Capítulo 4

TÉCNICAS DE VERIFICACIÓN Y PRUEBAS

4.1. Introducción

En el desarrollo del software las posibilidades de error son innumerables. Los errores pueden darse desde una mala especificación de los requisitos funcionales, una incorrecta selección de los métodos de resolución, uso indebido de las estructuras de datos, errores al enlazar módulos, ...

El desarrollo del software ha de ir acompañado de alguna actividad que garantice la *calidad*, la *prueba* es un elemento crítico para la garantía de calidad del software. La importancia de los costes asociados a los fallos motivan la creación de un proceso de pruebas minuciosas y bien planificadas.

Gran parte del esfuerzo que se dedica al desarrollo del software se invierte en la prueba (valor aproximado 40 %), e incluso en el caso de software para actividades críticas, como por ejemplo los sistemas de control en tiempo real, puede invertirse en la prueba un esfuerzo muy superior al que se invierte en el resto de etapas.

La prueba y validación de los resultados no es un proceso que se realiza una vez desarrollado el software sino que debe efectuarse en cada una de las etapas de desarrollo.

En la **especificación de requisitos** el tener una explicación clara, precisa y completa del problema facilita el análisis de errores y la generación de casos de prueba. Hay que generar los datos necesarios para determinar si se han cubierto todos los requisitos y determinar en base a éstos los valores esperados de los casos de prueba. Es importante asegurar la corrección, coherencia y exactitud de los requisitos.

En el **diseño** hay que comprobar los algoritmos individuales, las interfaces entre módulos y analizar las estructuras de datos para localizar posibles inconsistencias o torpezas en su construcción. Se analiza globalmente el diseño para determinar si satisface los requisitos y se diseñan datos de prueba basados en esta etapa.

En la **codificación** hay que comprobar la coherencia con el diseño, se ejecutan los casos de prueba, conservando la información referente al proceso de prueba. Se fuerza al máximo la consistencia de la estructura del programa, las estructuras de datos y su funcionalidad.

Una buena técnica consiste en que la prueba la realice un equipo diferente del que realizó hasta entonces todo el proceso de desarrollo, con el fin de estudiar la lógica del programa y buscar errores que se hayan escapado previamente.

Es fundamental medir la *cobertura de las pruebas*, es decir la determinación de cuando se han realizado las suficientes pruebas. Si se siguen encontrando errores cada vez que se procesa el programa, las pruebas deben continuar.

Durante el **mantenimiento** debe de existir documentación de pruebas que incluya casos de prueba y resultados esperados. Si se producen modificaciones en el programa, habrá que probar de nuevo todas las partes del programa afectadas por las modificaciones.

4.2. Fundamentos de la prueba del software

4.2.1. Objetivos de la prueba (reglas).

- La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
- Un buen caso de prueba es aquel que tiene una alta probabilidad de descubrir un error no descubierto hasta entonces.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Hay un cambio en el punto de vista. Anteriormente la prueba tenía éxito si no descubría errores, ahora lo que se pretende es diseñar pruebas que sistemáticamente saquen a la luz los errores con la mínima cantidad de tiempo y esfuerzo.

También la prueba servirá para demostrar hasta que punto el software se ajusta a las especificaciones funcionales y a los requerimientos de rendimiento para los cuales se diseñó. Los datos obtenidos durante el transcurso de la prueba proporcionan una medida de la *fiabilidad* del software y por tanto de la *calidad* del mismo. Pero hay que tener en cuenta que la prueba no puede asegurar la ausencia de errores, solo puede demostrar que existen defectos en el software.

4.2.2. Flujo de información de la prueba

Se proporcionan dos clases de entrada al proceso de prueba :

1.) Configuración del software: * Especificación de requerimientos.
* Documento de Diseño.
* Código fuente.

2.) Configuración de prueba: * Plan y procedimiento de prueba.
* Casos de prueba.
* Resultados esperados.

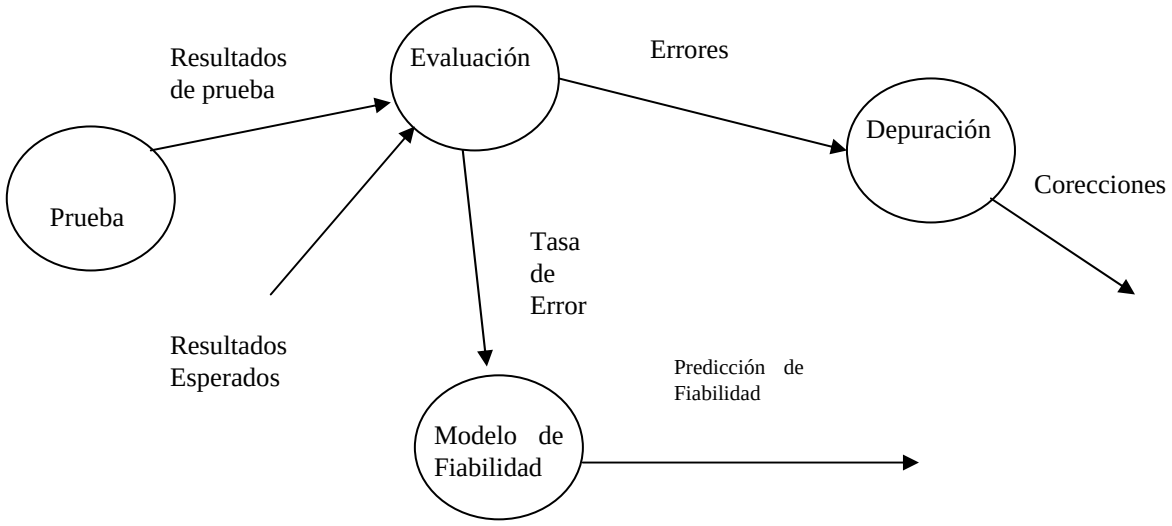
Se lleva a cabo la prueba y se evalúan los resultados obtenidos frente a los resultados esperados. Si se descubren datos erróneos implica que hay un error y hay que corregirlo y empieza el proceso de depuración de errores.

A medida que se van obteniendo los resultados de la prueba se empieza a disponer de una medida cualitativa de la calidad y fiabilidad del software. Las situaciones posibles que pueden aparecer son:

1.) Se encuentran con regularidad serios errores que requieren modificación en el diseño. La calidad y fiabilidad no parecen ser idóneas.

2.) El funcionamiento del software parece ser correcto y los errores que se detectan son fácilmente corregibles. En ese caso puede suceder que:
   - ✓ La calidad y fiabilidad del software sean aceptables.
   - ✓ Las pruebas son inadecuadas para descubrir serios errores.

3.) La prueba no descubre errores. Puede darse el caso de que no se ha llevado a cabo una prueba correcta y los errores siguen presentes en el software.



4.2.3. Diseño de casos de prueba

Se deben de diseñar métodos de prueba que tengan la mayor probabilidad de encontrar el mayor numero de errores con la mínima cantidad de esfuerzo y tiempo. Existen gran cantidad de métodos de diseño de casos de prueba que pretenden garantizar la obtención de un producto correcto. Se clasifican en:

- **MÉTODOS DE CAJA NEGRA :** se llevan a cabo sobre la interfaz del software. Los casos de prueba pretenden demostrar que las funciones del software se verifican , que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene.
- **MÉTODOS DE CAJA BLANCA :** se basan en un examen minucioso de los detalles procedimentales para comprobar los diferentes caminos lógicos del software, través de casos de prueba que los recorren.

Puede parecer que con una prueba de caja blanca profunda se obtendrían programas totalmente correctos ya que se diseñan casos de prueba para recorrer todos los caminos lógicos. Pero hay un problema para poder realizar esta prueba exhaustiva, el número de caminos lógicos de un programa suele ser enorme.

La solución estriba en elegir y recorrer una serie de importantes caminos lógicos y probar las estructuras de datos más importantes.

Lo normal es combinar las pruebas de caja blanca con pruebas de caja negra para garantizar la corrección del software.

4.3. Pruebas de caja blanca

Es una metodología de prueba que se basa en las estructuras de control del diseño procedimental para generar los casos de prueba que:

- ❖ Garanticen que se recorren por lo menos una vez todos los caminos independientes de cada módulo.
- ❖ Se ejecutan todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- ❖ Se recorren todos los bucles.
- ❖ Se utilizan las estructuras de datos internas para garantizar su validez.

Se invierte tiempo y esfuerzo en los detalles de control debido a que:

- ➢ Los errores suelen estar en situaciones fuera de las normales.
- ➢ A menudo caminos que pensamos que tienen pocas posibilidades de recorrerse, son recorridos regularmente.
- ➢ Los errores tipográficos son aleatorios. Puede que no sean detectados por los procesadores de la sintaxis del lenguaje particular y estar presentes en cualquier camino lógico.

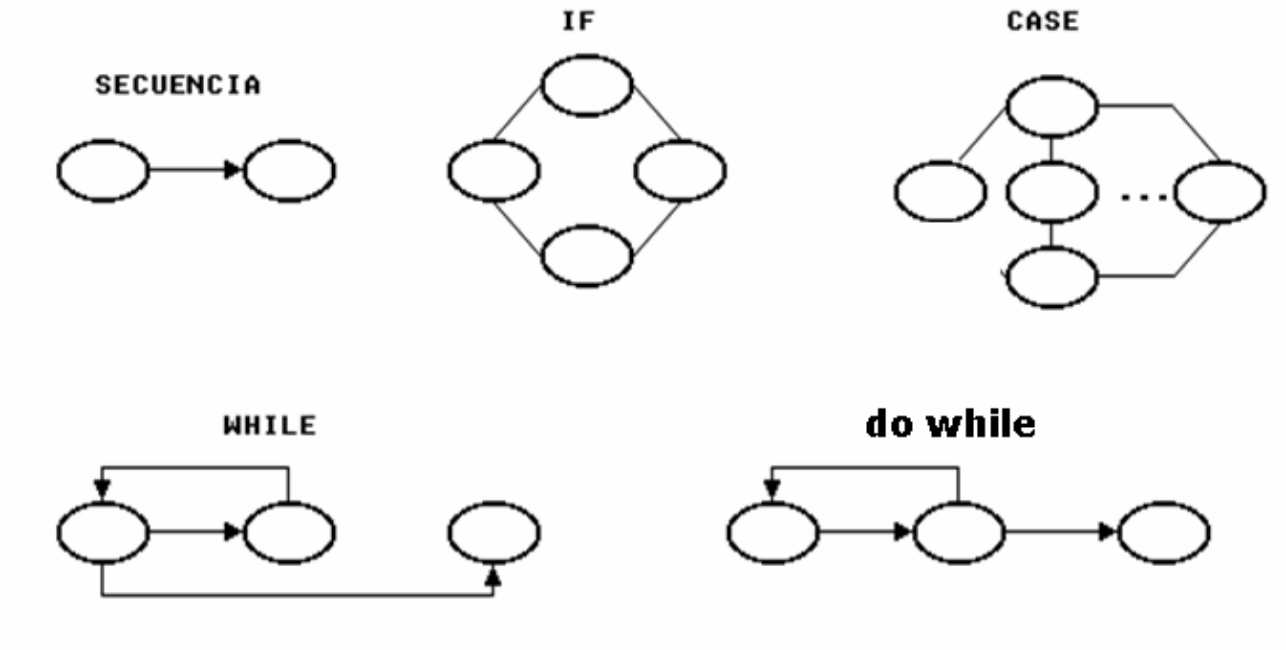
4.3.1. Prueba del camino básico

Permite obtener una medida de la complejidad lógica del diseño procedimental y usarla para definir un *conjunto básico* de caminos de ejecución que garanticen que cada sentencia del programa se recorre al menos una vez.

4.3.1.1. Grafo de flujo

Sirve para representar el flujo de control lógico. Se basa en la representación del código a probar a través de círculos y arcos que indican el flujo de control. Cada círculo representa una o más sentencias.

Las estructuras en forma de grafo de flujo son:



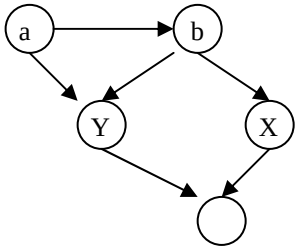
Un grafo de flujo está compuesto por :

- *Nodos,* una o más sentencias.
- *Aristas,* representan el flujo de control. Una arista debe terminar en un nodo.
- *Regiones,* áreas delimitadas por aristas y nodos.

Cualquier representación del diseño procedimental se puede traducir en un grafo de flujo, partiendo de la descripción del código a probar en forma de un diagrama de flujo, en pseudocódigo o incluso en un determinado lenguaje de programación.

Cuando existen *condiciones compuestas* se crea un nodo por cada condición simple y se diseña el grafo de flujo de acuerdo a la lógica de control de la condición compuesta. Si por ejemplo existen dos condiciones simples ligadas por el operador *and*, se refleja en el grafo de flujo el hecho de que si una de las condiciones es falsa se ejecutará la siguiente instrucción.

Cada nodo que contiene una condición se denomina **nodo predicado** y se caracteriza porque de él salen dos o más aristas. Una de las características para comprobar si un grafo de flujo esta bien diseñado es que los únicos nodos de los que pueden partir dos aristas son los nodos predicados.




```
if (a && b)
        X
else Y
```

nodos predicado:
a y b

4.3.1.2. Complejidad ciclomática

Es una medida cuantitativa de la complejidad lógica de un programa. Define el número de *caminos independientes* del *conjunto básico* de un módulo y por tanto nos proporciona una cota

superior del número máximo de pruebas que se han de realizar para garantizar que cada sentencia se ejecuta al menos una vez.

Un *camino independiente* es cualquier camino del programa que incorpora frente a otros caminos un nuevo conjunto de sentencias o una nueva condición, es decir, ha de recorrer al menos una arista no recorrida hasta entonces.

Un conjunto de caminos componen un *conjunto básico* de caminos si al recorrerlos se recorre al menos una vez cada sentencia del programa y cada condición por tanto se habrá ejecutado en su parte verdadera y falsa. Un conjunto básico de caminos no es único.

El número de caminos que componen un conjunto básico de caminos viene determinado por la complejidad ciclomática, que se calcula como:

1. El número de regiones del grafo de flujo.
2. C(G) = A-N+2 — C(G):complejidad ciclomática, G: grafo de flujo.
   A: número de aristas.
   N: número de nodos.
3. C(G) = P + 1 — P: número de nodos predicado.

4.3.1.3. Generación de casos de prueba

1. Diseñar el grafo de flujo.
2. Calcular la complejidad ciclomática del grafo obtenido.
3. Determinar un conjunto básico de caminos linealmente independientes.
4. Obtener los casos de prueba que permitan la ejecución de cada camino básico
5. Ejecutar cada paso de prueba y comparar con los resultados esperados.

Ejemplo: procedimiento para mezclar dos arrays.

```
        int [] Intercalar( int [] a1, int [] a2)
        {
        int i = 0;
1       int j = 0;
        int k = 0;
        int [] a3 = new int [a1.length + a2.length];
2,3     while (( i < a1.length ) && ( j < a2.length ) )
        {
4               if (a1[i] < a2[j])
5                       a3[k++] = a1[i++];
6               else  a3[k++] = a2[j++];
        }
7       if  (j == a2.length)        //¿El 2º array no tiene datos?
8               while (i < a1.length)
9                       a3[k++] = a1[i++];
10      else    while (j < a2.length)
11                      a3[k++] = a2[j++];
12      return a3;
        }
```

***CASOS DE PRUEBA**:* Complejidad ciclomática = 7

camino 1: 1-2-7-8-12 Los dos arrays no tienen elementos.

RESULTADOS ESPERADOS: *No hacer nada, a3 vacio.*

camino 2 : 1-2-3-7-8-12 a1 con datos, a2 sin datos. *Camino imposible.*

{ Se tendrían que verificar a la vez: por (2) i< a1.length es verdad y por (8) i < a1.length es falso}

Solución del camino 2 que entre en el bucle.

camino 3 : 1-2-3-7-8-9-8-12 a1 con datos y a2 sin datos.

RESULTADOS ESPERADOS: *Se introduce el elemento del array 1 en el array de Salida.*

camino 4 : 1-2-3-4-5-2-7-8-9-8-12 a1 con datos y a2 con datos. a1[i] < a2[j]. *Camino imposible.*

{ Se tendrían que verificar a la vez: por (3) j< a2.length es verdad y por (7) j == a2.length es verdad

Además, también se verifica: (2) i< a1.length es falso y por (8) i < a1.length es verdad}

Solución del camino 4, no ir por 8 e ir por 10, selecciono el camino sencillo.

camino 5 : 1-2-3-4-5-2-7-10-12 a1 con datos y a2 con datos. a1[i] < a2[j]. *Camino imposible.*

{ pues se tendrían que verificar simultáneamente: por (3) j < a2.length es verdad y por (10) j < a2.length es falso}

Solución del camino 5 que entre en el bucle.

camino 6 : 1-2-3-4-5-2-7-10-11-10-12 a1 con datos, a2 con datos. a1[i] < a2[j].

RESULTADOS ESPERADOS: *Se introducen en el array de Salida a3 el elemento del primer array y después el del segundo.*

Queda por analizar la arista de 6 a 2, 1-2-3-4-6-2 ahora hay dos opciones ir a 7 o ir a 3, con el análisis de los caminos anteriores determino que debo ir a 3 (el último elemento introducido es de a2, así que en a1 todavía quedan elementos)

camino 7 : 1-2-3-4-6-2-3-7-8-9-8-12 a1 con datos, a2 con datos. a1[i] > a2[j].

RESULTADOS ESPERADOS: *Se introducen en el array de Salida a3 el elemento del segundo array y después el elemento del primero.*

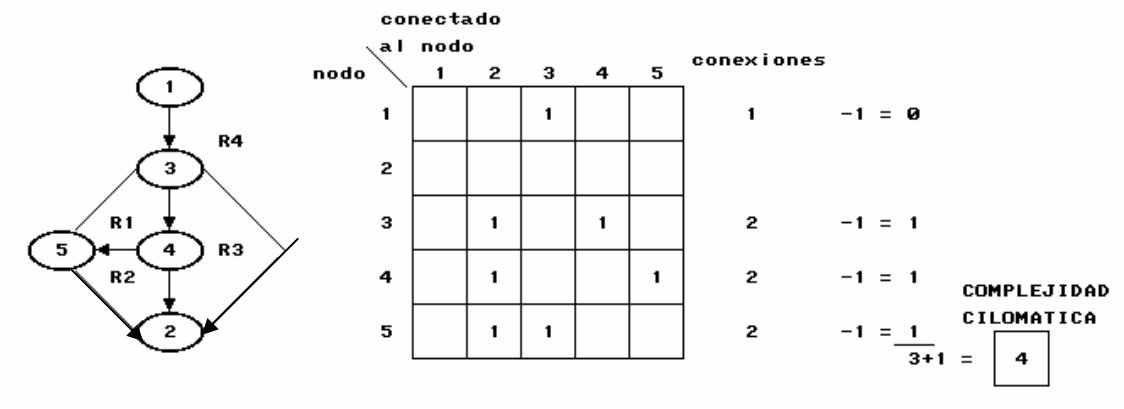
Hay caminos que no se pueden probar de forma aislada. Es decir, la combinación de datos que se necesita para recorrer el camino no se puede lograr con el flujo normal del programa y, por lo tanto, se prueban como parte de la prueba de otro camino.

4.3.1.4. Matriz del grafo

El mecanismo de prueba del camino básico es susceptible de ser automatizado. A tal efecto se utiliza una representación del grafo de flujo consistente en una *matriz de grafo.* Es una matriz cuadrada cuya longitud es el número de nodos del grafo de flujo. Cada fila y columna se corresponden a un nodo especifico y el contenido de la matriz son las aristas del grafo, estas pueden llevar un *peso de enlace* que da información sobre el flujo de control. Puede ser 0 ó 1, indicando si existe conexión entre los nodos o se le puede asociar a cada arista propiedades más interesantes como:

- Probabilidad de que la arista se recorra.
- Tiempo invertido en recorrer la arista.
- Recursos requeridos durante el recorrido (memoria,...).

Ejemplo*:* Si la matriz del grafo contiene 0 ó 1 indicando si existe o no conexión entre los nodos, a la matriz resultante se la denomina *matriz de conexiones.* A partir de la matriz se puede calcular la complejidad ciclomática y el conjunto básico de caminos.



4.3.2. Prueba de bucles

Los bucles junto a las decisiones son fundamentales en el desarrollo algorítmico, por lo que es importante el aplicar técnicas que verifiquen la corrección de los mismos. La prueba de bucles es una técnica de prueba de Caja Blanca que se centra en la validez de las construcciones de bucles. Se pueden definir cuatro clases:

BUCLES SIMPLES: Se les debe aplicar las siguientes pruebas :

1. Saltar el bucle.
2. Pasar una vez.
3. Pasar dos veces.
4. Pasar **j** veces con **j<n**, siendo **n** el número máximo de pasos permitidos.
5. Pasar **n-1, n, n+1** veces por el bucle.

BUCLES ANIDADOS: Si intentamos generalizar el proceso de prueba de bucles simples a un conjunto de bucles anidados, el número de posibles pruebas podría llegar a ser impracticable. Existe una metodología para aplicar eficientemente esta prueba :

1. Comenzar con el bucle más interior. El resto se dejan en valores mínimos.
2. Se aplican las pruebas de bucles simples al bucle más interior, mientras se mantienen los bucles exteriores en sus valores mínimos. Añadir otras pruebas para los valores fuera de rango o para valores excluidos.
3. Progresar hacia fuera, llevando a cabo las pruebas para el siguiente bucle manteniendo los bucles exteriores en sus valores mínimos y los interiores en sus valores típicos.
4. Repetir el proceso hasta que todos los bucles hayan sido probados.

BUCLES CONCATENADOS: Se pueden probar igual que los bucles simples si son independientes entre sí y siguiendo la técnica de prueba de bucles anidados si existe cualquier relación entre los diferentes bucles concatenados. Por ejemplo, si hay dos bucles concatenados y se usa el contador del bucle 1 como valor inicial del bucle 2, entonces los bucles no son independientes; luego en este caso, se aplicará el enfoque para bucles anidados.

BUCLES NO ESTRUCTURADOS : Este tipo de bucles son construcciones no deseables y lo que se debe de intentar en la medida de lo posible es el rediseñarlos para que se ajusten a las reglas de la programación estructurada.

4.4. Pruebas de caja negra

Se centran en probar que se verifican los requerimientos funcionales del software, es decir, que las funciones para las cuales se diseñó se cumplen sin errores. O sea, la prueba de Caja Negra permite al ingeniero del software derivar conjuntos de condiciones de entrada que ejerciten completamente todos los requerimientos funcionales de un programa.

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 (por ejemplo archivos de datos) se mantiene.

Una prueba de Caja Negra examina algunos aspectos del modelo fundamental del sistema sin tener mucho en cuenta la estructura lógica interna del software.

No se utilizan frente a las pruebas de Caja Blanca, sino que constituyen un conjunto de técnicas que tratan de detectar errores de diferente tipo. Se centran sobre el dominio de información frente a los detalles de control. Los errores que se pretenden detectar mediante las pruebas de caja negra son:

- Funciones incorrectas o ausentes.
- Errores de interfaz.
- Errores en las estructuras de datos.
- Errores de rendimiento.
- Errores de inicialización o terminación.

Las ventajas que presentan este tipo de métodos de prueba son dos:

- Reducen el número de casos de prueba que se deben diseñar.
- Detectan clases de errores en vez de errores simples.

4.4.1. Método de la partición equivalente

Consiste en dividir el dominio de entrada de un programa en *clases de datos,* de los que se pueden generar casos de prueba.

Un buen caso de prueba descubrirá una *clase de errores* que de otra forma requerirían la ejecución de muchos casos antes de detectar el error.

Se debe intentar dividir el dominio de entrada en un número finito de clases de equivalencia. Tal que probar un valor representativo de cada clase es equivalente a probar cualquier otro valor, es decir :

- Si un caso de prueba es tal que en una clase de equivalencia detecta un error, todos los demás posibles casos de prueba en la misma clase detectarán el mismo error.
- Si un caso de prueba no detecta error, podemos pensar que ningún otro caso de prueba en esa clase encontrará el error.

4.4.1.1 Diseño de casos de prueba

PASO 1 : IDENTIFICACIÓN DE LAS CLASES DE EQUIVALENCIA.

Se identifican dividiendo en dos o más grupos (clases) cada *condición de entrada*, la cual es :

- Un valor numérico específico.
- Un rango de valores.
- Un conjunto de valores relacionados.
- Una condición booleana.

Las clases de equivalencia se pueden definir de acuerdo a las siguientes directrices:

- Si una condición de entrada especifica un **rango** de valores (por ejemplo : valor entre 1 y 999), se definen una clase de equivalencia válida (1<=valor<=999) y dos inválidas (valor<1 y valor>999).
- Si una condición de entrada requiere un **valor** específico, (el tamaño de un código es de 5 dígitos), se definen una clase de equivalencia válida (tamaño del código 5) y dos inválidas (tamaño<5 y tamaño>5).
- Si una condición de entrada especifica un **conjunto** de valores de entrada (tipo de vehículo: AUTOBÚS, TAXI, TURISMO, MOTOCICLETA), se define una clase de equivalencia válida para cada valor y una clase de equivalencia inválida (por ejemplo PATINES).

- Si una condición de entrada es **booleana** (el primer carácter de un identificador debe ser una letra), se definen una clase válida (es letra) y una inválida (no es letra).

PASO 2 : IDENTIFICACIÓN DE LOS CASOS DE PRUEBA.

- Asignar un número único a cada clase de equivalencia.
- Escribir casos de prueba hasta que sean cubiertas todas las clases de equivalencia válidas, intentando cubrir en cada caso tantas clases de equivalencia válidas como sea posible.
- Escribir casos de prueba hasta que sean cubiertas todas las clases de equivalencia inválidas cubriendo en cada caso una y sólo una clase de equivalencia aún no cubierta. La razón por la que las clases de equivalencia inválidas se cubren de forma individual se debe a que al detectar una entrada errónea seguramente no se chequea la siguiente.

Ejemplo: DATOS CONTENIDOS EN UNA APLICACIÓN BANCARIA.

El software proporcionado por la aplicación bancaria acepta datos en la siguiente forma:

* *Código de área* : un número de tres dígitos.

* *Prefijo*: número de tres dígitos que no comience por 0 ó 1.

* *Sufijo* : número de cuatro dígitos.

* *palabra* clave : Valor alfanumérico de 6 dígitos.

* *Ordenes* : CHEQUE, DEPOSITO, PAGO.

Paso 1: Identificación de las condiciones de entrada y localización de las clases de equivalencia. Las condiciones de entrada son :

* Código de área : - Booleana : es un número.
- Valor : el número de dígitos es 3.

* Prefijo: - Booleana : es un número.
- Valor : el número de dígitos es 3.
- Rango : >200

* Sufijo : - Booleana : es un número.
- Valor : el número de dígitos es 4.

* Palabra clave : - Booleana : es alfanumérico.
- Valor : longitud es 6.

* Ordenes : - Conjunto : valores pertinentes.

CONDICIONES DE ENTRADA	**C.E. VALIDAS**	**C.E.INVÁLIDAS**
Código de área es	1. un número	2. otro valor
Longitud de código de área	3. tres	4.<3 5.>3
Prefijo es	6. un número	7. otro valor
Longitud de prefijo es	8. tres	9.<3 10.>3
Primer dígito de prefijo	11.>1	12. 0 13. 1
Sufijo es	14. un número	15.otro valor
Longitud de sufijo es	16. cuatro	17.<4 18.>4
Palabra clave es	19.alfanumérico	20.otro valor
Longitud de p.c. es	21. seis	22.<6 23.>6
Ordenes es	24.CHEQUE 25.DEPOSITO 26.PAGO	27.otro valor

Paso 2 : Identificación de los casos de prueba.

CLASES VALIDAS

cod.	prefijo	sufijo	palabra	ordenes	clases
234	967	9673	OR-234	CHEQUE	1,3,6,8,11, 14,16,19,21,24
856	956	5672	GR-527	DEPOSITO	...,25
623	723	2782	CR-244	PAGO	...,26

CLASES INVÁLIDAS

cod.	prefijo	sufijo	palabra	ordenes	clases
&D-	967	9673	OR-234	CHEQUE	2
7	123	5672	GR-527	DEPOSITO	4
1623	723	2782	CR-244	PAGO	5
.....					...

Ejemplo : TRATAMIENTO DE NOTAS: Se supone una aplicación de procesamiento de notas que acepta la siguiente información :

*** Código :** valor alfanumérico de longitud 6, donde los dos primeros caracteres son dos letras y el resto dígitos.

*** Tres notas :** tres valores enteros de 0 a 100.

CONDICIONES DE ENTRADA	C. E. VALIDAS	C.E.INVALIDAS
Primer carácter de código	1. una letra	2. otro valor
Segundo carácter de código	3. una letra	4. otro valor
Longitud de código es	5. seis	6.<6 7.>6
Caracteres 3-6 de código	8. números	9.otro valor
nota1 es	10. un entero	11.otro valor
el rango de nota1 es	12.>=0 y <=100	13.<0 14.>100
nota2 es	15. entero	16.otro valor
el rango de nota2 es	17.>=0 y <=100	18.<0 19.>100
nota3 es	20. entero	21.otro valor
el rango de nota3 es	22.>=0 y <=100	23.<0 24.>100

CLASES VÁLIDAS :

código	nota1	nota2	nota3	clases
in2349	45	74	89	1-3-5-8-10-12-15-17-20-22

CLASES INVÁLIDAS :

7n2342	34	53	12	2
a¿4323	78	93	23	4
in345	54	67	73	6
ma43432	76	23	4	7
ma3ñ21	45	5	89	9

.....

4.4.2. Análisis de valores límite

Los errores tienden a darse más en los límites del dominio de datos que en el interior. Es por

tanto natural el desarrollar un método de prueba centrada en estos valores límites que conducen a un mayor número de errores.

El análisis de valores límite se centra en generar casos de prueba para probar la corrección de las *condiciones límites*, que son aquellas situaciones que se dan cuando se introducen valores que están justo en el límite de las clases de equivalencia del dominio tanto de entrada como de salida. Es un método que complementa a la partición equivalente y que presenta notables diferencias con éste:

1. No se selecciona un elemento representativo de una clase de equivalencia sino uno o más, tal que se prueben los extremos de dicha clase.
2. Se presta atención no sólo al domino de entrada sino también al dominio de salida.

Una ventaja considerable de este método es que presenta una mayor probabilidad de detectar errores no detectados hasta entonces.

4.4.2.1 Criterios para la derivación de casos de prueba

- Si una condición de entrada especifica un **rango** de valores limitado por a y b, se deben escribir casos de prueba para los límites del rango, es decir, a y b, y casos de prueba inválidos para situaciones fuera de los límites. Ej: si el rango válido de un valor de entrada es -1.0 a +1.0, escribir casos de prueba para las situaciones: -1.0, +1.0, -1.0001, +1.0001
- Si una condición de entrada especifica un número de valores, desarrollar casos de prueba para el número máximo y mínimo de valores y otros casos para los valores justo por encima y justo por debajo del máximo y del mínimo. Ej: si el número de alumnos para calcular su nota final es de 125, escribir casos de prueba para 0, 1, 125 y 126 alumnos.
- Aplicar el primer criterio para cada condición de salida. Por ejemplo, si un programa calcula la deducción fiscal mensual, siendo el rango permitido entre 0 y 1000 €. escribir casos de prueba que proporcionen el resultado de 0 € y 1000 €. Comprobar si es posible diseñar casos de prueba que puedan proporcionar una deducción negativa o mayor de 1000 €.
- Es importante examinar los límites del espacio de resultados pues no siempre ocurre que los límites de los dominios de entrada representan el mismo conjunto de circunstancias que los límites de los rangos de salida. (por ejemplo la función **seno**).
- Usar el segundo criterio para cada condición de salida. Por ejemplo, si el número máximo de errores de tipo léxico que detecta un analizador léxico es de 10, escribir casos de prueba para que el analizador no saque ningún error, un error, diez errores, e intentar que el analizador produzca once errores.
- Si las estructuras de datos internas tienen límites preestablecidos, hay que asegurarse de diseñar casos de prueba que ejerciten la estructura de datos en sus límites. Por ejemplo, un array que tenga un límite definido de 100 elementos, habrá que probar el hecho de que no existan elementos y el de que se procesen bien 100 elementos.
- Usar el ingenio para buscar otras condiciones límites.

Ejemplo: CÁLCULO DE NÓMINAS.

Supongamos una aplicación de cálculo de la nómina de una empresa de 132 empleados. La información necesaria para calcular el salario de cada empleado es :

* *Código* : entero de 1 a 132.

* *Puesto* : alfanumérico de hasta 4 caracteres.

* *Antigüedad* : real de 0 a 25 años.

* *Horas semanales* : 0 a 60.

La información que se produce es :

* *Salario bruto:* 0 a 200.000.

* *Deducción*: 0 a 50%.

* *Salario neto*: 0 a 150.000.

	C.P. VALIDAS	**C.P. INVALIDAS**
Código sea	1.) 1 2.)132	3.)0 4.)133
Puesto tenga	5.) 1 6.)4	7.)0 8.)5
Antigüedad	9.)0 10.)25	11.)-0.1 12 .)25.1
Horas semanales	13.)0 14.)60	15.)-1 16.)61
Salario bruto	17.)0 18.)200.000	19.)-1 20.)200.001
Deducción	21.)0 % 22.)50%	23.)-1% 24.)51%
Salario neto	25.)0 26.)150.000	27.)-1 28.)150.001

4.5. **Verificación del comportamiento de los objetos.**

La fase de prueba es una parte de vital importancia en el proceso de desarrollo de software. Es subestimada a menudo por los estudiantes quienes, como futuros programadores que son, poseerán la responsabilidad de crear código correcto. *Las pruebas no son un elemento del que hay que preocuparse sólo en una fase posterior al desarrollo del programa.* Por el contrario, las pruebas son una parte integral del proceso de desarrollo de programas, a la que se le debe dedicar tiempo y esfuerzo.

Las pruebas representan un intento de minimizar errores en nuestros programas. Sin embargo, como dijo una vez uno de los programadores más famosos, Edsgar Dijkstra, “las pruebas sólo confirman la existencia de errores, nunca su ausencia”. Esto es, si durante las pruebas descubrimos errores, determinamos su causa y los arreglamos. Sin embargo, aunque no encontremos errores durante la fase de pruebas, no hay garantía de que el código no tiene errores; nuestra única certeza es que nuestros tests no los han descubierto.

Las pruebas son algo más que ejecutar el programa utilizando un número de casos arbitrariamente escogidos. Requiere un estudio concienzudo y un conocimiento tanto de la aplicación como de la solución software. Con unas pruebas mal diseñadas, los errores más obvios pueden permanecer en los programas (y de hecho lo harán).

Aunque los errores son de muchos tipos, se pueden clasificar en tres categorías básicas. Los errores que ocurren durante la traducción del código fuente (es decir, durante la compilación) se llaman *errores de compilación*. Los errores que causan que un programa termine de forma inesperada sin terminar la tarea para la que fueron diseñados se llaman *errores en tiempo de ejecución* (*run-time errors*). En un programa Java, esto normalmente va unido al anuncio de una “excepción” seguida de un torrente de mensajes de error. Finalmente, existen errores que no causan ningún error explícito, aunque hacen que el programa produzca un resultado incorrecto. Este tipo de error se conoce como *error de lógica interna*. Los errores en tiempo de ejecución y los de lógica interna se conocen a menudo como *bugs*.

4.5.1. Métodos de prueba.

Una aplicación como la que se describe a continuación se debe probar introduciendo distintos valores y comprobando los resultados obtenidos para las distintas entradas con los resultados esperados. ¿Pero qué ocurre con clases accesorias?, es decir, clases que existen principalmente para ser utilizadas por otras clases. No se pueden ejecutar hasta que no forman parte de un programa mayor. Todo código, incluso las clases accesorias, se debe probar concienzudamente antes de usarlo. Surge la cuestión de cómo probar una clase cuya utilidad está supeditada a ser incluida en programas más grandes. La respuesta es crear un método de prueba para la clase, esto es, un método cuyo único propósito es probar la validez de la clase. Por conveniencia, este método se puede incorporar a la clase a la que se va a probar.

Si tenemos el siguiente problema: Modele un sistema de nóminas que contemple a empleados que ganan un sueldo por horas. El sistema debe ser capaz de calcular la nómina de un empleado basándose en la paga por hora de trabajo y las horas trabajadas e imprimir el nombre del empleado, horas trabajadas y la nómina. Los empleados que trabajen más de 40 horas reciben, una paga extra de 1.5 veces la paga normal por hora de trabajo. Para evitar el abuso de horas extra, si un empleado ha trabajado 30 o más horas extra en las dos semanas anteriores, se muestra un mensaje de aviso.

```
class Empleado{
   //Métodos
   public Empleado (String nombre, int pagaHora)
   {
      this.nombre = nombre;
      this.pagaHora = pagaHora;
      //Un empleado recien creado su numero de horas extra es 0
      this.horasExtraSemanaPasada = 0;
   }
   public int calculaNomina (int horas)
   {
      int nomina;
      int horasExtraActuales;
      if (horas <= 40){     //Comprobación de horas Extra
         //nomina sin pagas extra
         nomina = horas*pagaHora;
         horasExtraActuales = 0;
      }
      else {
         //las primeras 40 horas se pagan a precio normal
         //el resto se pagan 1.5 veces más caras
         //nomina con pagas extra
         nomina = (40*pagaHora)+(horas-40)*(pagaHora+pagaHora/2);
         horasExtraActuales = horas-40;
      }
      /*Se imprime un aviso si la suma de las horas extra actuales y
      las de la semana pasada es mayor o igual a 30*/
      if (horasExtraActuales+horasExtraSemanaPasada >=30)
      {
         System.out.print(nombre);
         System.out.print( "ha hecho 30 horas extra en las dos ultimas semanas");
      }
      /*El número de horas extra de esta semana se convierte en horas
      extra de la semana anterior para el siguiente periodo de pago*/
      horasExtraSemanaPasada = horasExtraActuales;
      return nomina;
   }
   public String getNombre(){
      return this.nombre;
   }
   //Variables de instancia
   private String nombre;
   private int pagaHora;
```

```
   private int horasExtraSemanaPasada;
}
class Nomina{
   public static void main (String a[]){
      Empleado e;
      e = new Empleado("Maria Gomez",10);
      //NOmbre, paga hora
      int nomina;
      nomina = e.calculaNomina(30);
      System.out.println(e.getNombre()+" ha ganado "+nomina);
   }}
```

La clase Empleado podría incluir su propio método de prueba. Dicho método es static ya que no se invoca con ningún objeto. Un método de prueba podría ser el siguiente:

```
class Empleado
....
public static void metodoPrueba() {

        Empleado e = new Empleado (“Maria Gomez “, 25);
        int horas = 36;
        System.out.print(“Nombre del empleado: “);
        System.out.println(e.getNombre());
        System.out.print(“Paga por hora : “);
        System.out.println(e.pagaHora);
        System.out.print(“Horas: “);
        System.out.println(horas);
        System.out.print(“Paga: “);
        System.out.println(e.calculaNomina(horas));
}}
```

Así, un programa que prueba la clase Empleado es simplemente:

```
class PruebaEmpleado{

public static void main (String [] a){
        Empleado.metodoPrueba();
}}
```

El método de prueba simplemente invoca métodos de la clase e imprime los resultados; el usuario de la clase debe comprobar si los resultados son correctos.

Ventajas de los métodos de prueba.

Escribir métodos de prueba nos ayuda en los siguientes aspectos:

- Un método de prueba automatiza (al menos de forma parcial) las pruebas de una clase.
- Un método de prueba elimina la necesidad de recordar con qué valores se probó la clase. Esto hace más fácil volver a probar la clase después de corregir algún fallo, en el caso de que la corrección también hubiera fallado en algo.
- Una clase con un método de prueba beneficia a otros programadores que la usan ya que si

los programadores descubren fallos en sus programas, pueden ejecutar el método de prueba para confirmar la corrección de las clases.

4.5.2. **Prueba automatizada.**

Incluso con un método de prueba, todavía no hemos solucionado el problema de comprobar si una clase ha pasado o no la prueba. En el ejemplo anterior, se imprimían tanto los datos de entrada como los resultados, permitiéndonos realizar los resultados pertinentes para determinar si la nómina se calculó correctamente. Esta estrategia tiene las siguientes desventajas:

- Si el cálculo es relativamente simple, es sencillo calcular la respuesta correcta y compararla con la que se muestra en el método de prueba. Sin embargo, si el cálculo es medianamente complicado, volver a calcular el valor cada vez que el código se vuelva a probar no es viable.
- Un buen método de prueba normalmente efectúa muchas pruebas sobre la clase. Cada método que representa algún comportamiento externo se debe probar. También se debe probar cualquier caso especial. La verificación, incluso de cálculos simples, se vuelve algo compleja cuando hay muchos de esos casos especiales.

Una solución es calcular los valores correctos una vez, e incluirlos en el método de prueba, imprimiendo esos valores junto con los que la clase ha calculado. Sin embargo, esto no nos resuelve nuestro segundo problema, la dificultad de comparar todos los resultados correctos y calculados para un número relativamente grande de pruebas. Para solucionar este problema, podemos hacer que el computados compare los valores calculados con los correctos y que nos informe sólo cuando haya una divergencia.

```
if (respuestaCalculada ¡= respuestaCorrecta) {
                System.out.print("*** Error: la nomina calculada");
                System.out.print(" no corresponde con la correcta de ");
                System.out.println(respuestaCorrecta);
                System.out.println("¡Falló la prueba!");
}
```

Este método sí realiza las pruebas y comprueba los resultados por nosotros. La estrategia de pruebas en la que el método de prueba comprueba la corrección de los resultados se llama *prueba automatizada*.

Sin embargo, resulta algo engorroso tener que escribir repetidas veces una secuencia de ifs para comparar resultados. Este tipo de comprobaciones interfieren con el código dedicado a la prueba, distraen y hacen más difícil la lectura y seguimiento del flujo que determina la lógica del este método. Para facilitar la escritura de métodos de prueba automatizados, nos gustaría poseer un método que realiza la prueba anterior y la acción subsiguiente. Además, sería ideal conocer también la secuencia de llamadas de métodos (y los números de línea) que llevaron al error. Podemos obtener esto utilizando el método static dumpStack de la clase predefinida Thread.

```
class Comprobador{
        public static void verifica (bolean condicion, String mensaje){
                if (!condicion){
                        System.out.print("*** Error – fallo del test: ");
                        System.out.println(mensaje);
                        Thread.dumpStack();
                        }
                }
}
```

Por lo tanto, en el método de prueba se podría sustituir, el fragmento:

```
if (respuestaCalculada ¡= respuestaCorrecta) {
                    // Conjunto de System.out.print que muestran el error
}
```

por este otro, más conciso y fácil de leer:

```
Comprobador.verifica (respuestaCalculada == respuestaCorrecta, “Prueba de calculaNomina”);
```

Cuanto más fácil sea escribir una prueba, más probabilidades habrá de que el programador lo haga; un método verifica puede convertirse en una herramienta indispensable para escribir métodos de prueba.

¿En qué consisten unas pruebas adecuadas y concienzudas?

Se debe probar todo el comportamiento. La clase Empleado contiene los siguientes métodos: Empleado(constructor), getNombre, calculaNomina. Nos puede parecer al principio una buena idea probar sólo el método calculaNomina; al fin y al cabo, éste es el método fundamental de la clase. Sin embargo, ignorar los métodos restantes es arriesgarnos a futuros errores. Un constructor incorrecto, por ejemplo, uno que inicie de forma incorrecta la variable pagaHora, sería desastroso. De igual modo, si el método getNombre fuera defectuoso, el nombre que se imprimiría en el cheque estaría equivocado.

Si un método se invoca por un usuario de la clase, debe ser probado directamente por el banco de pruebas. Si el método no es utilizado desde fuera, sino que existe como un método accesorio que se llama sólo desde otros métodos de la clase, también debería ser probado si es posible.

Se debe buscar un orden lógico en las pruebas. Muchas pruebas son independientes unas de otras. Por ejemplo, no importa en que orden probamos calculaNomina con horas extra o sin ellas. Por otro lado, parece sensato probar el constructor antes de probar calculaNomina; después de todo, si el primero no funciona, el segundo producirá definitivamente un resultado erróneo. A menudo los métodos de una clase se pueden ordenar atendiendo a cómo de primitivos son, esto es, en qué medida su implementación depende de otros métodos de la clase.
Cuanto más primitivo es un método, antes se debe probar en la batería de pruebas. Si estamos probando un método que utiliza otros métodos de la clase, la depuración se hace más sencilla si antes hemos probado estos métodos.
Además, nos gustaría ser capaces de imprimir el valor de nuestro objeto para su depuración. Nos gustaría confiar en los métodos que imprimen el objeto, por lo que éstos deben ser probados en primera instancia por la batería de pruebas.

Se debe procurar que todas las sentencias se ejecuten al menos una vez. La *prueba de todas las sentencias* es un técnica muy útil, más cuando hay condicionales en el código, donde resulta de vital importancia. Requiere que la sentencia del código se ejecute al menos una vez durante las pruebas. Existe una alternativa a la técnica de todas las sentencias, conocida como *prueba de todos los caminos*. En esta técnica, no sólo se prueba cada sentencia, sino que se prueba también toda secuencia posible de sentencias del programa.

*Se deben buscar y probar los casos especiales***.** Además de probar cada alternativa de una condición, es una buena idea comprobar ciertos valores que están situados en los límites de la condición.
Los valores de las pruebas a veces requieren especial atención incluso si no hay condicionales implicados. Uno de los ejemplos más comunes es la operación de división. La división por cero no

está definida, por lo tanto dividir por cero siempre es un error. Un método que utiliza el operador de división se debe asegurar de que nunca se aplica cuando el divisor es 0.
Para realizar estas pruebas se pueden emplear los métodos de Caja Blanca y Caja Negra.