



# Programación 2

## -Presentación y Modularidad-

Fernando Orejas

1. Introducción a la asignatura.

2. Modularidad

Nombre:

- Fernando Orejas (orejas@cs.upc.edu)

Despacho:

- 234 (Edificio Omega)

Consultas:

previa cita

Consultas por correo electrónico:

Sí, si la respuesta es poco compleja y no puede ser obtenida fácilmente por otros medios

## Evaluación

- Control de Laboratorio (10%)
- Práctica (25%)
- Examen de la práctica (15%)
- 2 Exámenes parciales (25% cada uno)

Objetivo de la asignatura:

Aprender a programar

Objetivo de la asignatura:

Aprender a programar ... un poco más.







## Objetivos concretos:

Aprender a construir programas (algo) grandes.

Aprender a asegurarnos de que nuestros  
programas son correctos

## Programa:

1. Programación modular
2. Estructuras de datos lineales
3. Árboles
4. Corrección de programas iterativos y recursivos
5. Programación recursiva
6. Mejoras en la eficiencia de programas iterativos y recursivos
7. Tipos recursivos de datos

# *Programación modular*

Qué cualidades ha de tener un buen programa?

# 1. Corrección



## 2. Legibilidad.



### 3. Eficiencia.



## Dos aspectos básicos:

- Descomposición en *partes*
- Especificación

# Descomposición modular

- Hace más comprensibles los programas
- Facilita el diseño y la corrección
- Puede facilitar la modificación
- Incrementa la reutilización de software
- Posibilita el trabajo en equipo

# Descomposición modular

- Módulos con cohesión interna fuerte
- Módulos con acoplamiento débil

## Descomposición:

Módulos basados en abstracciones:

- Abstracciones funcionales
- Abstracciones de datos

## Contar palabras

Se desea diseñar un programa que lea un texto acabado en un punto y nos devuelva la lista de palabras que hay en el texto, ordenada por orden alfabético, indicando el número de veces que aparece cada palabra. Se supone que, como máximo, habrá 10.000 palabras distintas.

## Contar palabras

Por ejemplo, dado el texto:

Mi mama me mima, mi mama me ama, ¿me ama  
mi mama?.

La respuesta debería ser:

ama 2

mama 3

me 3

mi 3

mima 1

```
struct num_palabra{
    string p;
    int n; // numero de veces que aparece p
}
```

```
const int max_pals = 10000;
```

```
struct Lista_pal{
    vector <num_palabra> v;
    int sl; // 0 <= sl <= 10000
}
```

```
// Pre: --
// Post: lee la siguiente palabra del texto y devuelve un
// booleano que nos dice si se ha acabado el texto
bool leer_palabra(string & S) ;
```

```
// Pre: --
// Post: guarda la palabra en el vector L, si ya estaba
// incrementa su numero de apariciones, si no, la añade,
// indicando que ha aparecido una vez.
void guardar_palabra(Lista_pal& L, string & S) ;
```

```
// Pre: --
// Post: Se han escrito las palabras de L
void escribir (const Lista_pal& L);
```

```
int main() {
    Lista_pal L;
    L.v = vector <num_palabra>(max_pals);
    L.sl = 0;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") guardar_palabra(L,S);
    }
    escribir(L)
}
```

```
bool leer_palabra(string & S){  
    bool fin = false;  
    bool ini_pal = false;  
    bool fin_pal = false;  
    string S; char c;  
    while ((not fin) and (not fin_pal) and (cin>>c)){  
        if (c == '.') fin = true;  
        else if (not es_letra(c) and ini_pal)  
            fin_pal = true;  
        else if (es_letra(c) and ini_pal){  
            S.push_back(minusc(c)); ini_pal = true;  
        }  
    }  
    return fin;  
}
```

```
void guardar_palabra(Lista_pal& L,
                      string & s) {
    int i = 0;
    while (i < L.sl) {
        if (s == L.v[i].p) {
            ++L.v[i].n;
            return;
        }
        ++i;
    }
    L.v[L.sl] = {s, 1};
    ++L.sl;
}
```

```
bool comp(<num_palabra> np1, <num_palabra> np2) {
    return np1.S < np2.S;
}

void escribir(const Lista_pal& L) {
    sort(L.v.begin(), L.v.begin() + L.sl, comp);
    for (int i = 0; i < L.sl; ++i) {
        cout << L.v[i].S << " " << L.v[i].n << endl;
    }
}
```

## Problemas:

- Protección de la implementación
- Modificabilidad
- Poca estructura

## La clase Lista\_pal

```
class Lista_pal {  
    private:  
        static const int max_pals = 10000;  
        vector <num_palabra> v;  
        int sl; // 0 <= sl <= 10000  
        bool comp(<num_palabra> np1, <num_palabra> np1) {  
            return np1.S < np2.S;  
        }  
    public:  
        // Constructora  
        // Pre: --  
        // Post: crea una lista de palabras vacía  
        Lista_pal (){  
            v = vector <num_palabra>(max_pals);  
            sl = 0;  
        }  
}
```

## Clase Lista\_pal

```
// Modificadora
void guardar_palabra(string & s) {
    int i = 0;
    while (i < sl) {
        if (s == v[i].p) {
            ++v[i].n;
            return;
        }
        ++i;
    }
    v[sl] = {s,1};
    ++sl;
}
```

## Clase Lista\_pal

```
// Operación de escritura
void escribir() {
    sort(v.begin(), v.begin()+sl, comp);
    for (int i = 0; i < sl; ++i) {
        cout << v[i].S << " " << v[i].n << endl;
    }
}
```

```
int main() {
    Lista_pal L;
    //    L.v = vector <num_palabra>(max_pals);
    //    L.sl = 0;
    string S;
    bool fin = false;
    while (not fin) {
        fin = leer_palabra(S);
        if (S != "") L.guardar_palabra(S);
    }
    L.escribir()
}
```

# Clases y Objetos

- Las clases tienen una parte privada y una pública
- Los objetos son los elementos (variables y constantes declaradas) que tienen de tipo a la clase
- Los objetos de la clase "tienen" todos los atributos de la clase (salvo los estáticos).
- Las operaciones (métodos) tienen como *parámetro implícito* al objeto al que "pertenecen" (salvo que sean estáticas). Por ejemplo

**L.guardar\_palabra(s)**

en vez de

**guardar\_palabra(L,s)**

- Las otras operaciones que trabajen sobre ese tipo, pero que no estén en la clase, no se consideran métodos.

# Operaciones de una Clase

Las operaciones de una clase se dividen en

- Constructoras: son operaciones para construir los objetos básicos. No tienen tipo de resultado
- Destructoras: destruyen los objetos (los eliminan de la memoria)
- Modificadoras: modifican el parámetro implícito
- Consultoras: suministran información contenida en el objeto.
- Entrada/Salida: Escriben información contenida en el objeto.

# Diseño Orientado a Objetos

1. Identificar las clases que juegan un papel en el programa

El propio enunciado es una buena fuente de información. También el esquema de implementación que tengamos en la cabeza.

2. Especificar dichas clases
3. Implementar el programa principal en términos de las operaciones y objetos definidos por las clases.
4. Implementar las clases especificadas.
5. Para implementar las clases puede ser necesario definir, especificar e implementar nuevas clases

# Especificación

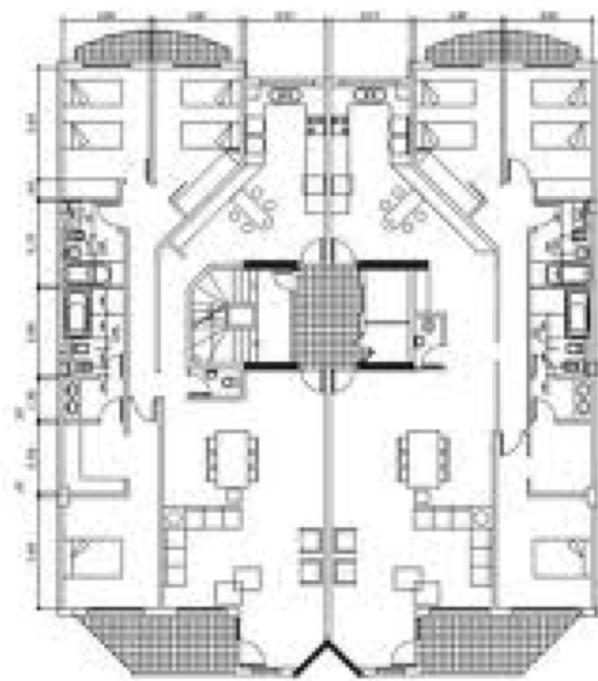
- La especificación es muy importante:

Sin especificación no sabemos que hacen los módulos que usamos.

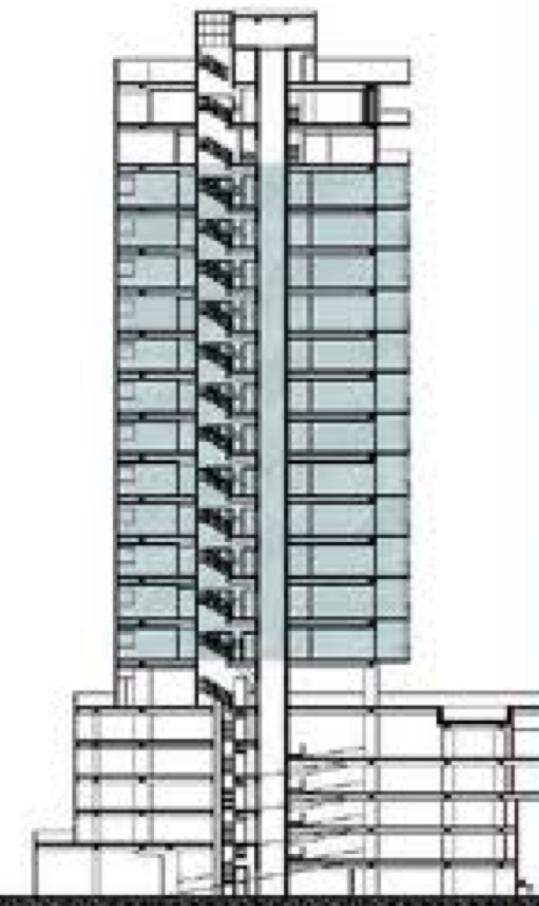
- La especificación es un contrato

Cualquier cambio en la implementación, si respeta la especificación no afecta al resto del programa

## Especificación o modelado



22  
21  
20  
19  
18  
17  
16  
15  
14  
13  
12  
11  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1



## Especificación o modelado

- Funcional
- Relación con otras partes
- Comportamiento
- Deploying
- Calidad de servicio

# Especificación e implementación de una Clase

1. Describimos qué son los objetos de la clase
2. Para cada operación de la clase definimos su Pre y su Post
3. Elegimos una representación para los objetos: especificamos su invariante y su equivalencia
4. Implementamos sus operaciones
5. Si hace falta, utilizamos funciones auxiliares (privadas)

## Ficheros de una Clase

Es conveniente separar en ficheros diferentes la especificación de la implementación de una clase:

- .hh : Contiene la especificación de la clase, aunque también las cabeceras y atributos de la parte privada.
- .cc : Contiene la implementación de las operaciones de la clase

## Especificación de una clase (fichero .hh)

```
class Lista_palabras {  
    private:  
        static const int max_pals = 10000;  
        vector<num_palabra> v;  
        int sl; // 0 <= sl <= 10000  
  
        static bool comp(<num_palabra> np1, <num_palabra> np2) {  
            return np1.S < np2.S;  
        }  
/* Equivalencia de la representación: dos listas son iguales  
si contienen las mismas palabras, con la misma cantidad de  
apariciones */  
  
/* Invariante de la representación: sl nos dice el número de  
palabras que hay en la lista */
```

## Especificación de una clase (fichero .hh)

```
// Pre: --
// Post: crea una lista de palabras vacía
Lista_pal ();
```

```
// Pre: --
// Post: crea una lista de palabras vacía
Lista_palabras ();
```

```
// Pre: --
// Post: Añade la aparición de una palabra a la lista
void guardar_palabra(string & s);
```

```
// Pre: --
// Post: Escribe por orden alfabético las palabras de la
//lista junto con el número de veces que han aparecido
void escribir();
```

```
}
```

## En la implementación de una Clase

- En la cabecera de las operaciones, calificamos sus nombres con el nombre de la clase. Por ejemplo:

**Lista\_pal::guardar\_palabra**

- Nos referimos a los atributos de la clase directamente. Por ejemplo:

**v[s1]** en lugar de **L.v[L.s1]**

- Cuando nos queremos referir al parámetro implícito, usamos **this** y **this->s1** para referirnos a un atributo del objeto

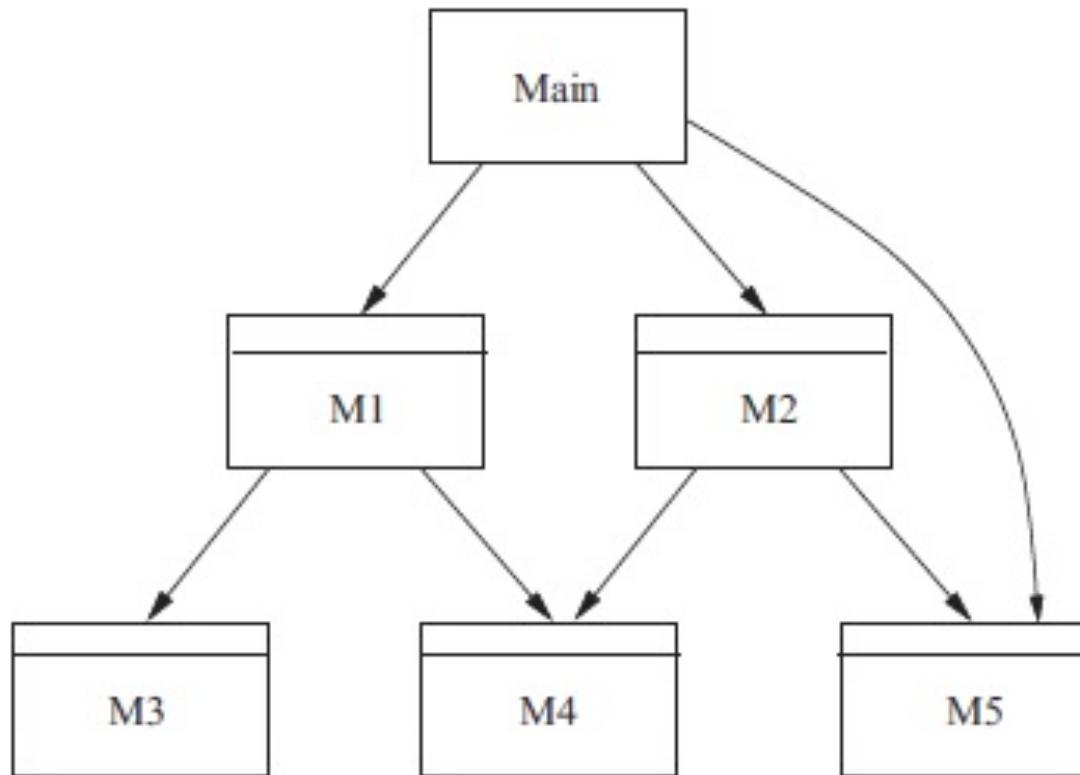
## Implementación de una clase (fichero .cc)

```
#include "Lista_palabras.hh"
void Lista_pal::guardar_palabra(string & s) {
    int i = 0;
    while (i < sl) {
        if (s == v[i].p) {
            ++v[i].n;
            return;
        }
        ++i;
    }
    v[sl] = {s,1};
    ++sl;
}
```

## Implementación de una clase (fichero .cc)

```
Lista_pal::Lista_palabras (){  
    v = vector <num_palabra>(max_pals);  
    sl = 0;  
}  
  
void Lista_pal::escribir() {  
    sort(v.begin(), v.begin()+sl, comp);  
    for (int i = 0; i < sl; ++i) {  
        cout << v[i].S << " " << v[i].n << endl;  
    }  
}
```

# Diagramas modulares



# Relaciones entre módulos

Programa = conjunto de módulos relacionados/dependientes

Un módulo puede:

- Definir un nuevo tipo de datos
- Enriquecer o ampliar tipos con nuevas operaciones

Las relaciones de uso pueden ser:

- Visibles (en la especificación)
- Ocultas para una implementación concreta

# Ampliación de tipos de datos

Si queremos añadir nuevas operaciones a un tipo de datos, podemos hacer tres cosas:

- Definir las nuevas operaciones fuera de la clase
- Añadir los nuevos métodos a la clase.
  - Si necesitamos acceder a la parte privada de la clase
- Usar el mecanismo de herencia (no en P2)

## Solución 1

- No se modifica ni la especificación, ni la implementación de la clase.
- Se puede hacer en un módulo nuevo o en una clase que la utilice.
- Sin parámetro implícito
- Puede ser ineficiente

## Solución 2

- Hay que poder modificar la clase y entender cómo está implementada.
- Hay que añadir las cabeceras (o el código) en los ficheros de la clase.
- Puede ser una solución más eficiente
- Adicionalmente, se puede cambiar la implementación de la clase, lo que implicará modificar las operaciones