1. TÉCNICAS DE EVALUACIÓN DINÁMICA

1.1 Características y Fases de la Prueba

A la aplicación de técnicas de evaluación dinámicas se le denomina también *prueba* del software.

La Figura 1 muestra el contexto en el que se realiza la prueba de software. Concretamente la Prueba de software se puede definir como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas (configuración de la prueba), registrándose los resultados obtenidos. Seguidamente se realiza un proceso de Evaluación en el que los resultados obtenidos se comparan con los resultados esperados para localizar fallos en el software. Estos fallos conducen a un proceso de Depuración en el que es necesario identificar la falta asociada con cada fallo y corregirla, pudiendo dar lugar a una nueva prueba. Como resultado final se puede obtener una determinada Predicción de Fiabilidad, tal como se indicó anteriormente, o un cierto nivel de confianza en el software probado.

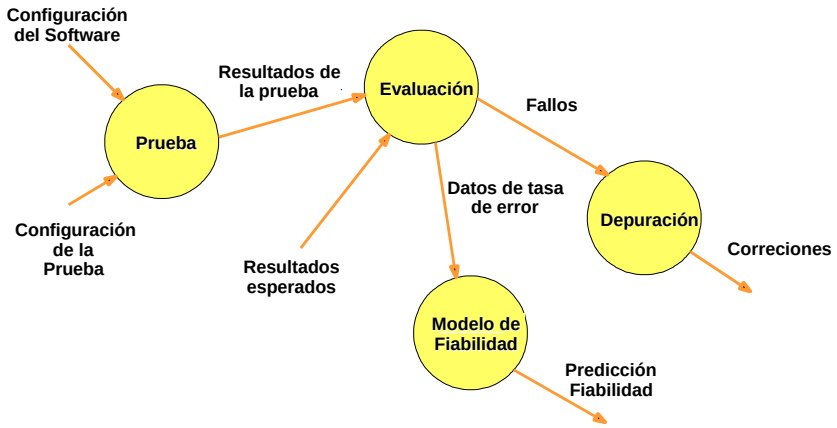

Figura 1. Contexto de la Prueba de Software

El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, únicamente puede demostrar que existen defectos en el software. Nuestro objetivo es pues, diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

Para ser más eficaces (es decir, con más alta probabilidad de encontrar errores), las pruebas deberían ser realizadas por un equipo independiente al que realizó el software. El ingeniero de software que creó el sistema no es el más adecuado para llevar a cabo las pruebas de dicho software, ya que inconscientemente tratará de demostrar que el software funciona, y no que no lo hace, por lo que la prueba puede tener menos éxito.

Una prueba de software, comparando los resultados obtenidos con los esperados. A continuación se presentan algunas características de una buena prueba:

- Una buena prueba ha de tener una alta probabilidad de encontrar un fallo. Para alcanzar este objetivo el responsable de la prueba debe entender el software e intentar desarrollar una *imagen mental* de cómo podría fallar.
- Una buena prueba debe centrarse en dos objetivos: 1) *probar si el software no hace lo que debe hacer*, y 2) *probar si el software hace lo que no debe hacer*.
- Una buena prueba no debe ser redundante. El tiempo y los recursos son limitados, así que *todas las pruebas deberían tener un propósito diferente*.
- Una buena prueba debería ser la "mejor de la cosecha". Esto es, se debería emplear la prueba que tenga la *más alta probabilidad de descubrir una clase entera de errores*.

- Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja, pero si se quieren combinar varias pruebas a la vez se pueden enmascarar errores, por lo que en general, *cada prueba debería realizarse separadamente*.

Veamos ahora cuáles son las tareas a realizar para probar un software:

1. *Diseño de las pruebas*. Esto es, identificación de la técnica o técnicas de pruebas que se utilizarán para probar el software. Distintas técnicas de prueba ejercitan diferentes criterios como guía para realizar las pruebas. Seguidamente veremos algunas de estas técnicas.
2. *Generación de los casos de prueba*. Los casos de prueba representan los datos que se utilizarán como entrada para ejecutar el software a probar. Más concretamente los casos de prueba determinan un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular. Como veremos posteriormente, cada técnica de pruebas proporciona unos criterios distintos para generar estos casos o datos de prueba. Por lo tanto, durante la tarea de generación de casos de prueba, se han de confeccionar los distintos casos de prueba según la técnica o técnicas identificadas previamente. La generación de cada caso de prueba debe ir acompañada del resultado que ha de producir el software al ejecutar dicho caso (como se verá más adelante, esto es necesario para detectar un posible fallo en el programa).
3. *Definición de los procedimientos de la prueba*. Esto es, especificación de cómo se va a llevar a cabo el proceso, quién lo va a realizar, cuándo, …
4. *Ejecución de la prueba*, aplicando los casos de prueba generados previamente e identificando los posibles fallos producidos al comparar los resultados esperados con los resultados obtenidos.
5. *Realización de un informe de la prueba, c*on el resultado de la ejecución de las pruebas, qué casos de prueba pasaron satisfactoriamente, cuáles no, y qué fallos se detectaron.

Tras estas tareas es necesario realizar un proceso de depuración de las faltas asociadas a los fallos identificados. Nosotros nos centraremos en el segundo paso, explicando cómo distintas técnicas de pruebas pueden proporcionar criterios para generar distintos datos de prueba.

1.2 Técnicas de Prueba

Como se ha indicado anteriormente, las técnicas de evaluación dinámica o prueba proporcionan distintos criterios para generar casos de prueba que provoquen fallos en los programas. Estas técnicas se agrupan en:

- *Técnicas de caja blanca o estructurales*, que se basan en un minucioso examen de los detalles procedimentales del código a evaluar, por lo que es necesario conocer la lógica del programa.
- *Técnicas de caja negra o funcionales*, que realizan pruebas sobre la interfaz del programa a probar, entendiendo por interfaz las entradas y salidas de dicho programa. No es necesario conocer la lógica del programa, únicamente la funcionalidad que debe realizar.

La Figura 2 representa gráficamente la filosofía de las pruebas de caja blanca y caja negra. Como se puede observar las pruebas de caja blanca necesitan conocer los detalles procedimentales del código, mientras que las de caja negra únicamente necesitan saber el objetivo o funcionalidad que el código ha de proporcionar.

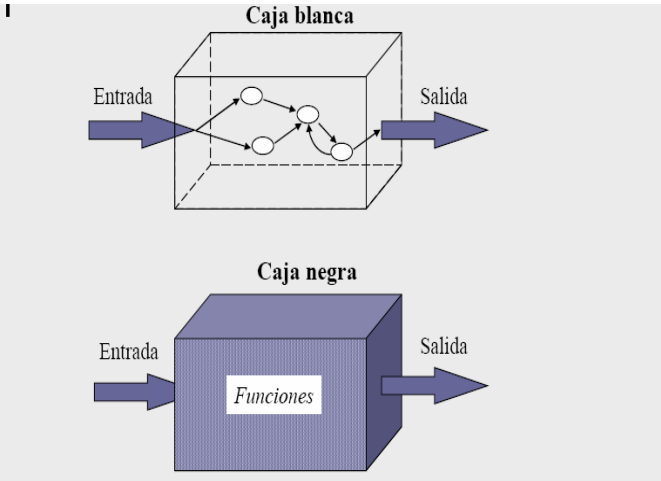

Figura 2. Representación de pruebas de Caja Blanca y Caja Negra

A primera vista parecería que una prueba de caja blanca completa nos llevaría a disponer de un código perfectamente correcto. De hecho esto ocurriría si se han probado todos los posibles caminos por los que puede pasar el flujo de control de un programa. Sin embargo, para programas de cierta envergadura, el número de casos de prueba que habría que generar sería excesivo, nótese que el número de caminos incrementa exponencialmente a medida que el número de sentencias condicionales y bucles aumenta. Sin embargo, este tipo de prueba no se desecha como impracticable. Se pueden elegir y ejercitar ciertos caminos representativos de un programa.

Por su parte, tampoco sería factible en una prueba de caja negra probar todas y cada una de las posibles entradas a un programa, por lo que análogamente a como ocurría con las técnicas de caja blanca, se seleccionan un conjunto representativo de entradas y se generan los correspondientes casos de prueba, con el fin de provocar fallos en los programas.

En realidad estos dos tipos de técnicas son técnicas complementarias que han de aplicarse al realizar una prueba dinámica, ya que pueden ayudar a identificar distintos tipos de faltas en un programa.

A continuación, se describen en detalle los procedimientos propuestos por ambos tipos de técnicas para generar casos de prueba.

1.2.1 Pruebas de Caja Blanca o Estructurales

A este tipo de técnicas se le conoce también como Técnicas de Caja Transparente o de Cristal. Este método se centra en cómo diseñar los casos de prueba atendiendo al *comportamiento interno y la estructura del programa*. Se examina así la lógica interna del programa sin considerar los aspectos de rendimiento.

El objetivo de la técnica es diseñar casos de prueba para que se ejecuten, al menos una vez, todas las sentencias del programa, y todas las condiciones tanto en su vertiente verdadera como falsa.

Como se ha indicado ya, puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos criterios de cobertura lógica, que permiten decidir qué sentencias o caminos se deben examinar con los casos de prueba. Estos criterios son:

- *Cobertura de Sentencias*: Se escriben casos de prueba suficientes para que cada sentencia en el programa se ejecute, al menos, una vez.
- *Cobertura de Decisión*: Se escriben casos de prueba suficientes para que cada decisión en el programa se ejecute una vez con resultado verdadero y otra con el falso.
- *Cobertura de Condiciones*: Se escriben casos de prueba suficientes para que cada condición en una decisión tenga una vez resultado verdadero y otra falso.
- *Cobertura Decisión/Condición*: Se escriben casos de prueba suficientes para que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.
- *Cobertura de Condición Múltiple*: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.
- *Cobertura de Caminos*: Se escriben casos de prueba suficientes para que se ejecuten todos los caminos de un programa. Entendiendo camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida.

Este último criterio es el que se va a estudiar.

1.2.1.1 Cobertura de Caminos

La aplicación de este criterio de cobertura asegura que los casos de prueba diseñados permiten que todas las sentencias del programa sean ejecutadas al menos una vez y que las condiciones sean probadas tanto para su valor verdadero como falso.

Una de las técnicas empleadas para aplicar este criterio de cobertura es la **Prueba del Camino Básico**. Esta técnica se basa en obtener una medida de la complejidad del diseño procedimental de un programa (o de la lógica del programa). Esta medida es la complejidad ciclomática de McCabe, y representa un límite superior para el número de casos de prueba que se deben realizar para asegurar que se ejecuta cada camino del programa.

Los pasos a realizar para aplicar esta técnica son:

- Representar el programa en un grafo de flujo
- Calcular la complejidad ciclomática
- Determinar el conjunto básico de caminos independientes
- Derivar los casos de prueba que fuerzan la ejecución de cada camino.

A continuación, se detallan cada uno de estos pasos.

1.2.1.1.1 Representar el programa en un grafo de flujo

El grafo de flujo se utiliza para representar flujo de control lógico de un programa. Para ello se utilizan los tres elementos siguientes:

- *Nodos*: representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
- *Aristas*: líneas que unen dos nodos.
- *Regiones*: áreas delimitadas por aristas y nodos. Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
- *Nodos predicado*: cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR, ...) se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado. La Figura 3 muestra un ejemplo de condición múltiple.

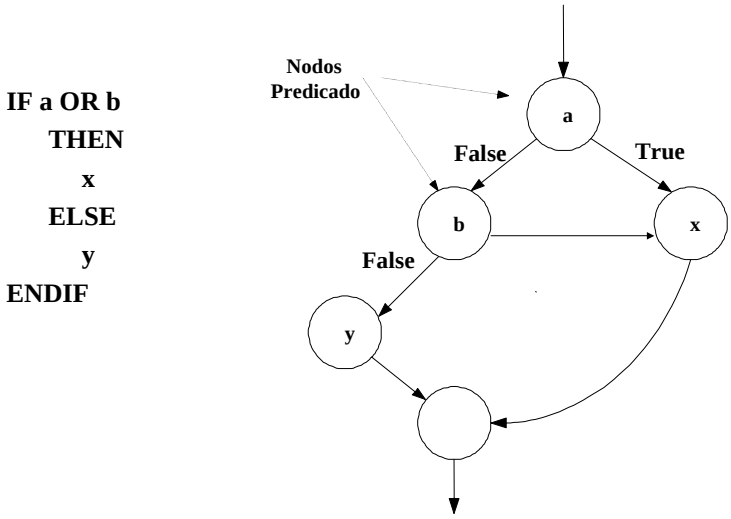

Figura 3. Representación de condición múltiple

Así, cada construcción lógica de un programa tiene una representación. La Figura 4 muestra dichas representaciones.

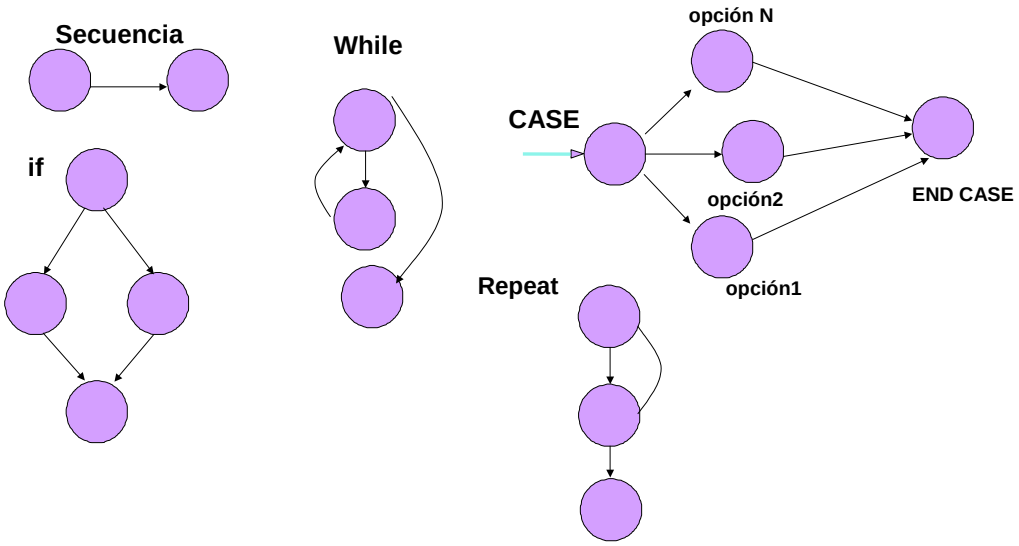

Figura 4. Representación en grafo de flujo de las estructuras lógicas de un programa

La Figura 5 muestra un grafo de flujo del diagrama de módulos correspondiente. Nótese cómo la estructura principal corresponde a un *while*, y dentro del bucle se encuentran anidados dos constructores *if*.

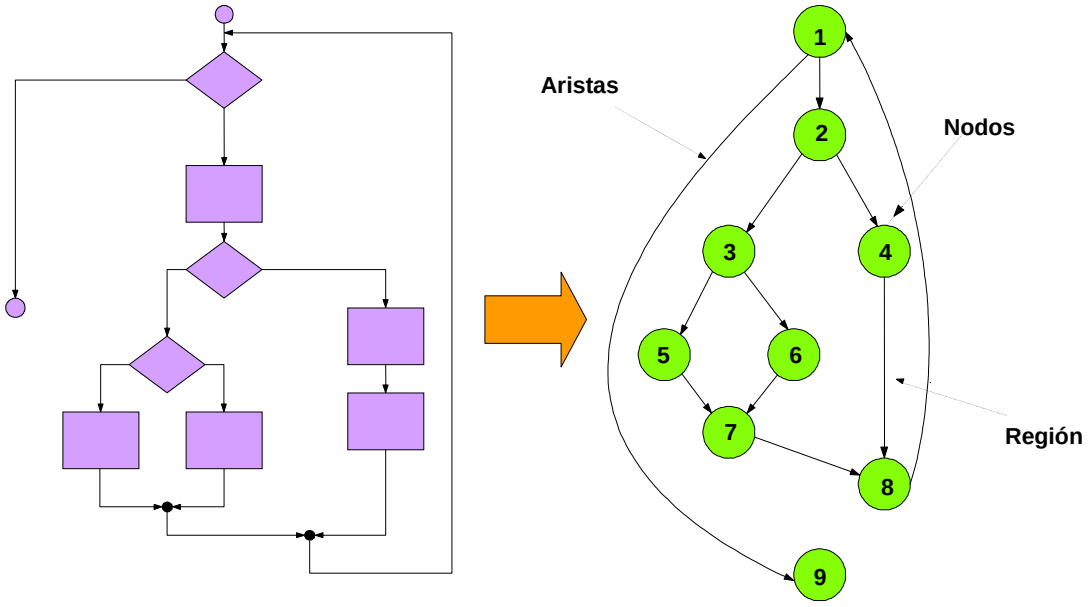

Figura 5. Ejemplo de grafo de flujo correspondiente a un diagrama de módulos

1.2.1.1.2 Calcular la complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, el valor de la complejidad ciclomática define el número de caminos independientes de dicho programa, y por lo tanto, el número de casos de prueba a realizar. Posteriormente veremos cómo se identifican esos caminos, pero primero veamos cómo se puede calcular la complejidad ciclomática a partir de un grafo de flujo, para obtener el número de caminos a identificar.

Existen varias formas de calcular la complejidad ciclomática de un programa a partir de un grafo de flujo:

1. El número de regiones del grafo coincide con la complejidad ciclomática, V(G).
2. La complejidad ciclomática, V(G), de un grafo de flujo G se define como

   V(G) = Aristas – Nodos + 2
3. La complejidad ciclomática, V(G), de un grafo de flujo G se define como

   V(G) = Nodos Predicado + 1

La Figura 6 representa, por ejemplo, las cuatro regiones del grafo de flujo, obteniéndose así la complejidad ciclomática de 4. Análogamente se puede calcular el número de aristas y nodos predicados para confirmar la complejidad ciclomática. Así:

V(G) = Número de regiones = 4

V(G) = Aristas – Nodos + 2 = 11-9 + 2 = 4

V(G) = Nodos Predicado + 1 = 3 +1 = 4

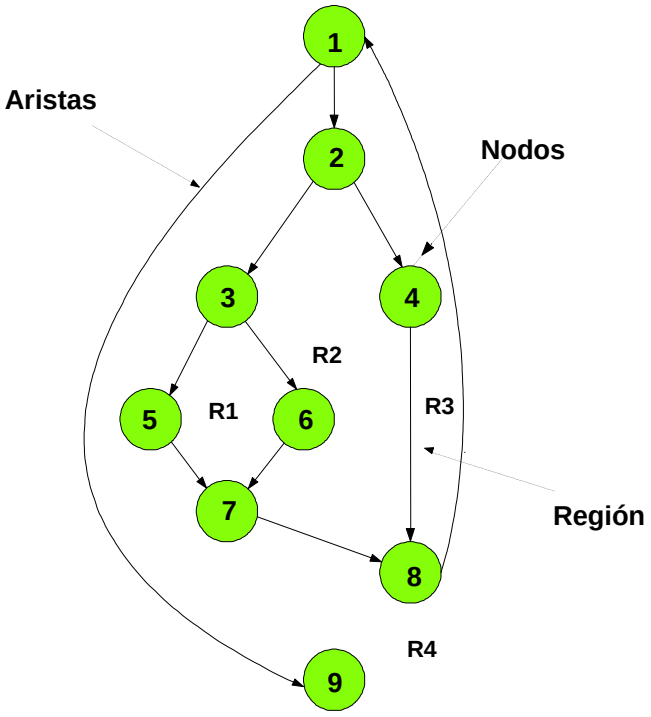

Figura 6. Número de regiones del grafo de flujo

Esta complejidad ciclomática determina el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un programa se han ejecutado al menos una vez, y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa. Veamos ahora, cómo se identifican estos caminos.

1.2.1.1.3 Determinar el conjunto básico de caminos independientes

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición, respecto a los caminos existentes. En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino. En la identificación de los distintos caminos de un programa para probar se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen. De esta manera se intenta que el proceso de depuración sea más sencillo.

El conjunto de caminos independientes de un grafo no es único. No obstante, a continuación, se muestran algunas heurísticas para identificar dichos caminos:

(a) Elegir un *camino principal* que represente una función válida que no sea un tratamiento de error. Debe intentar elegirse el camino que atraviese el máximo número de decisiones en el grafo.

(b) Identificar el segundo camino mediante la localización de la *primera decisión* en el camino de la línea básica alternando su resultado mientras se mantiene el máximo número de decisiones originales del camino inicial.

(c) Identificar un tercer camino, colocando la primera decisión en su valor original a la vez que se altera la *segunda decisión* del camino básico, mientras se intenta mantener el resto de decisiones originales.

(d) Continuar el proceso hasta haber conseguido *tratar todas las decisiones*, intentando mantener como en su origen el resto de ellas.

Este método permite obtener V(G) caminos independientes cubriendo el criterio de cobertura de decisión y sentencia.

Así por ejemplo, para la el grafo de la Figura 6 los cuatro posibles caminos independientes generados serían:

Camino 1: 1-10

Camino 2: 1-2-4-8-1-9

Camino 3: 1-2-3-5-7-8-1-9

Camino 4: 1-2-5-6-7-8-1-9

Estos cuatro caminos constituyen el *camino básico* para el grafo de flujo correspondiente.

1.2.1.1.4 Derivar los casos de prueba que fuerzan la ejecución de cada camino.

El último paso es construir los casos de prueba que fuerzan la ejecución de cada camino. Una forma de representar el conjunto de casos de prueba es como se muestra en la Tabla 1.

Número del Camino	Caso de Prueba	Resultado Esperado

Tabla 1. Posible representación de casos de prueba para pruebas estructurales

1.2.2 Pruebas de Caja Negra o Funcionales

También conocidas como Pruebas de Comportamiento, estas pruebas se basan en la *especificación* del programa o componente a ser probado para elaborar los casos de prueba. El componente se ve como una "Caja Negra" cuyo comportamiento sólo puede ser determinado estudiando sus entradas y las salidas obtenidas a partir de ellas. No obstante, como el estudio de todas las posibles entradas y salidas de un programa sería impracticable se selecciona un conjunto de ellas sobre las que se realizan las pruebas. Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un comportamiento erróneo en nuestro sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos. Entonces, dado que la prueba exhaustiva es imposible, el objetivo final es pues, encontrar una serie de datos de entrada cuya probabilidad de pertenecer al conjunto de entradas que causan dicho comportamiento erróneo sea lo más alto posible.

Al igual que ocurría con las técnicas de Caja Blanca, para confeccionar los casos de prueba de Caja Negra existen distintos criterios. Algunos de ellos son:

- Particiones de Equivalencia.
- Análisis de Valores Límite.
- Métodos Basados en Grafos.
- Pruebas de Comparación.
- Análisis Causa-Efecto.

De ellas, las técnicas que estudiaremos son las dos primeras, esto es, Particiones de Equivalencia y Análisis de Valores Límite.

1.2.2.1 Particiones de Equivalencia

La partición de equivalencia es un método de prueba de Caja Negra que divide el campo de entrada de un programa en *clases de datos* de los que se pueden derivar casos de prueba. La partición equivalente se dirige a una definición de casos de prueba que descubran *clases de errores*, reduciendo así el número total de casos de prueba que hay que desarrollar.

En otras palabras, este método intenta dividir el dominio de entrada de un programa en un número finito de *clases de equivalencia.* De tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realzada con cualquier otro valor de dicha clase. Esto quiere decir que si el caso de prueba correspondiente a una clase de equivalencia detecta un error, el resto de los casos de prueba de dicha clase de equivalencia deben detectar el mismo error. Y viceversa, si un caso de prueba no ha detectado ningún error, es de esperar que ninguno de los casos de prueba correspondientes a la misma clase de equivalencia encuentre ningún error.

El diseño de casos de prueba según esta técnica consta de dos pasos:

1. Identificar las clases de equivalencia.
2. Identificar los casos de prueba.

1.2.2.1.1 Identificar las clases de equivalencia

Una clase de equivalencia representa un conjunto de estados válidos y no válidos para las condiciones de entrada de un programa. Las clases de equivalencia se identifican examinando cada condición de entrada (normalmente una frase en la especificación) y dividiéndola en dos o más grupos. Se definen dos tipos de clases de equivalencia, las *clases de equivalencia válidas*, que representan entradas válidas al programa, y las *clases de equivalencia no válidas*, que representan valores de entrada erróneos. Estas clases se pueden representar en una tabla como la Tabla 2.

Condición Externa	Clases de Equivalencia Válidas	Clases de Equivalencia No Válidas

Tabla 2. Tabla para la identificación de clases de equivalencia

En función de cuál sea la condición de entrada se pueden seguir las siguientes pautas identificar las clases de equivalencia correspondientes:

- Si una condición de entrada especifica un *rango de valores*, identificar una clase de equivalencia válida y dos clases no válidas. Por ejemplo, si un contador puede ir de 1 a 999, la clase válida sería “1 <= contador <= 999. Mientras que las clases no válidas serían “contador < 1” y “contador > 999”
- Si una condición de entrada especifica un *valor* o *número de valores*, identificar una clase válida y dos clases no válidas. Por ejemplo, si tenemos que puede haber desde uno hasta seis propietarios en la vida de un coche. Habrá una clase válida y dos no válidas: “no hay propietarios” y “más de seis propietarios”.

- Si una condición de entrada especifica un conjunto de valores de entrada, identificar una clase de equivalencia válida y una no válida. Sin embargo, si hay razones para creer que cada uno de los miembros del conjunto será tratado de distinto modo por el programa, identificar una clase válida por cada miembro y una clase no válida. Por ejemplo, el tipo de un vehículo puede ser: autobús, camión, taxi, coche o moto. Habrá una clase válida por cada tipo de vehículo admitido, y la clase no válida estará formada por otro tipo de vehículo.
- Si una condición de entrada especifica una situación que debe ocurrir, esto es, es lógica, identificar una clase válida y una no válida. Por ejemplo, el primer carácter del identificador debe ser una letra. La clase válida sería "identificador que comienza con letra", y la clase inválida sería "identificador que no comienza con letra".
- En general, si hay alguna razón para creer que los elementos de una clase de equivalencia no se tratan de igual modo por el programa, dividir la clase de equivalencia entre clases de equivalencia más pequeñas para cada tipo de elementos.

1.2.2.1.2 Identificar los casos de prueba

El objetivo es minimizar el número de casos de prueba, así cada caso de prueba debe considerar tantas condiciones de entrada como sea posible. No obstante, es necesario realizar con cierto cuidado los casos de prueba de manera que no se enmascaren faltas. Así, para crear los casos de prueba a partir de las clases de equivalencia se han de seguir los siguientes pasos:

1. Asignar a cada clase de equivalencia un número único.
2. Hasta que todas las clases de equivalencia hayan sido cubiertas por los casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para cubrir una única clase no válida no cubierta.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares. Por ejemplo, si tenemos dos clases válidas: "introducir cantidad entre 1 y 99" y "seguir con letra entre A y Z", el caso *105 1* (dos errores) puede dar como resultado *105 fuera de rango de cantidad*, y no examinar el resto de la entrada no comprobando así la respuesta del sistema ante una posible entrada no válida.

1.2.2.2 Análisis de Valores Límite

La experiencia muestra que los casos de prueba que exploran las condiciones límite producen mejor resultado que aquellos que no lo hacen. Las condicione límite son aquellas que se hayan en los márgenes de la clase de equivalencia, tanto de entrada como de salida. Por ello, se ha desarrollado el *análisis de valores límite* como técnica de prueba. Esta técnica nos lleva a elegir los casos de prueba que ejerciten los valores límite.

Por lo tanto, el análisis de valores límite complementa la técnica de partición de equivalencia de manera que:

- En lugar de seleccionar cualquier caso de prueba de las clases válidas e inválidas, se eligen los casos de prueba en los extremos.
- En lugar de centrase sólo en el dominio de entrada, los casos de prueba se diseñan también considerando el dominio de salida.

Las pautas para desarrollar casos de prueba con esta técnica son:

- Si una condición de entrada especifica un rango de valores, se diseñarán casos de prueba para los dos límites del rango, y otros dos casos para situaciones justo por debajo y por encima de los extremos.
- Si una condición de entrada especifica un número de valores, se diseñan dos casos de prueba para los valores mínimo y máximo, además de otros dos casos de prueba para valores justo por encima del máximo y justo por debajo del mínimo.
- Aplicar las reglas anteriores a los datos de salida.
- Si la entrada o salida de un programa es un conjunto ordenado, habrá que prestar atención a los elementos primero y último del conjunto.

1.3 Estrategia de Pruebas

La estrategia que se ha de seguir a la hora de evaluar dinámicamente un sistema software debe permitir comenzar por los componentes más simples y más pequeños e ir avanzando progresivamente hasta probar todo el software en su conjunto. Más concretamente, los pasos a seguir son:

1. Pruebas Unitarias. Comienzan con la prueba de cada módulo.
2. Pruebas de Integración. A partir del esquema del diseño, los módulos probados se vuelven a probar combinados para probar sus interfaces.
3. Prueba del Sistema. El software ensamblado totalmente con cualquier componente hardware que requiere se prueba para comprobar que se cumplen los requisitos funcionales.
4. Pruebas de Aceptación. El cliente comprueba que el software funciona según sus expectativas.

1.3.1 Pruebas Unitarias

La prueba de unidad es la primera fase de las pruebas dinámicas y se realizan sobre cada módulo del software de manera independiente. El objetivo es comprobar que el módulo, entendido como una unidad funcional de un programa independiente, está correctamente codificado. En estas pruebas cada módulo será probado por separado y lo hará, generalmente, la persona que lo creo. En general, un módulo se entiende como un componente software que cumple las siguientes características:

- Debe ser un bloque básico de construcción de programas.
- Debe implementar una función independiente simple.
- Podrá ser probado al cien por cien por separado.
- No deberá tener más de 500 líneas de código.

Tanto las pruebas de caja blanca como las de caja negra han de aplicar para probar de la manera más completa posible un módulo. Nótese que las pruebas de caja negra (los casos de prueba) se pueden especificar antes de que módulo sea programado, no así las pruebas de caja blanca.

1.3.2 Pruebas de Integración

Aún cuando los módulos de un programa funcionen bien por separado es necesario probarlos conjuntamente: un módulo puede tener un efecto adverso o inadvertido sobre otro módulo; las

subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individuamente puede crecer hasta niveles inaceptables al combinar los módulos; los datos pueden perderse o malinterpretarse entre interfaces, etc.

Por lo tanto, es necesario probar el software ensamblando todos los módulos probados previamente. Ésta es el objetivo de la pruebas de integración.

A menudo hay una tendencia a intentar una integración *no incremental*; es decir, a combinar todos los módulos y probar todo el programa en su conjunto. El resultado puede ser un poco caótico con un gran conjunto de fallos y la consiguiente dificultad para identificar el módulo (o módulos) que los provocó.

En contra, se puede aplicar la integración *incremental* en la que el programa se prueba en pequeñas porciones en las que los fallos son más fáciles de detectar. Existen dos tipos de integración incremental, la denominada *ascendente* y *descendente*. Veamos los pasos a seguir para cada caso:

Integración incremental ascendente:

1. Se combinan los módulos de bajo nivel en grupos que realicen una subfunción específica
2. Se escribe un *controlador* (un programa de control de la prueba) para coordinar la entrada y salida de los casos de prueba.
3. Se prueba el grupo
4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

La Figura 7 muestra este proceso. Concretamente, se forman los grupos 1, 2 y 3 de módulos relacionados, y cada uno de estos grupos se prueba con el controlador C1, C2 y C3 respectivamente. Seguidamente, los grupos 1 y 2 son subordinados de Ma, luego se eliminan los controladores correspondientes y se prueban los grupos directamente con Ma. Análogamente se procede con el grupo 3 eliminando el controlador C3 y probando el grupo directamente con Mb. Tanto Ma y Mb se integran finalmente con el módulo Mc y así sucesivamente.

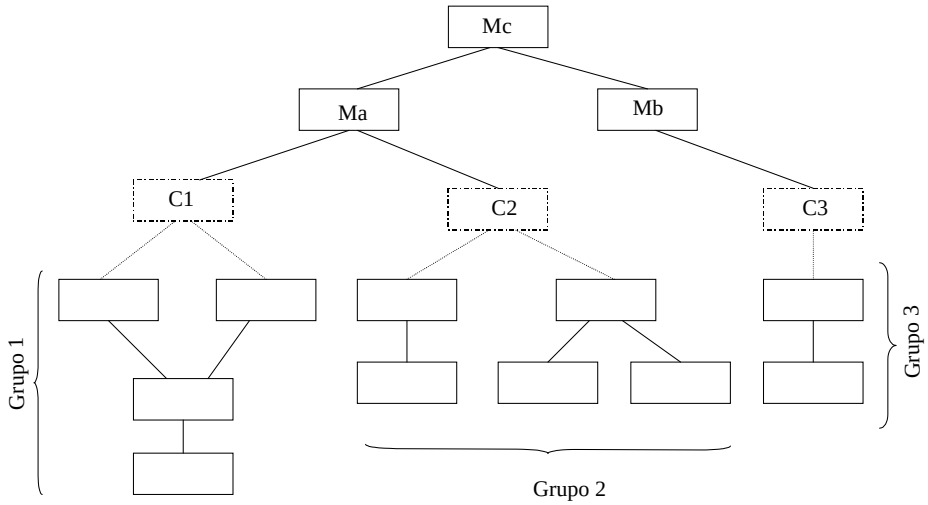

Figura 7. Integración ascendente

Integración incremental descendente:

1. Se usa el módulo de control principal como controlador de la prueba, creando *resguardos* (módulos que simulan el funcionamiento de los módulos que utiliza el que

está probando) para todos los módulos directamente subordinados al módulo de control principal.

2. Dependiendo del enfoque e integración elegido (es decir, primero-en-profundidad, o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.

La Figura 8 muestra un ejemplo de integración descendiente. Supongamos que se selecciona una integración descendiente por profundidad, y que por ejemplo se prueba M1, M2 y M4. Sería entonces necesario preparar resguardos para M5 y M6, y para M7 y M3. Estos resguardos se ha representado en la figura como R5, R6, R7 y R4 respectivamente. Una vez realizada esta primera prueba se sustituiría R5 por M5, seguidamente R6 por M6, y así sucesivamente hasta probar todos los módulos.

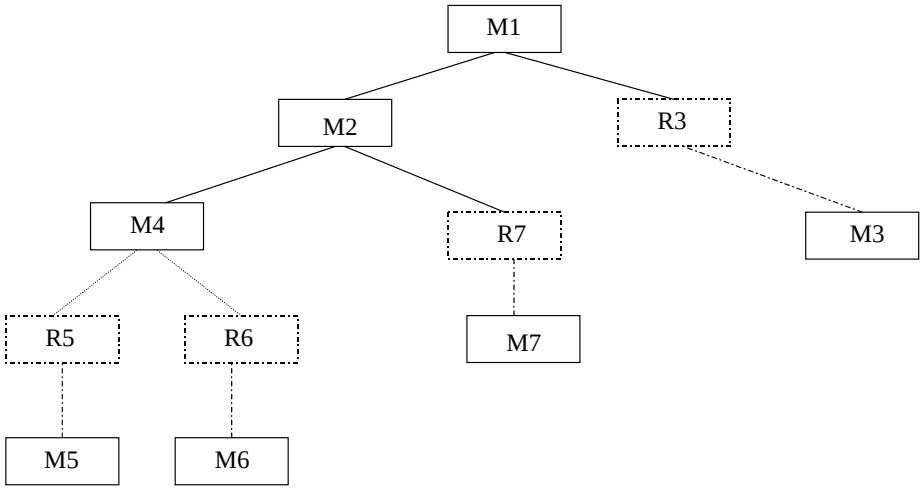

Figura 8. Integración descendiente

Para la generación de casos de prueba de integración, ya sea descendente o ascendente se utilizan técnicas de caja negra.

1.3.3 Pruebas del Sistema

Este tipo de pruebas tiene como propósito ejercitar profundamente el sistema para verificar que se han integrado adecuadamente todos los elementos del sistema (hardware, otro software, etc.) y que realizan las funciones adecuadas. Concretamente se debe comprobar que:

- Se cumplen los requisitos funcionales establecidos.
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario.
- La adecuación de la documentación de usuario.
- Rendimiento y respuesta en condiciones límite y de sobrecarga.

Para la generación de casos de prueba de sistema se utilizan técnicas de caja negra.

Este tipo de pruebas se suelen hacer inicialmente en el entrono del desarrollador, denominadas *Pruebas Alfa,* y seguidamente en el entrono del cliente denominadas *Pruebas Beta*.

1.3.4 Pruebas de Aceptación

A la hora de realizar estas pruebas, el producto está listo para implantarse en el entorno del cliente. El usuario debe ser el que realice las pruebas, ayudado por personas del equipo de pruebas, siendo deseable, que sea el mismo usuario quien aporte los casos de prueba.

Estas pruebas se caracterizan por:

- Participación activa del usuario, que debe ejecutar los casos de prueba ayudado por miembros del equipo de pruebas.
- Están enfocadas a probar los requisitos de usuario, o mejor dicho a demostrar que no se cumplen los requisitos, los criterios de aceptación o el contrato. Si no se consigue demostrar esto el cliente deberá aceptar el producto
- Corresponden a la fase final del proceso de desarrollo de software.

Es muy recomendable que las pruebas de aceptación se realicen en el entorno en que se va a explotar el sistema (incluido el personal que lo maneje). En caso de un producto de interés general, se realizan pruebas con varios usuarios que reportarán sus valoraciones sobre el producto.

Para la generación de casos de prueba de aceptación se utilizan técnicas de caja negra.

1.3.5 Pruebas de Regresión

La regresión consiste en la repetición selectiva de pruebas para detectar fallos introducidos durante la modificación de un sistema o componente de un sistema. Se efectuarán para comprobar que los cambios no han originado efectos adversos no intencionados o que se siguen cumpliendo los requisitos especificados.

En las pruebas de regresión hay que:

- Probar íntegramente los módulos que se han cambiado.
- Decidir las pruebas a efectuar para los módulos que no han cambiado y que han sido afectados por los cambios producidos.

Este tipo de pruebas ha de realizarse, tanto durante el desarrollo cuando se produzcan cambios en el software, como durante el mantenimiento.