# TEMA 30

# PRUEBA Y DOCUMENTACIÓN DE PROGRAMAS. TÉCNICAS.

**ÍNDICE**

## 1. INTRODUCCIÓN

Uno de los problemas fundamentales cuando se desarrolla un algoritmo y se plasma en un programa de ordenador, es asegurarse de que el programa es, en efecto, idéntico al algoritmo, en el sentido de que ambos producen los mismos resultados para cualquier combinación de datos de entrada. Si esto es cierto, decidimos que el programa es *correcto*. Si no lo es, cada discrepancia entre el programa y el algoritmo se considera un *error* del programa. La fase de pruebas (depuración y verificación) de un programa consiste esencialmente en la detección y eliminación de todos los errores.

Desgraciadamente, cuando el algoritmo es muy complejo (lo que implica que el programa que lo plasma también lo será) es muy difícil asegurarse de que el programa es correcto. Todos los fabricantes de software son conscientes de ello y dedican recursos, a veces considerables, al mantenimiento de sus productos.

Si, a pesar de nuestros esfuerzos, se nos ha escapado algún error, el programa puede reaccionar ante esta situación de maneras muy diferentes:

> Generando un resultado incorrecto, sin avisar que lo es.
> No terminando nunca, por ejemplo, metiéndose en un bucle permanente sin salida.
> Terminando de forma desordenada, por ejemplo, saliendo al sistema operativo sin generar mensaje alguno. Un caso peor ocurre cuando el sistema operativo queda bloqueado a causa del error y es preciso reinicializar la máquina.
> Terminando de forma ordenada, generando un mensaje de error significativo que informe al usuario del problema detectado.
> Detectando el error y dando al usuario oportunidad de corregirlo y continuar con la ejecución sin necesidad de volver atrás.

De las cinco posibilidades, las tres primeras son erróneas.

Decimos que un programa es **robusto** si, cualquiera que sea la combinación de sus datos de entrada, no se presenta ninguna de las tres primeras situaciones. Si además, en todos los casos, se da la situación quinta, diremos que es **amigable**.

## 2. PRUEBA DE PROGRAMAS

La prueba del software es un elemento crítico para la garantía de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. La prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software.

### 2.1. Fundamentos de la prueba de programas

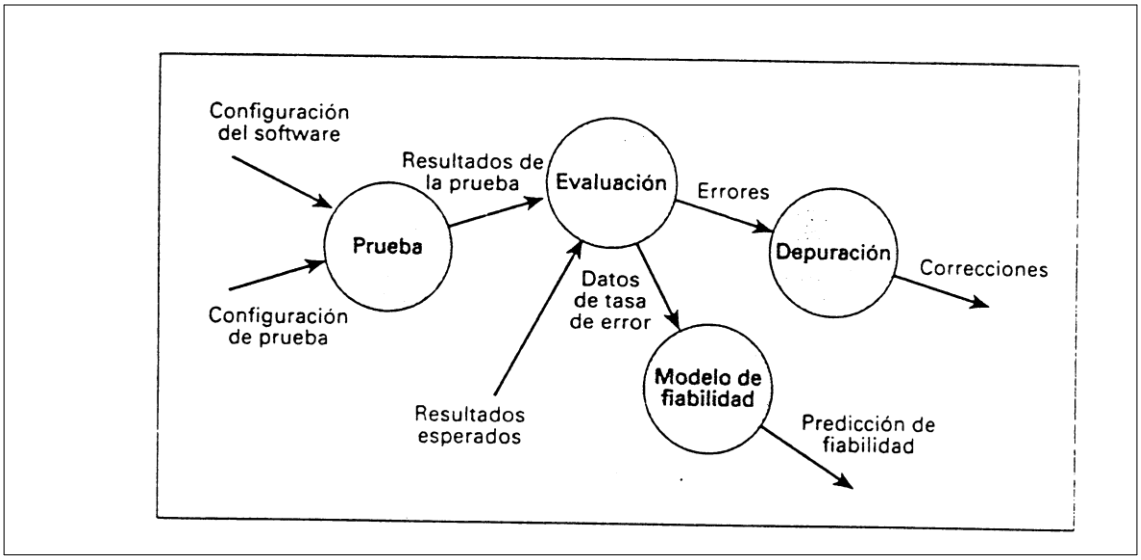
Para un buen diseño de las pruebas de programas se han de tener presentes las siguientes reglas:

1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.

2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.

3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

El flujo de información para la prueba sigue el esquema descrito en la Figura. Se proporcionan dos clases de entradas al proceso de prueba: **(1)** una configuración del software que incluye la *especificación de requisitos del software*, la *especificación del diseño* y el código fuente; **(2)** una *configuración de prueba* que incluye un *plan* y *procedimiento de prueba*, casos de prueba y resultados esperados.




Se llevan a cabo las pruebas y se evalúan los resultados. Es decir, se comparan los resultados de la prueba con los esperados. Cuando se descubren datos erróneos, implica que hay en error y comienza la depuración.

### 2.2. Técnicas de prueba

Cualquier conjunto de programas puede ser probado de una de las dos formas: **(1)** conociendo la función específica para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa; **(2)** conociendo el funcionamiento del producto, se pueden desarrollar pruebas que aseguren que "todas las piezas encajan"; o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada. El primer enfoque de prueba se denomina *prueba de la caja negra* y el segundo *prueba de la caja blanca*.

#### 2.2.1. Prueba de la caja blanca

Cuando se considera el software de computadora, la prueba de la caja negra se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa (p. ej.: archivos de datos) se mantiene. Una prueba de la caja negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

Mediante los métodos de prueba de la caja blanca, el ingeniero del software puede obtener casos de prueba que: **(1)** garanticen que se ejercitan por lo menos una vez todos los *caminos independientes* de cada módulo; **(2)** ejerciten todas las decisiones lógicas en sus vertientes *verdadera* y *falsa*; **(3)** ejecuten todos los bucles en sus límites y con sus límites operacionales y **(4)** ejerciten las estructuras internas de datos para asegurar su validez.

¿Por qué gastar tiempo y energía preocupándose de (y probando) las minuciosidades lógicas cuando podríamos gastar mejor el esfuerzo asegurando que se han alcanzado los requisitos del programa?

> Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
> A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma regular.
> Los errores tipográficos son aleatorios.

**Prueba del camino básico**

La *prueba del camino básico* es una técnica de prueba de la caja blanca.

El método del camino básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un *conjunto básico* de caminos de ejecución. Los casos de prueba derivados del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

Los pasos a seguir para realizar la prueba del camino básico son:

1. Usando el diseño o el código como base, dibujamos el correspondiente grafo de flujo.

2. Determinamos la complejidad ciclomática del grafo de flujo resultante.

3. Determinamos un conjunto básico de caminos linealmente independientes.

4. Preparamos los casos de prueba que forzarán la ejecución de cada camino del conjunto básico.

**Prueba de la estructura de control**

La técnica de *prueba del camino base* es una de las muchas técnicas para la *prueba de la estructura de control*. Aunque la prueba del camino base es sencilla y altamente efectiva, no es suficiente por sí sola. En esta sección se tratan otras variantes de la prueba de estructura de control. Estas variantes amplían la cobertura de la prueba y mejoran la calidad de la prueba de la caja blanca.

Las variantes son:

**Prueba de condiciones**. El método de la *prueba de condiciones* se centra en la prueba de cada una de las condiciones del programa. Las estrategias de prueba de condiciones tienen, generalmente, dos ventajas. La primera, es que la medición de la cobertura de la prueba de una condición es sencilla. La segunda, es que la cobertura de la prueba de las condiciones de un programa da una orientación para la generación de pruebas adicionales del programa.

**Prueba de flujo de datos**. El método de "prueba de flujo de datos" selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa.

**Prueba de bucles**. La prueba de bucles es una técnica de prueba de la caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases diferentes de bucles: *bucles simples*, *bucles concatenados*, *bucles anidados* y *bucles no estructurados*.

**Bucles simples**. A los bucles simples se les debe aplicar el siguiente conjunto de pruebas, donde *n* es el número máximo de pasos permitidos por el bucle:

1. Pasar por alto totalmente el bucle.
2. Pasar una sola vez por el bucle.
3. Pasar dos veces por el bucle.
4. Hacer *m* pasos por el bucle con $m < n$ .
5. Hacer $n - 1$, $n$ y $n + 1$ pasos por el bucle.

**Bucles anidados**. Si extendemos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumenta geométricamente a medida que aumenta el nivel de anidamiento. Esto llevará a un número impracticable de pruebas. Se sugiere un enfoque que ayuda a reducir el número de pruebas:

1. Comenzar con el bucle más interior. Establecer los demás bucles en sus valores mínimos.

2. Llevar a cabo las pruebas de bucles simples para el bucle más interior, mientras se mantienen los parámetros de iteración (p. ej.: contadores de bucles) de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.

3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores *típicos*.

4. Continuar hasta que se hayan probado todos los bucles.

### 2.2.2. Prueba de la caja negra

La prueba de la caja blanca del software se basa en el minucioso examen de los detalles procedimentales. Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el *estado del programa* en varios puntos para determinar si el estado real coincide con el esperado o afirmado.

La prueba de la caja negra intenta encontrar errores de las siguientes categorías: **(1)** funciones incorrectas o ausentes; **(2)** errores de interfaz; **(3)** errores en estructuras de datos o en accesos a bases de datos externas; **(4)** errores de rendimiento y **(5)** errores de inicialización y terminación.

Entre las técnicas de caja negra más usuales se encuentran:

**Partición equivalente**. La *partición equivalente* es un método de prueba de la caja negra que divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. Un caso de prueba ideal descubre de forma inmediata una clase de errores (p. ej.: procesamiento incorrecto de todos los datos de caracteres) que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico. La partición equivalente se dirige a la definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar.

El diseño de casos de prueba para la partición equivalente se basa en una evaluación de las clases de equivalencia para una *condición de entrada*. Una *clase de equivalencia* representa un conjunto de estados válidos o inválidos para condiciones de entrada. Típicamente, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición lógica. Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:

1. Si una condición de entrada especifica un *rango*, se define una clase de equivalencia válida y dos inválidas.

2. Si una condición de entrada requiere un *valor* específico, se define una clase de equivalencia válida y dos inválidas.

3. Si una condición de entrada especifica un miembro de un *conjunto*, se define una clase de equivalencia válida y una inválida.

4. Si una condición de entrada es *lógica*, se define una clase válida y una inválida.

**Análisis de valores límite**. Por razones que no están del todo claras, los errores tienden a darse más en los límites del campo de entrada que en el "centro". Por ello, se ha desarrollado el *análisis de valores límites (AVL)* como técnica de prueba. El análisis de valores límite nos lleva a una elección de casos de prueba que ejerciten los valores límite.

Las directrices de AVL son similares en muchos aspectos a las que proporciona la partición equivalente:

1. Si una condición de entrada especifica un *rango* delimitado por los valores *a* y *b*, se deben diseñar casos de prueba para los valores *a* y *b*, y para los valores justo por debajo y justo por encima de *a* y *b*, respectivamente.

2. Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo. También se deben probar los valores justo por encima y justo por debajo del máximo y del mínimo.

3. Aplicar las directrices 1 y 2 a las condiciones de salida. Por ejemplo, supongamos que se requiere una tabla de temperatura frente a presión como salida de un programa de análisis de ingeniería. Se deben diseñar casos de prueba que creen un informe de salida que produzca el máximo (y el mínimo) número permitido de entradas en la tabla.

4. Si las estructuras de datos internas tienen límites preestablecidos (p. ej.: un array que tenga un límite definido de 100 entradas), hay que asegurarse de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

**Prueba de comparación**. Hay situaciones (p. ej.: control de vuelo, de centrales nucleares) en las que la fiabilidad del software es algo absolutamente crítico. En ese tipo de aplicaciones, a menudo se utiliza hardware y software redundante, para minimizar la posibilidad de error. Cuando se desarrolla software redundante, varios equipos de ingeniería de software separados desarrollan versiones independientes de una aplicación, usando las mismas especificaciones. En esas situaciones, se debe probar cada versión con los mismos datos de prueba, para asegurar que todas proporcionan una salida idéntica. Luego, se ejecutan todas las versiones en paralelo y se hace una comparación en tiempo real de los resultados, para garantizar la consistencia.

## 3. DOCUMENTACIÓN DE PROGRAMAS

Una vez que se genera el código fuente, la función de un módulo debe resultar clara sin necesidad de referirse a ninguna especificación del diseño. En otras palabras, el código debe ser comprensible. El estilo de codificación conlleva una filosofía de codificación que mezcle la simplicidad con la claridad.

Los elementos a tener en cuenta a la hora de documentar los programas son la documentación interna (al nivel de código fuente), los métodos de declaración de los datos, el enfoque a la construcción de sentencias y las técnicas de E/S.

### 3.1. Documentación del código

La documentación interna del código fuente comienza con la elección de los nombres de los identificadores (variables y etiquetas), continúa con la localización y la composición de comentarios y termina con la organización visual del programa.

La elección de nombres de identificadores significativos es crucial para la legibilidad. Los lenguajes que limitan la longitud de los nombres de las variables o de las etiquetas a unos pocos caracteres, implícitamente limitan la comprensión.

Los nombres significativos *simplifican la conversión de la sintaxis del programa a la estructura semántica interna.*

Existen muchos modelos propuestos para la creación de comentarios. Los **comentarios de prólogo** y los **comentarios descriptivos** son dos categorías que requieren enfoques algo diferentes. Al principio de cada módulo debe haber un *comentario de prólogo*. El formato para esos comentarios es:

1. Una sentencia de propósito que indique la función del módulo.

2. Una descripción de la interfaz que incluya:

   * Un ejemplo de *secuencia de llamada.*
   * Una descripción de todos los argumentos.
   * Una lista de todos los módulos subordinados.

2. Una explicación de los datos pertinentes, tales como las variables importantes y su uso, de las restricciones y limitaciones y de otra información importante.

3. Una historia del desarrollo que incluya:

   * El diseñador del módulo (autor).
   * El revisor (auditor) y la fecha.
   * Fechas de modificación y descripción.

Los *comentarios descriptivos* se incluyen en el cuerpo del código fuente y se usan para describir las funciones de procesamiento.

Los comentarios descriptivos deben:

- Describir los bloques de código en lugar de comentar cada línea.
- Usar líneas en blanco o tabulaciones de forma que sean fácilmente distinguibles del código.
- Ser correctos; un comentario incorrecto o que se pueda interpretar mal es peor que no ponerlo.

### 3.2. Declaración de datos

La complejidad y la organización de las estructuras de datos se definen durante el paso de diseño. El estilo en la declaración de datos se establece cuando se genera el código.

El orden de las declaraciones de datos se debe estandarizar incluso aunque el lenguaje de programación no tenga requisitos específicos.

El orden hace que los atributos sean fáciles de descubrir, comprobar, depurar y mantener.

Si el diseño preescribe una estructura de datos compleja, se deben usar comentarios para explicar las particularidades inherentes a la implementación en el lenguaje de programación. Por ejemplo, una estructura de datos de lista enlazada en C o un tipo de datos definido por el usuario en PASCAL pueden requerir documentación suplementaria por medio de comentarios.

### 3.3. Construcción de sentencias

La construcción del flujo lógico del software se establece durante el diseño. La construcción de sentencias individuales, sin embargo, es parte del paso de codificación. La construcción de sentencias se debe basar en una regla general: cada sentencia debe ser simple y directa; el código debe ser reorganizado aunque se precise una mayor eficiencia.

Muchos lenguajes de programación permiten disponer múltiples sentencias en una misma línea. El ahorro de espacio que esto implica está difícilmente justificado por la pobre legibilidad del resultado. Considere los dos siguientes segmentos de código:

```
DO I=l TO N-1; T=I DO J=I+1 TO N; IF A(J)<A(T) THEN DO T=J;
END;
IF T < > I THEN DO H=A(T); A(T) =A(I) =T; END; END;
```

La estructura de bucles y las operaciones condicionales del anterior segmento están enmascaradas por la construcción de múltiples sentencias por línea. Reorganizando el formato del código:

```
DO I = 1 TO N - 1;
T = I;
DO J = I + 1 TO N;
    IF A(J) < A(T) THEN DO
    T = J;
    END;
IF T <> I THEN DO
    H = A(T);
    A(T) = A(I);
    A(I) = T;
    END;
END;
```

Aquí la sencilla construcción de las sentencias y el sangrado dan luz sobre las características lógicas y funcionales del segmento. Las sentencias de código fuente individuales se pueden simplificar al:

Evitar el uso de complicadas comparaciones condicionales.
Eliminar las comparaciones con condiciones negativas.
Evitar un gran anidamiento de bucles o de condiciones.
Usar paréntesis para clarificar las expresiones lógicas o aritméticas.
Usar espacios y/o símbolos claros para incrementar la legibilidad del contenido de la sentencia.
Pensar: *¿Podría entender yo esto si no fuera la persona que lo codificó?*.

### 3.4. Entrada/salida

El estilo de la entrada y la salida variará con el grado de interacción humana. Para una E/S orientada a lotes serán deseables características, tales como una organización lógica de la entrada, una comprobación de errores de entrada/salida significativa, una buena recuperación de errores de E/S y unos formatos de informes de salida racionales. Para la E/S interactiva lo principal será un esquema de entrada simple y dirigido, una extensa comprobación y recuperación de errores, una salida humanizada y una consistencia de formato de E/S.

Dejando a un lado la naturaleza interactiva o no del software, se deben considerar una serie de principios de estilo para la E/S durante el diseño y la codificación:

Validar todos los datos de entrada.

Comprobar las importantes combinaciones plausibles de elementos de entrada.

Mantener el formato de entrada simple.

Usar indicativos de fin de dato, en lugar de requerir al usuario que especifique el *número de elementos.*

Etiquetar las peticiones interactivas de entrada, especificando las opciones posibles o los valores límite.

Mantener el formato de entrada uniforme cuando un lenguaje de programación tenga estrictos requisitos de formato.

Etiquetar todas las salidas y diseñar todos los informes.

El estilo de la E/S se ve afectado por otras muchas características, tales como los dispositivos de E/S (p. ej.: tipo de terminal, dispositivo de gráficos, ratón, etc.), la sofisticación del usuario y el entorno de comunicación.

## 4. BIBLIOGRAFÍA

Pressman, R.S.
**Ingeniería del software**
Mc Graw-Hill

Gregorio Fernández
**Fundamentos de Informática**
Anaya Multimedia, 1995