# Parte I: El computador y el proceso de programación

- 1.Introducción a los computadores y su programación
- 2. Introducción al análisis y diseño de algoritmos
- 3. Introducción al análisis y diseño de programas
- 4. Verificación de programas



## Notas:

**1. Introducción a los computadores y su programación**

- Arquitectura básica de un computador.El software del sistema. Lenguajes de Alto Nivel. El proceso de compilación. El ciclo de vida del software.

**2. Introducción al análisis y diseño de algoritmos.**

- Diseño de un programa. Concepto de algoritmo. Descripción de algoritmos: el pseudolenguaje. Tiempo de ejecución de algoritmos. La notación O(n). Ejemplos de análisis.

**3. Introducción al análisis y diseño de programas**

- Actividades del ciclo de vida del software. Paradigmas de desarrollo de programas. Análisis y especificación. Diseño arquitectónico. Técnicas de diseño detallado.

**4. Verificación de programas**

- Importancia de la verificación. Estrategias de prueba. Depuración. Elección de datos para la prueba.

# 1. Importancia de la verificación

**Aproximadamente la mitad del esfuerzo total del desarrollo se emplea en las pruebas del programa.**

**La prueba de un programa depende mucho de su dimensión**

- **cuanto más grande es el programa más etapas son necesarias en el proceso de prueba.**

**Para las pruebas de un programa debe hacerse una planificación rigurosa.**

**Para probar un programa incompleto o módulos de programa es necesario escribir software de prueba**

- **en ocasiones, el software de prueba es más grande que el propio programa.**



## Notas:

Cuando tratamos el esfuerzo necesario para el desarrollo de un sistema software, vimos que aproximadamente la mitad del esfuerzo total se empleaba en las pruebas del programa, lo cual pone de manifiesto la gran importancia de esta etapa del desarrollo.

Según sea la dimensión del programa la problemática asociada a las pruebas de un programa pueden ser completamente diferentes. En programas grandes normalmente la prueba necesita mayor número de etapas que en programas pequeños.

Al igual que para el diseño de un programa, para su prueba es necesario hacer una planificación cuidadosa de todos los elementos que se van a probar. Para ello, se suele elaborar un documento de pruebas, incluso antes de escribir el programa. El rango de las técnicas empleadas para la prueba de programas es muy amplio, y abarca desde el estudio de viabilidad de las especificaciones hasta los programas generadores de las secuencias de pruebas.

Cuando se prueba un programa incompleto o un módulo de programa, es necesario crear software de prueba, que permite introducir o recoger los datos y eventos necesarios. En programas grandes es habitual que el software de pruebas sea más voluminoso que el propio programa.

El software de pruebas de un programa debe conservarse, para poder probar de nuevo el programa cuando se hagan modificaciones.

# Dependencia con el tamaño del programa

| Categoría | Características | Dependencia |
|---|---|---|
| Simple | - < 1000 instrucciones<br>- 1 persona, < 6 meses<br>- Sin interacciones con otros programas | Del programador |
| Complejidad intermedia | - < 10.000 instrucciones<br>- 1-5 personas, < 2 años<br>- Pocas interacciones con otros sistemas<br>- 10 a 100 módulos | Del programador y de la dirección |
| Complejo | - < 100.000 instrucciones<br>- 5-20 personas, < 3 años<br>- Frecuentemente interacciona con otros sistemas.<br>- 100 a 1000 módulos. | Técnicas modernas de diseño y direc-ción |



## Notas:

| Categoría | Características | Dependencia |
|---|---|---|
| Muy complejo | - < $10^6$ instrucciones<br>- 20-100 programadores<br>- Requiere mantenimiento continuo y por gente distinta.<br>- Fuertes interacciones con otros sistemas.<br>- 1000 a 10.000 módulos | Técnicas modernas de diseño y dirección |
| Super com-plejo | - > $10^6$ instrucciones<br>- > 100 programadores<br>- Requiere mantenimiento continuo y por gente distinta.<br>- Casi siempre incluye procesado en tiempo real, telecomunicaciones, etc.<br>- Corresponde, generalmente, a procesos críticos (tráfico aéreo, defensa, etc.) | Idem |

# Definiciones básicas

***Prueba o "Test"*: ejecución para encontrar errores**

***Demostración*: prueba matemática por inspección del código**

***Verificación*: búsqueda de fallos en ambiente simulado**

***Validación*: comprobación en el ambiente real**

***Certificación*: se certifica la corrección cuando se han probado exhaustivamente todas las posibilidades**

***Depuración o "Debugging"*: localizar y corregir errores**



## Notas:

*Testing o Prueba*: Es el proceso de ejecución de un programa (o parte de él) con la intención o meta de encontrar errores.

*Demostración*: Es un intento de encontrar errores si tener en cuenta el ambiente del programa. Es decir, se trata de obtener o probar teoremas matemáticos acerca de la corrección de un programa.

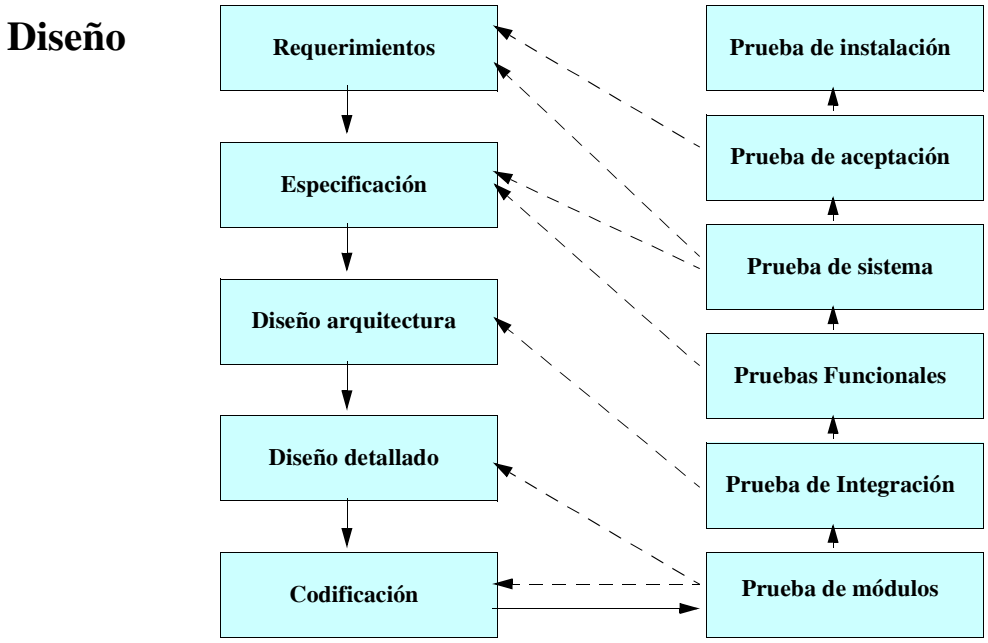
*Verificación*: Es un intento de encontrar errores en un ambiente de test o simulado.

*Validación*: Es un intento de encontrar errores ejecutando un programa en un ambiente real dado.

*Certificación*: Es un certificado de corrección. La prueba para certificación debe realizarse sobre algún tipo de estándar predefinido.

*Depuración*: No es una forma de prueba y, aunque a veces las palabras depuración y prueba se usan de forma intercambiable, son actividades distintas. La prueba es una actividad de encontrar errores, y la depuración es la actividad de diagnosticar la naturaleza precisa de un error, conocerlo y corregirlo. Las dos están relacionadas porque la salida de una actividad de prueba es la entrada de una actividad de depuración.

# Relación: prueba-diseño






## Notas:

*Prueba de módulo*: Consiste en probar cada módulo de programa individualmente. Cuando tratamos un programa pequeño, generalmente es desarrollado y probado como una utilidad única y por una sola persona.

*Prueba de integración*: En la prueba de módulos normalmente no se encuentran los errores relacionados con la interacción entre los diferentes módulos. El proceso de juntar los diferentes módulos individuales para realizar subsecciones o funciones de un programa se denomina integración de sistemas. Cuando se utilizan pruebas para averiguar la corrección de las interfaces entre los módulos se denomina a este proceso de prueba, prueba de integración.

*Pruebas de función y de sistema*: En estas pruebas se analiza el funcionamiento de los módulos ya unidos para comprobar su correcto funcionamiento conjunto en una función o subsistema del programa (prueba de función), o del programa completo (prueba de sistema).

*Prueba de aceptación*: Es el proceso de prueba realizado sobre el programa completo, en ambiente simulado o de prueba, para darlo por válido. Requiere la prueba bajo las condiciones más variadas. Normalmente participa el usuario, que da su visto bueno.

*Prueba de instalación*: Es el proceso de prueba realizado con el programa funcionando en su sistema y ambiente real, una vez instalado.

# 2.Comparación entre las distintas técnicas de prueba

## TOP-DOWN:

| Características | Ventajas | Desventajas |
|---|---|---|
| - El programa principal o de control se prueba en primer lugar<br>- Los módulos se integran uno por uno<br>- Se hace hincapié en las pruebas de las interfaces | - No necesita programas "driver" de prueba<br>- El programa de control y unos pocos módulos forman un prototipo básico<br>- Los errores de interfaces se detectan pronto<br>- La característica modular ayuda a la depuración | - Necesita módulos simulados ("stubs")<br>- Los errores en los módulos críticos se encuentran en último lugar |



## Notas:

Cuando se realiza un diseño top-down o bottom-up puro, lo más lógico es realizar también la prueba con la misma técnica.

En el caso de una prueba **top-down**, ésta comienza analizando el funcionamiento del flujo de control del programa principal examinando cómo pasa el control y los datos a los distintos módulos, cómo éstos los devuelven y cómo el programa de control pasa los datos a los dispositivos de salida.

Para ello, se necesita incluir en esta fase del desarrollo un cierto conjunto de instrucciones en cada módulo, aún cuando éstos aún no han sido diseñados. A estos módulos simplificados que permiten la prueba se les denomina "stubs".

Los "stubs" incluyen la implementación de las interfaces del módulo y un conjunto, normalmente sencillo, de código para su utilización en el test. La complejidad de estos módulos simulados vendrá en función del interés por profundizar en el funcionamiento del sistema.

# Comparación entre distintas técnicas de prueba (cont.)

## BOTTOM-UP:

| Características | Ventajas | Desventajas |
|---|---|---|
| - Permite una prueba temprana de módulos particulares<br>- Los módulos se pueden integrar en diversos grupos<br>- Se hace hincapié en la funcionalidad y prestaciones de los módulos | - No necesita módulos simulados<br>- Se encuentran primero los errores de los módulos críticos<br>- Se ajustan las necesidades humanas a lo existente más fácilmente | - Necesitan programas principales simulados ("test drivers")<br>- Los errores de interfaz se descubren después<br>- Es necesario desarrollar muchos módulos antes de tener un programa parcialmente operativo |



## Notas:

Cuando el diseño es **bottom-up** puro y la prueba también, no necesitaremos los módulos del test anteriores, pero sin embargo necesitaremos "drivers", esto es, programas principales que pasan datos al módulo bajo test y reciben datos de éste, una vez procesados.

Lógicamente, casi nunca, la metodología es una u otra únicamente, sino que ambas técnicas, "top-down" y "bottom-up", se utilizan mezcladas de igual forma que se realiza en el diseño de forma práctica.

# 3. Depuración

## Es el proceso de localización y eliminación de un error una vez detectado.

## Técnicas antiguas:

- Volcado de memoria
- Trazas
- Escritura en pantalla
- Programas de depuración de bajo nivel

## Técnicas modernas:

- Programas de depuración de alto nivel



## Notas:

Cuando se detecta un error, es necesario precisar su localización en el programa (cual es el módulo causante) y una vez localizado se pasa a su eliminación. A este proceso se le denomina depuración o "debugging".

**Volcado de memoria**. Se realiza un registro de todas las posiciones de memoria o variables relevantes de forma periódica. Ventajas: Se obtiene el contenido de la memoria en instantes de tiempo cruciales. Desventajas: Requiere tiempo de CPU. Requiere mucho tiempo de análisis.

**Trazas**. Esencialmente es igual que la anterior, excepto que contiene solamente ciertas porciones de memoria y registros y sus salidas están condicionadas por la ocurrencia de un evento.

**Instrucciones write o put.** Se utiliza la instrucción "Put" en pantalla o la equivalente del lenguaje de programación utilizado, para obtener el valor de las variables deseadas. Ventajas: Es muy simple. Se puede ver la variación del valor de la variable deseada. Desventajas: Son muy molestas de usar en grandes programas. Usadas indiscriminadamente no dan ninguna información. Requieren recompilación.

**Programa de Depuración**. Es un programa que corre concurrentemente con el programa bajo test y suministra comandos para examinar la memoria, parar la ejecución en algún punto, buscar por referencia constantes o variables particulares, etc. Antiguamente los depuradores sólo valían para lenguaje ensamblador, pero hoy en día existen muchos depuradores de lenguajes de alto nivel.

# 4 Elección de datos para la prueba

1. Probar cada una de las grandes características del sistema
2. Casos límite, extremos de los rangos, etc.
3. Datos de entrada inusuales o incorrectos
4. Cobertura de todos los caminos del programa
5. Pruebas adicionales aleatorias o en función del algoritmo



## Notas:

La mejor prueba es aquella que realiza la comprobación para todos los valores de los datos de entrada y todas las condiciones iniciales. Este test, salvo en casos triviales, es prácticamente imposible. Por ello, es importante tener una guía para escoger el subconjunto de datos de entrada más adecuado, aunque no existe una formalización clara acerca de esta elección:

1. Generar un conjunto de datos de prueba para ejercitar cada una de las grandes características del programa
2. Generar pruebas para comprobar los casos límite entre dos conjuntos de soluciones, extremos de los rangos de solución, etc.
3. Generar pruebas con datos de entrada inusuales o incorrectos.
4. Comprobar si ya se ha realizado una cobertura de todos los caminos del programa y, si no es así, añadir las pruebas necesarias para ello.
5. Escoger casos adicionales mediante técnicas aleatorias y/o por un análisis en profundidad del algoritmo.