Metodología de la Programación II

Análisis y Diseño en Programación

Objetivos

- Entender el proceso de Ingeniería del Software.
- Comprender el concepto de software de calidad.
- Conocer la evolución de los paradigmas del desarrollo del software.
- Conocer los modelos fundamentales de descripción del sistema.
- Comprender los diferentes paradigmas de la programación.
- Conocer y aplicar correctamente los principios de Análisis y Diseño.

1.1 Concepto de Ingeniería del Software

Ingeniería del Software: El establecimiento y uso de principios de ingeniería robustos, orientados a obtener software que sea fiable y funcione eficientemente sobre máquinas reales.

La I.S. se dedica al estudio y desarrollo de métodos, herramientas y procedimientos para el desarrollo eficaz de software de calidad.

Abarca desde el análisis del problema hasta el mantenimiento y la evolución.

Los objetivos de la I.S. se centran en que el desarrollo del software y el producto resultante cumplan diversas propiedades que establezcan su calidad.

1.2 Propiedades del software

- Corrección. Capacidad del software para realizar con exactitud las tareas encomendadas tal y como se definen en las especificaciones.

- Robustez. Capacidad para reaccionar apropiadamente ante condiciones excepcionales. Un sistema que no sea robusto puede ser fatal si el sistema controla algun proceso crítico. Ejemplo: un sistema para controlar la temperatura de un reactor nuclear (¡o algo tan simple y frecuente como escribir en un disco duro!).

- Comprensibilidad. El software debe ser perfectamente inteligible para cualquier persona con formación en programación.

- Coste efectivo. El coste en recursos empleados durante el desarrollo debe estar en conformidad con la funcionalidad necesaria.

- Reusabilidad. Capacidad de los elementos software de servir para la construcción de muchas aplicaciones diferentes.

- Extensibilidad y mantenibilidad. Capacidad para adaptar el software a los cambios de especificación. En particular, para modificar o aumentar su funcionalidad y realizar correcciones.

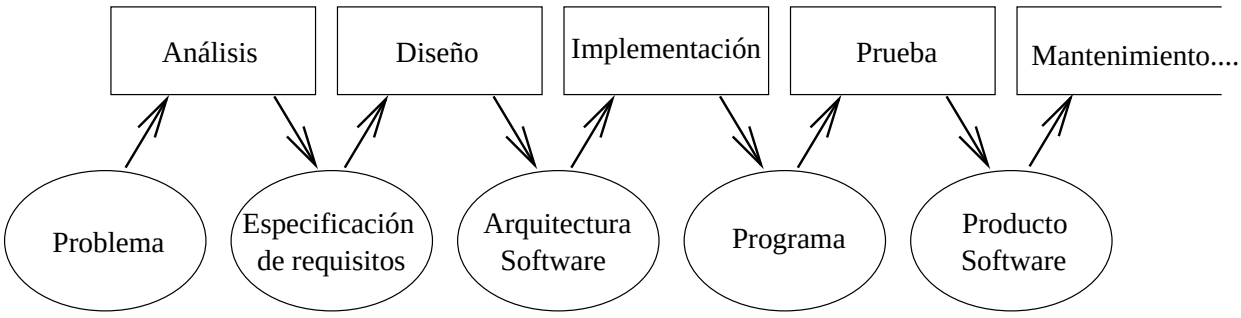
- Eficiencia. Capacidad de un sistema software para exigir la menor cantidad posible de recursos hardware, tales como tiempo del procesador, espacio de memoria interna y externa ocupado o ancho de banda utilizado en los dispositivos de comunicación.

- Portabilidad. Facilidad para transferir los productos software a diferentes entornos hardware y software.

- Facilidad de uso. Facilidad con la que personas con diferentes formaciones y aptitudes pueden aprender a usar los productos software y aplicarlos a la resolucion de problemas. Incluye la facilidad de instalación, de operación y de supervisión.

1.3 Etapas en la Ingeniería del Software

- Ingeniería de Sistemas
- Análisis
- Diseño
- Implementación
- Prueba
- Mantenimiento



1.4 Paradigmas de desarrollo del software

- Lineal secuencial (en cascada) Las actividades se ejecutan linealmente, sin solapamiento.

- Evolutivos. Obtención de versiones sucesivas mejoradas mediante el solapamiento y refinamento de las actividades.

- Ensamblaje de componentes. Adquisición o elaboración de componentes reusables, para su integración en el software objetivo.

- Métodos formales. Desarrollo del software a partir de especificaciones y formalizaciones matemáticas.

Normalmente se emplea una mezcla de los paradigmas anteriores.

1.5 Modelo del sistema

El sistema debe describirse mediante un modelo preciso y sin ambigüedades para poder ser abordado correctamente según sus características.

- Orientado a componentes. Descomposición en componentes autónomas, con posibilidad de interacción.

- Orientado al flujo de datos. Descripción de los datos y de su procesamiento para la obtención de los resultados.

- Orientado a objetos. Determinación de los objetos que componen el sistema y la interacción entre ellos.

- Entidad-relación. Orientado a BD, donde aparecen entidades y relaciones.

1.6 Paradigmas de Programación

Paradigma de programación: Conjunto de principios y reglas según los cuales se deben aplicar los elementos de un lenguaje de programación para desarrollar buenos programas.

Principales paradigmas de programación:

- Procedimental
- Modular
- Abstracción de Datos
- Orientada a objetos
- Genérica

C++ soporta todos estos paradigmas

1.6.1 "Programación" Anárquica

Empleamos el término "Programación Anárquica" para hacer referencia a un "estilo" de programación que **NO** hace consideraciones de diseño. Simplemente se enlaza una sentencia del lenguaje de programación tras otra, creando un programa monolítico sólo entendible por su creador y en los momentos cercanos a su génesis.

Cualquier modificación que hubiese que realizar posteriormente requiere un enorme esfuerzo.

Nunca debe programarse así

1.6.2 Programación Procedimental

Se analizan las tareas necesarias para resolver el problema, se diseñan los algoritmos más eficaces para su realización y se implementan mediante **funciones** y/o **procedimientos**.

Los lenguajes que la soportan incluyen mecanismos para definir unidades de procesamiento (**procedimientos o funciones**) y las formas de interaccionar entre ellos y usarlos (paso de parámetros).

Las funciones y procedimientos permiten la ocultación de información. Una vez implementado un algoritmo, nos olvidamos de "cómo se hace" y sólo atendemos a "qué hace" y cómo usarlo (llamadas y paso de argumentos).

C++ soporta la programación procedimental mediante el uso de funciones

Ejemplo: Función para el cálculo de una raíz cuadrada.

```
double raiz_cuadrada(double x)
{
  // Calculo de la raiz cuadrada de x
}
...
double x = 4.0;
double y = raiz_cuadrada(x);
```

1.6.3 Programación Modular

Cuando aumenta la complejidad de los problemas, es necesario dividirlos en subproblemas para facilitar su resolución. Cada subproblema se aborda independientemente, combinando las subsoluciones para llegar a la solución final.

La programación modular se basa en la creación de unidades llamadas *módulos*, que agrupan datos y procedimientos relacionados entre sí.

La filosofía de este paradigma es la **ocultación de datos**: se agrupan un conjunto de procedimientos relacionados junto con los datos que manipulan en una misma unidad (módulo). Los datos sólo deben ser accesibles a través de los procedimientos del módulo.

Cada módulo tiene una interfaz bien definida, que permite un uso fácil por parte de otros módulos. Además, cada uno es desarrollado independientemente de los demás. Esto requiere la posibilidad de **compilación separada**.

Los módulos ofrecen un segundo nivel de ocultación de información, no sólo con respecto a los algoritmos, sino también a los datos.

Ejemplo: módulo *matrices-bidimensionales*. Proporciona un conjunto de procedimientos (suma, resta, ...) para usar y manipular una matriz bidimensional.

C++ soporta la programación modular mediante la compilación separada

1.6.4 Programación basada en la Abstracción de Datos

Surge de la necesidad de ampliar el conjunto de tipos de datos ofrecidos por un lenguaje, para lo que se definen nuevos tipos de datos abstractos:

- Modelado de entidades mediante un tipo de dato.
- Conjunto completo de operaciones para manipular los objetos del tipo, garantizando que siempre están en un estado adecuado (p.ej. iniciados correctamente). Se establece un vínculo estrecho entre los datos y los procedimientos que los manipulan.
- Ocultamiento de información: el usuario no conoce la forma concreta en que se representan los datos, ni los detalles de implementación de los procedimientos que los manipulan.

Los nuevos TDA se usan igual que los tipos de datos *nativos* del lenguaje.

Ejemplos de TDAs: Tipo *número_complejo*, *Matriz*, *cuenta_bancaria*, *imagen*, etc.

C++ soporta la abstracción de datos mediante la definición de nuevos tipos de datos mediante clases

```
class Matriz {
public:
   /// @brief Constructor.
   Matriz(n);

   /// @brief Operador de suma.
   Matriz operator+(const Matriz & m);

   /// @brief Calculo del rango.
   int rango() const;
private:
  ...
};

int main()
{
  int a, b, c;

  a = 2;   b = 5;
  c = a + b;

  Matriz(2) M, N, O;
  M = {{1, 0}, {0, 1}};
  N = {{2, 3}, {4, 5}};
  O = M + N;
  cout << M << " + " << N << " = " << O << endl;
}
```

1.6.5 Programación Orientada a Objetos

La POO está basada en los objetos, que son modelizaciones de entidades relacionadas del mundo real.

Hace uso de:

- *Encapsulamiento*. Permite la abstracción de datos combinando datos y operaciones en un mismo objeto.

- *Herencia*. Permite representar objetos relacionados (relación es-un).

- *Polimorfismo*. Permite la construcción de operaciones que se actuarán de diferente forma según el tipo de los objetos implicados.

Ejemplo: Supongamos un sistema de representación gráfica. Necesitamos procesar rectángulos y triángulos. Los modelamos con sendos TDAS. Ambos comparten propiedades (por ejemplo, posición) y comportamiento (función pintar).

Podemos *reutilizar el código* definiendo un TDA figura_gráfica que recoja los elementos comunes de todos los TDAs gráficos y después, definiendo cada TDA particular como un TDA *derivado* o *descendiente* del base figura_gráfica.

C++ soporta la POO mediante el concepto de clase y los mecanismos de encapsulamiento, herencia y polimorfismo

1.6.6 Programación Genérica

Se basa en la creación de algoritmos que se pueden aplicar no sólo a datos distintos del mismo tipo, sino a distintos tipos de datos. Son esquemas de algoritmos parametrizados que se pueden usar con datos de distintos tipos y no sólo de un tipo único preespecificado como ocurría en el paradigma procedimental.

C++ soporta la programación genérica mediante plantillas e iteradores

Ejemplo: Ordenación de vectores de datos

```
/**
  @brief Ordena un vector de enteros.
  @param v: vector a ordenar. Debe tener 'tam'
            elementos.
  @param tam: dimensión de 'v'. tam > 0.

  Dispone los elememos te 'v' en sentido creciente.
*/
void ordena_enteros(int v[], int tam)
{
  ...
}
```

```
/**
  @brief Ordena un vector de elementos T
  @param v: vector a ordenar. Debe tener 'tam'
            elementos.
  @param tam: dimensión de 'v'. tam > 0.

  @precondition Los datos del tipo T deben tener
           definida una operacion de comparación:
           bool operator < (const T &, const T &);

  Dispone los elememos te 'v' en sentido creciente.
*/
template <class T>
void ordena_generica(T v[], int tam)
{
  ...
}


int main() {
  int ve[10] = { ... };
  string vs[10] = { ... };

  ordena_enteros(ve, 10);
  ordena_generica(ve, 10);
  ordena_generica(vs, 10);
}
```

1.7 Tendencias del desarrollo del software

- Análisis y diseño estructurado: Modelo orientado al flujo de datos y a componentes, pudiendo aplicar los paradigmas procedimental, modular, abstracción de datos e incluso programación genérica

- Análisis y diseño orientado a objetos: Modelo orientado a objetos, aplicando la programación orientada a objetos

- Análisis y diseño basado en modelos entidad-relación: Programación de bases de datos.

Vamos a considerar el análisis y diseño basado en el desarrollo de software secuencial lineal.

1.8 Análisis

El Análisis consiste básicamente en una descripción clara y precisa del problema, siendo necesaria para garantizar una comprensión clara y completa del problema a resolver.

- Los conceptos relacionados con el problema se exploran y refinan, descomponiendo el problema en subproblemas.

- Se redacta un documento de especificación: "lo que se supone que es el producto".

- Como resultado de esta fase se tiene un plan redactado en el que se describe el desarrollo del software de forma detallada: el plan de gestión del producto software.

1.8.1 Problemas y consideraciones en el Análisis

- Información incompleta.
- Existencia de contradicciones en la información.
- Las soluciones de los subproblemas, deben posibilitar la combinación de las mismas para la obtención de la solución general del problema.
- Los subproblemas deben ser lo suficientemente genéricos, para poder ser reutilizados en otros problemas, y modificables con pequeño coste
- Generalmente deben evitarse los detalles sobre diseño e implementación.
- Se utilizarán guiones (secuencia de pasos para conseguir objetivos)
- Contemplar los casos excepcionales
- Contemplar la posibilidad de modificaciones.

1.8.2 Pasos en el Análisis

1. Especificar de forma inequívoca el problema.

2. Resolver el problema descomponiéndolo en subproblemas.

3. Para cada subproblema:

   - Identificar entradas, número y tipo
   - Identificar salidas, número y tipo
   - Establecer relaciones entre entradas y salidas
   - Identificar condiciones del problema y la información necesaria

1.9 Diseño

En la etapa de Diseño de forma general, se elabora cómo se realizará el producto

- Se descompone la solución en módulos: partes independientes con una comunicación muy bien definida con el resto de la aplicación.

- Se debe definir de forma clara y precisa, qué hace cada módulo y cómo debe hacerlo.

- Deben definirse las entradas y las salidas con el máximo detalle.

- Se especificarán los algoritmos y las estructuras de datos que se utilizarán en cada módulo

1.9.1 Descomposición Modular

- Cada subproblema deberá resolverse utilizando la metodología de la programación modular.

- Cada módulo debe considerar el ocultamiento de información y la abstracción, diferenciando en cada módulo dos partes:
  - Interfaz: Elementos públicos, visibles por el resto de los módulos.
  - Parte interna. Elementos privados, no visibles por los demás módulos.

- Los elementos que intervengan en las conexiones entre los módulos derivados de los subproblemas deben ser cuantos menos mejor, ya que el aumento de conexiones conllevará:
  - Mayor coste en la modificación de los módulos.
  - Aumento de la dificultad para controlar la fase de pruebas.
  - Aumento de la dificultad en la búsqueda de elementos responsables de errores.

- El uso de datos globales de los que dependan varios módulos dificultará la búsqueda de errores, por lo que debería ser mínimo.

- Los módulos no deben permitir la alteración directa de sus datos internos por parte de otros módulos.

- Debe existir un equilibrio con respecto al número de módulos, teniendo en cuenta por una parte, que cada módulo debe resolver una tarea independiente, y por otra, que mayor número de módulos no implica una solución mejor.

1.9.2 Beneficios de la descomposición modular

- Se facilita el desarrollo de programas por un equipo de programadores
- Facilidad de mantenimiento del software (modificaciones, pruebas, depuraciones, etc.)
- Mayor productividad (eliminación de redundancias, posibilidad de reutilización).

1.9.3 Pasos en el Diseño

- Descomposición modular
  - Diseño de módulos a nivel general y relación entre ellos.
  - Diseño de interfaces con el usuario, el propio sistema u otros sistemas.

- Diseño Modular
  - Selección/Diseño de tipos de datos
  - Selección/Diseño de estructuras de datos necesarias
  - Selección/Diseño de algoritmos para procesar la información

1.10 Implementación

- La implementación debería ser casi directa, si se ha realizado un buen diseño.
- Debe seguirse un buen estilo de codificación en aras de una buena legibilidad, unificando criterios como el de los nombres de identificadores.
- Equilibrio entre eficiencia y legibilidad.
- Inclusión de comentarios no redundantes.
- Inclusión de parte del diseño en el código que permita una mayor comprensión, utilizando alguna herramienta del tipo `doxygen`.

Ejemplo de Análisis

Problema:
A partir del radio de un círculo, calcular y mostrar el área del círculo y la longitud de su circunferencia.

Análisis:
Claramente la entrada es el radio del círculo, que debe ser un valor real positivo. Se piden dos valores de salida: el área del círculo y la longitud de la circunferencia, que también son valores reales positivos. Necesitaremos conocer las fórmulas para el cálculo del área del círculo y la longitud de la circunferencia.

$$\text{área} = \text{radio}^2 \times \pi$$
$$\text{longitud} = 2 \times \text{radio} \times \pi$$

con $\pi = 3{,}14159$.

Interfaz: la comunicación con el usuario será en modo "consola".

Casos excepcionales: no se aceptarán valores negativos de los datos de entrada.

Ejemplo de Diseño

Tipos de datos para entradas y salidas:

Entrada del problema:
`radio`: Tipo real (radio de un círculo).

Salidas del problema:
`area`: Tipo real (área del círculo)
`longCircun`: Tipo real (long. de la circunf.)

Una vez identificadas las entradas y las salidas del problema, debemos enumerar la serie de pasos a seguir para resolver el problema.

Algoritmo inicial:

1. Obtener el radio del círculo.
2. Calcular el área del círculo.
3. Calcular la longitud de la circunferencia.
4. Mostrar el área y la longitud

Refinamiento del algoritmo: refinamos los pasos que no tienen una solución obvia.
Paso 2. Asignamos a `area` = PI x radio x radio
Paso 3. Asignamos a `longCircun` = 2 x PI x radio

Ejemplo de Codificación

```
/// @file: circulo.cpp
/// @brief Calcula y muestra el área de un círculo y
/// @brief la long. de la circunferencia
#include <iostream>
using namespace std;
int main()
{
  const double PI = 3.14159;
  double radio;      // entrada: radio del círculo
  double area;       // salida: área del círculo
  double longCircun; // salida: long. de circunf.

  // Obtener el círculo del radio
  cout << "Introducir el radio del círculo:";
  cin >> radio;

  // Calcular el área del círculo
  area = PI * radio * radio;
  // Calcular la longitud de la circunferencia
  longCircun = 2 * PI * radio;

  // Mostra área y longitud
  cout << "El área del círculo es " << area << endl;
  cout << "La longitud de la circunferencia es "
       << longCircun << endl;
  return 0;
}
```

Ejemplo de Prueba

Probamos varios valores poniendo especial énfasis en aquellos valores que resultan especiales o límite. (Por ejemplo si para una fórmula existen distintos rangos o casos para los que el resultado se sale fuera de la norma).

Los valores "coherentes" para un radio son números reales positivos. Esto divide el conjunto de posibles entradas en dos clases: positivos y negativos y aparece un valor límite: el cero. Debemos **seleccionar** un valor de cada clase y **comprobar** el funcionamiento del programa con estos valores.

Así, seleccionamos 5, como valor positivo y comprobamos que salen como valores 78,54 para el área y 31,42 para la longitud (valores que se saben son correctos, por ejemplo por haber sido obtenidos de forma manual).

¿Qué ocurre con valores negativos del radio?

¿Qué pasa con una entrada igual a 0? ¿Es correcto?

Segundo ejemplo de Análisis

Problema. Supongamos una red telefónica con conexiones directas entre los distintos teléfonos, sin conmutador central. ¿Cuántas líneas se precisan para conectar n teléfonos? ¿Cuántas líneas es preciso añadir para incorporar m teléfonos más a la red?

Análisis: Para este problema se necesitan dos entradas: el número de teléfonos n y el número de teléfonos en que se ampliará la red m. Ambas entradas vienen dadas por valores enteros positivos.

Se piden dos salidas: el número de líneas necesarias para interconectar n teléfonos; y el número de líneas necesarias para incorporar m teléfonos más a la red. Ambas se expresan mediante números enteros positivos. Además, el número de líneas de teléfono requeridas para conectar n teléfonos es l:

$$l = \frac{n(n-1)}{2}$$

Con esta expresión podemos vincular las entradas con las salidas.

Segundo ejemplo de Diseño

Las entradas se representarán usando un tipo entero:

- `tel`: Tipo entero (número de teléfonos (n))
- `tel_adic`: Tipo entero (número de teléfonos a añadir (m))

Para las salidas también usaremos un tipo entero:

- `lin`: Tipo entero (número de líneas)
- `lin_adic`: Tipo entero (número de líneas adicionales)

Algoritmo:

1. Calular `lin` a partir de `tel` usando la fórmula.
2. Calcular el número de líneas para `tel` + `tel_adic` teléfonos usando la fórmula.
3. El número de líneales adicionales necesarias es la diferencia entre el resultado anterior y `lin`.

Segundo ejemplo de Codificación

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

int main()
{
  int tel, tel_adic, lin, lin_adic;

  cout << "Introduce el número de teléfonos:";
  cin >> tel;
  cout << "Introduce el número de teléfonos adicionales:";
  cin >> tel_adic;

  lin = (tel * (tel - 1))/2;
  int t = tel + tel_adic;
  lin_adic = (t * (t - 1))/2 - lin;
  cout << "Número inicial de líneas: " << lin << endl;
  cout << "Número de lineas adicionales: " << lin_adic
       << endl;

  return 0;
}
```

Segundo ejemplo de Prueba

Identificación de grupos de entradas: Las entradas correctas son pares de enteros positivos. Las clases que resultan son:

- Pares de enteros positivos
- Pares con la primera componente negativa
- Pares con la segunda componente negativa
- Pares con una componente no entera (p.ej. real)

Casos límite: pares con una componente igual a 0.