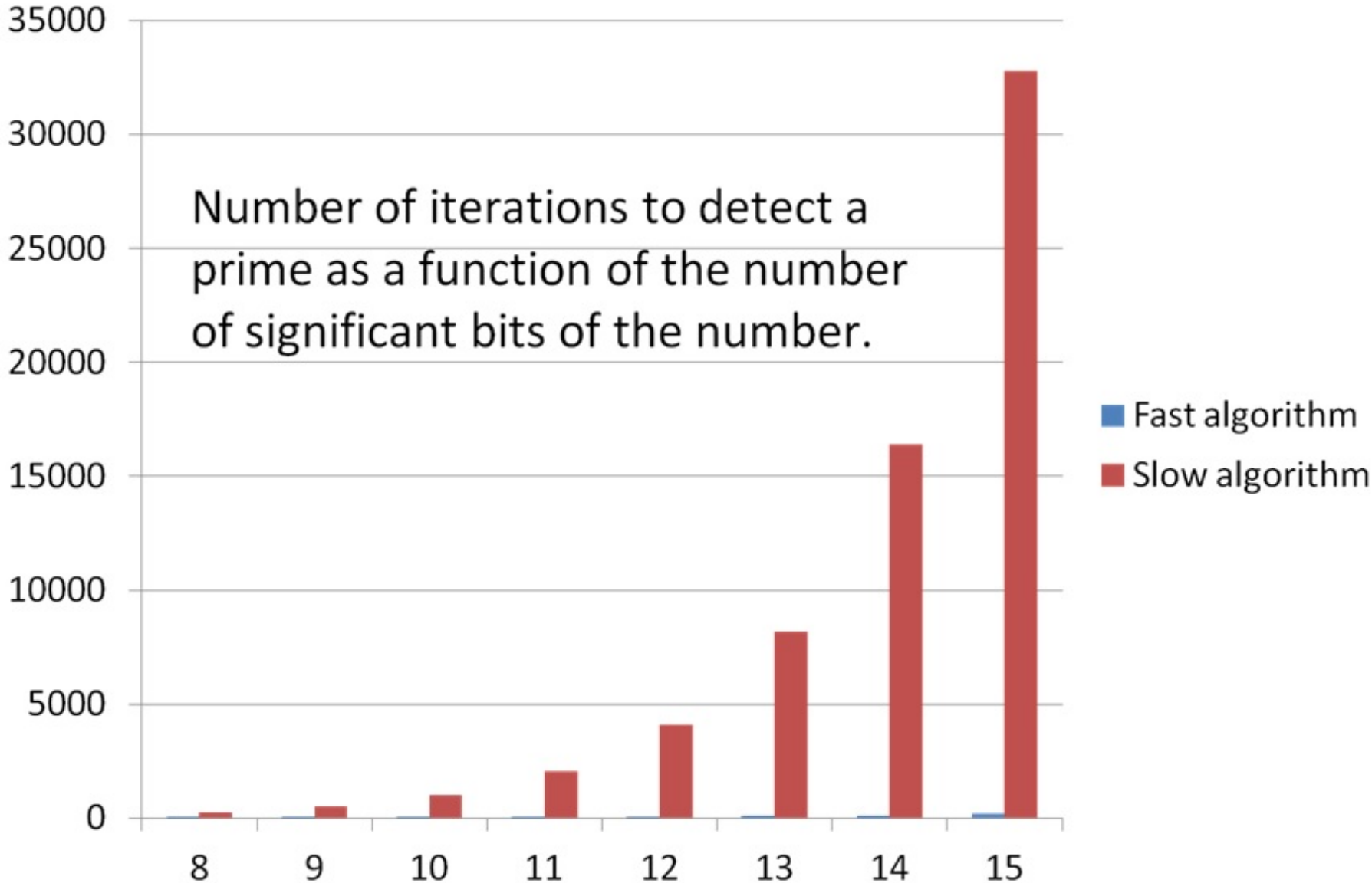
Para usar esta función sólo se necesita conocer su especificación

```
bool is_prime(int n);
/* Pre: n > 0 */
/* Post: Devuelve "true" si n es primo, y
   "false" en otro caso. */
```

Abstracción mediante Especificación Pre/Post

Si mejoramos la implementación de la función **is_prime** sin alterar su especificación, los programas que utilizaban esta función seguirán funcionando sin necesidad de modificarlos y además lo harán de forma mucho más eficiente.

```
bool is_prime(int n) {
/* Pre: n > 0 */
/* Post: Devuelve "true" si n es primo, y
   "false" en otro caso. */
   int divisor = 2;
   while (divisor*divisor <= n) {
      if (n%divisor == 0) return false;
      else divisor = divisor + 1;
   }
   return n != 1;
}
```


Number of iterations to detect a prime as a function of the number of significant bits of the number.
35000
30000
25000
20000
15000
10000
5000
0
8
9
10
11
12
13
14
15
Fast algorithm
Slow algorithm

Abstracción de Datos

Consiste en **añadir un nuevo tipo de datos** al lenguaje de programación. Su **especificación** debe incluir **un nombre para el tipo**, y **una serie de operaciones** (especificadas mediante Pre/Post) que permitan **construir, consultar y modificar** datos de dicho tipo.

Para conseguir que el uso del nuevo tipo de datos sea similar al de otros tipos del lenguaje de programación, hemos de asegurar que

- **para usar dicho tipo sólo sea necesario conocer su especificación**), y
- **las modificaciones a la implementación** de dicho tipo de datos **no afecten a su uso**.

Presentaremos varios **ejemplos de abstracción de datos** más adelante, concretamente las clases `Estudiant` y `Cjt_estudiants`.

Abstracción mediante Especificación Funcional y de Datos

En el diseño de programas utilizaremos la técnica de **abstracción mediante especificación** tanto para funciones como para datos. En general, ambos tipos de **especificación** (funcional y de datos) **ocultan los detalles de la implementación**.

- **Abstracción funcional**: consiste en encargar la solución de una parte del problema a una operación independiente, descrita simplemente mediante su especificación Pre/Post, dejando su implementación para más adelante.
  $\rightarrow$ *Añadir nuevas operaciones* al lenguaje de programación.
- **Abstracción de datos**: consiste en suponer la existencia de un nuevo tipo de datos, descrito simplemente mediante su especificación, dejando su implementación para más adelante. La **especificación** incluirá **un nombre para el tipo**, y **una serie de operaciones** (especificadas a su vez mediante Pre/Post) para **construir, consultar y modificar** datos de dicho tipo.
  $\rightarrow$ *Añadir un nuevo tipo de datos* al lenguaje de programación.

Descomposición Funcional y de Datos: Tipos de Módulos

Los dos tipos de abstracción mediante especificación (funcional y de datos) permiten identificar a su vez **dos tipos de módulos** en el **diseño modular de un programa**.

Si sólo encapsulamos un conjunto de operaciones, definiremos un módulo funcional. Si además las operaciones están definidas sobre un nuevo tipo de datos, especificaremos un módulo de datos.

- **Módulo funcional** contiene un conjunto de operaciones nuevas (con sus especificaciones Pre/Post) que realizan ciertas tareas.
- **Módulo de datos** contiene la definición de un nuevo tipo de datos y sus operaciones (con sus especificaciones Pre/Post).

En la **especificación** de ambos tipos de módulos **se ocultará la implementación de las operaciones**, y en los módulos de datos también se ocultará **la representación del nuevo tipo de datos**.

Fases del Diseño Modular: Especificación e Implementación

La técnica de abstracción mediante especificación permite descomponer un programa en módulos independientes. Pero para conseguir esta independencia es fundamental distinguir las siguientes **fases del diseño modular**, y respetar **su orden de ejecución**.

1. Fase de especificación: consite en asumir la existencia de una representación de los datos y unas operaciones para manipularla (para cada módulo de datos), o la existencia de unas operaciones (para cada módulo funcional).

- No se eligen representaciones concretas para los datos (para los módulos de datos), pero sí se determina el comportamiento de las operaciones mediante sus especificaciones Pre/Post.
- El resultado de esta fase es la especificación de cada módulo, una especie de *contrato de uso* de dicho módulo.

2. Fase de implementación: consiste en elegir la representación de los datos (para módulos de datos) e implementar las operaciones manipulando dicha representación.