Curso 2023/24

Tema-03 (3ªEval)

Diseño y realización de Pruebas / Testing

Entornos de desarrollo – IES BARAJAS
MIGUEL HH

Tema-03(3ªEval)

Contenido

1.- DISEÑO Y REALIZACIÓN DE PRUEBAS:

1.1.- Qué es el software de pruebas o Testing.

El software de pruebas o testing se basa en un conjunto de actividades dirigidas a facilitar la identificación y/o evaluación de propiedades en el software.

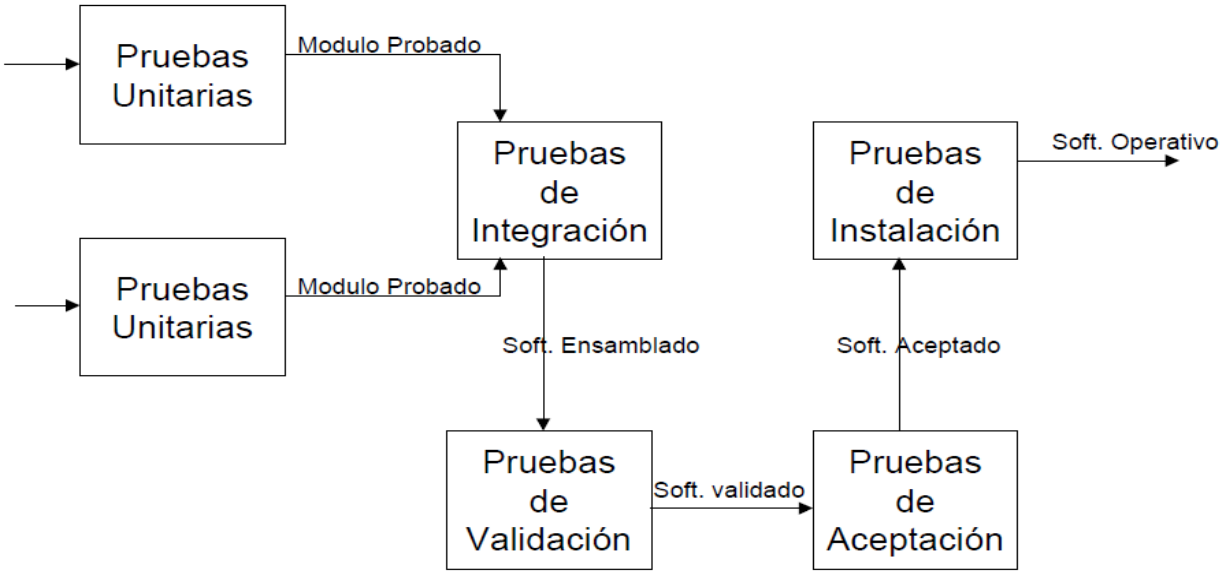
Una de las propiedades más interesantes consiste en analizar un producto software para detectar las diferencias entre el comportamiento real con el esperado. Es decir, comparar "lo que es" con "lo que debería ser".

En resumen, podemos decir:

- **Prueba**: Proceso de ejecutar un programa con la intención de encontrar errores.
- **Buen caso de prueba**: Un buen caso de prueba es el que tiene alta probabilidad de detectar un error todavía no descubierto.
- **Caso de éxito**: Un caso con éxito es el que detecta un error todavía no descubierto.
- **Verificación**: El proceso de evaluación de un sistema (o de uno de sus componentes) para determinar si los productos de una fase dada, satisfacen las condiciones impuestas al comienzo de dicha fase: *¿Estamos construyendo correctamente el producto?*.
- **Validación**: El proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos marcados por el usuario. *¿Estamos construyendo el producto correcto?*.
- **Error**: Equivocación realizada por una persona (en la actividad del desarrollo software).
- **Defecto**: resultado de introducir un error en un artefacto software.
- **Fallo**: representación del defecto en el comportamiento del sistema.
- **Caso de prueba**: Es un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados. Tiene un objetivo concreto (probar algo).
- **Procedimiento de prueba**: Pasos que hay que llevar a cabo para probar uno (o varios) casos de prueba: ¿Cómo probar el caso de prueba y verificar si ha tenido éxito?.
- **Componente de prueba**: Programa que automatiza la ejecución de uno (o varios) casos de prueba. Una vez escrito, se puede probar muchas veces (cada vez que haya un cambio en el código de una clase que pueda afectarle).
    - **¿Cómo escribir componentes de prueba?**:
        - *Se puede escribir "ad hoc"(Para esto).* Por cada caso de prueba, se escribe el código correspondiente en el componente (cambiaría el código).
        - *Se pueden usar entornos de trabajo disponibles para pruebas*. Ejemplo: JUnit para Java.
    - **Técnicas de testing**:
        - *Estáticas*. Buscan fallos sobre el sistema en reposo. Se puede aplicar sobre mecanismos de requerimientos, análisis, diseño y código.
            - *Técnicas*: Revisiones, Inspecciones, Walkthrough y Auditorías de calidad.
        - *Dinámicas*. Se ejecuta y observa el comportamiento de un producto: Estímulo → Proceso → Respuesta.
            - *Tipos:* Caja Blanca: Se centran en el código. Caja Negra: Se centran en los requisitos funcionales.

1.2.- Flujo de información en la prueba.

El flujo de información en la prueba se puede ver en el siguiente esquema:



1.3.- Principios de las pruebas o del Testing.

Los principios de las pruebas o testing son:

- El objetivo del testing consiste en la detección de defectos en el software. Una vez identificados, el proceso de debugging, está relacionado con la búsqueda de dónde ocurren los defectos.
- Representa la última revisión de la especificación de los requisitos, el diseño y el código.
- Es uno de los métodos utilizados para asegurar la calidad del software.
- Muchas organizaciones consumen entre el 40-50% del tiempo de desarrollo en el testing.
- Aunque se utilicen los mejores métodos de revisión, el testing es siempre necesario.
- Se debe hacer un seguimiento hasta ver si se cumplen los requisitos del cliente.
- El principio de Pareto es aplicable a las pruebas del software.
    - El 80% de los errores está en el 20% de los módulos.
    - Hay que identificar esos módulos y probarlos muy bien.
- Empezar por lo pequeño y progresar hacia lo grande.
- No son posibles las pruebas exhaustivas.
- Son más eficientes las pruebas dirigidas por un equipo independiente al que ha creado el código.
- La prueba es un proceso de ejecución con la intención de descubrir defectos.
- Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo defecto.
    - Un caso de prueba no debe ser redundante.
    - Debe ser el mejor de un conjunto de pruebas de propósito similar.
    - No debe ser ni muy sencillo ni excesivamente complejo: es mejor realizar cada prueba de forma separada si se quiere probar diferentes casos.
- Una prueba tiene éxito si descubre un nuevo defecto.

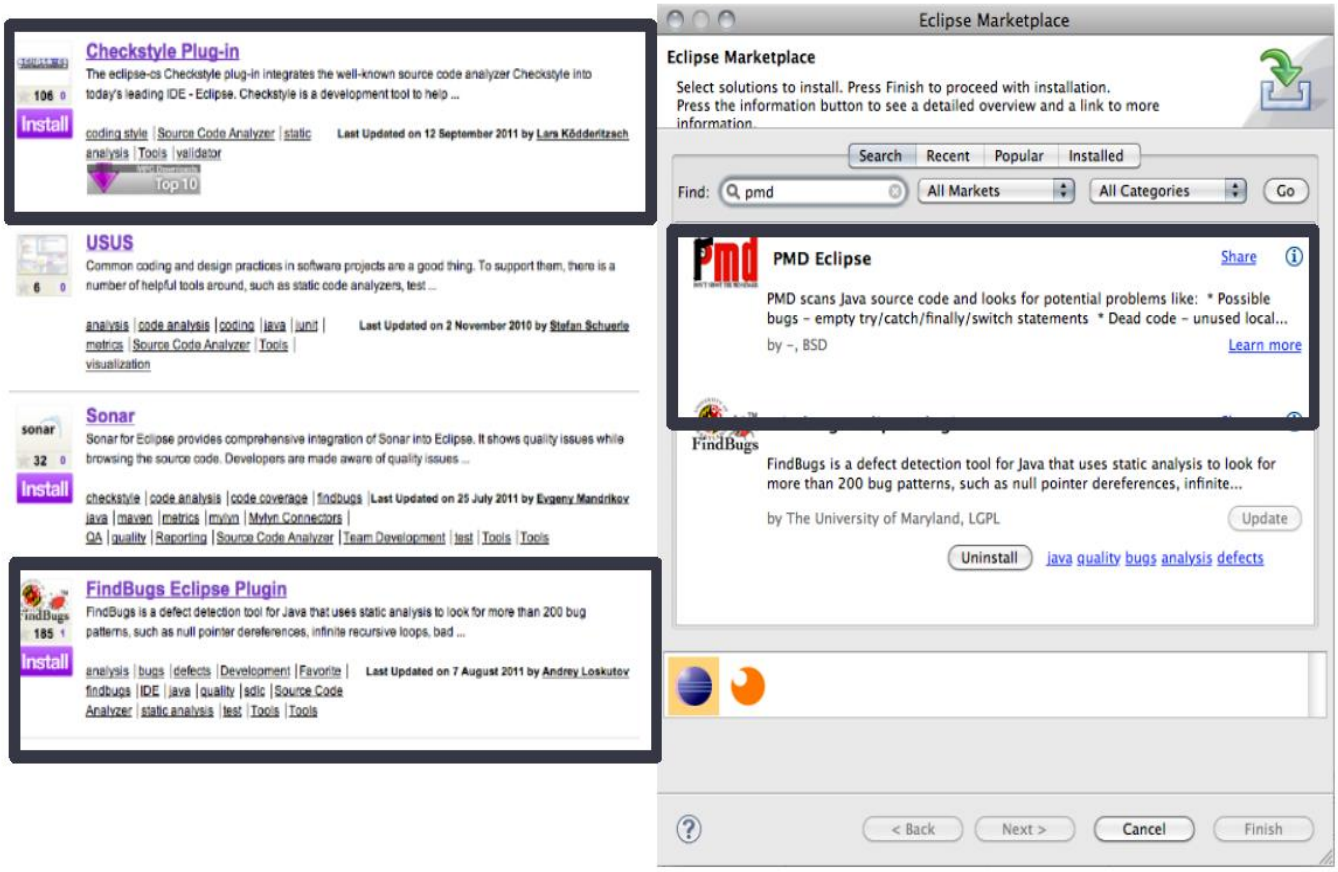
1.4.- Objetivos de una prueba.

Los principales objetivos de una prueba son:

- La prueba es un proceso de ejecución con la intención de descubrir defectos.
- Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo defecto.
- Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo defecto.
    - Un caso de prueba no debe ser redundante.
    - Debe ser el mejor de un conjunto de pruebas de propósito similar.
    - No debe ser ni muy sencillo ni excesivamente complejo: es mejor realizar cada prueba de forma separada si se quiere probar diferentes casos.
- Una prueba tiene éxito si descubre un nuevo defecto.

1.5.- Herramientas de una prueba.

Las principales herramientas existente para hacer pruebas son:



2.- TIPOS DE PRUEBAS QUE SE PUEDEN REALIZAR:

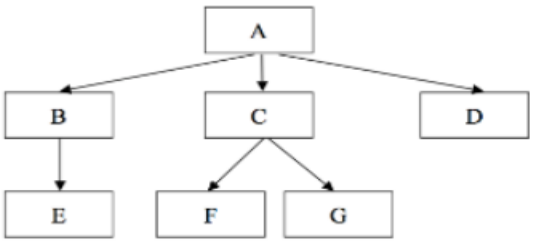
2.1.- Pruebas de unidad.

- Se trata de las pruebas formales que permiten declarar que un módulo está listo y terminado (no las informales que se realizan mientras se desarrollan los módulos).
- Las pruebas de unidad pueden abarcar desde un módulo hasta un grupo de módulos (incluso un programa completo).
- Estas pruebas suelen realizarlas el propio personal de desarrollo, pero evitando que sea el propio programador del módulo.

2.2.- Pruebas de Integración.

- Implican una progresión ordenada de pruebas que van desde los componentes o módulos y que culminan en el sistema completo.
- Tipos fundamentales de integración:
  - ***Integración incremental***: Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados.
    - *Ascendente*: Se comienza por los módulos hoja.
    - *Descendente*: Se comienza por el módulo raíz.
  - ***Integración no incremental***: Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo.
- Habitualmente las pruebas de unidad y de integración se solapan y mezclan en el tiempo.

Esquema de las pruebas de integración



2.3.- Pruebas de Sistema.

- Es el proceso de prueba de un sistema integrado de hardware y software para comprobar lo siguiente:
  - Cumplimiento de todos los requisitos funcionales, considerando el producto software final al completo en un entorno de sistema.
  - El funcionamiento y rendimiento en las interfaces hardware, software, de usuario y de operador.
  - El funcionamiento y rendimiento en las interfaces hardware, software, de usuario y de operador.
  - Adecuación de la documentación de usuario.
  - Ejecución y rendimiento en condiciones límite y de sobrecarga.

2.4.- Pruebas de Aceptación.

- Es la prueba planificada y organizada formalmente para determinar si se cumplen los requisitos de aceptación marcados por el cliente.
- Sus características principales son las siguientes:
  - Participación del usuario.
  - Enfocadas hacia la prueba de los requisitos de usuario especificados.
- Está considerada como la fase final del proceso para crear una confianza en que el producto es el apropiado para su uso en explotación.

3.- CLASIFICACIÓN DE LAS PRUEBAS:

3.1.- Clasificación por objetivos.

- **Funcionales**
  - Indican “lo que el sistema hace”.
  - Se basan en las funcionalidades detalladas en la especificación de requerimientos para corroborar que el sistema desarrollado contenga todas las funcionalidades necesarias.

- **No Funcionales**
  - Indican “cómo trabaja el sistema”. Se pueden medir de diversas maneras, por ejemplo, por medio de tiempos de respuesta en el caso de pruebas de desempeño.

- **Estructurales**
  - Prueban el comportamiento interno del sistema, o sea su arquitectura.
  - Son pruebas de “Caja Blanca” (sinónimos).

3.2.- Clasificación por Dinámicas/Estáticas.

- **Pruebas dinámicas**
  - Son todas aquellas pruebas que para su ejecución requieren la ejecución de la aplicación.
  - Debido a la naturaleza dinámica de la ejecución de pruebas es posible medir con mayor precisión el comportamiento de la aplicación desarrollada.

- **Pruebas estáticas**
  - Se realizan sin ejecutar el código de la aplicación.
  - Buscan testear los documentos, diagramas, diseños que son parte del sistema, pero no son ‘código ejecutable’.
  - Verificación objetiva en función de los estándares de calidad vigentes.
  - Buscan defectos (no corregirlos) de forma temprana.

3.3.- Clasificación por Manuales / Automatizadas.

❖ **Pruebas manuales**

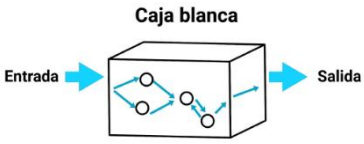
- Son aquellas pruebas que ejecuta una persona, siguiendo el 'paso a paso' definido y verificando los resultados obtenidos contra los esperados.
- Se basan en las funcionalidades detalladas en la especificación de requerimientos para corroborar que el sistema desarrollado contenga todas las funcionalidades necesarias.

❖ **Pruebas automatizadas**

- Utilizan un software especial adicional, donde se configuran / codifican las pruebas y luego se ejecutan sin intervención humana.
- Compara automáticamente los resultados obtenidos y los resultados esperados, informando cuáles pruebas pasaron y cuáles fallaron.
- Son beneficiosas para pruebas repetitivas.
- Tipos de Test para pruebas manuales y automatizadas:

Automatizado y Manual Test Funcional Simulaciones	**Manual** Test Exploratorio - Test de Usabilidad - Test de Aceptación - Alpha y Beta
Test Unitarios **Automatizado**	Test de Performance - Test de Seguridad **Herramientas**

3.4.- Clasificación por Caja Blanca / Caja Negra.

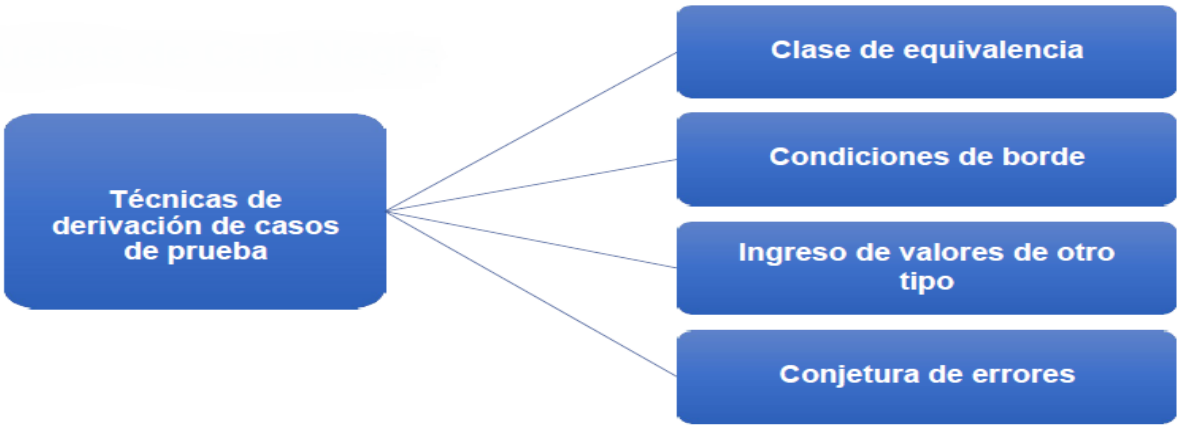


❖ Caja Blanca

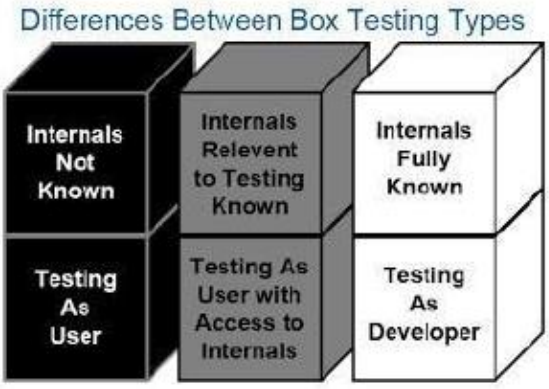
- Son pruebas estructurales.
- Prueban lo que el software hace.
- Evalúa el resultado aprovechando el conocimiento interno del código para dirigir la prueba.
- Se basa en cómo está estructurado el componente internamente y cómo está definido.
- Se trata de ejecutar todas las sentencias para encontrar errores lo antes posible.
- No garantizan el cumplimiento de las especificaciones funcionales (son pruebas complementarias a éstas).
- Cada caso de prueba debe probar sólo un camino de todos los posibles, por ejemplo, si la función contiene una condición "if / else", se deben hacer 2 casos de prueba.
- Si aparece "or" o "and" también se deben generar 2 casos de prueba para la misma condición.
- El desarrollador al conocer la estructura interna puede dirigirse directamente a las partes del código más riesgosas.
- Generalmente es realizado por/con el desarrollador.
- Se representa el flujo de control de una pieza de código a través de un grafo de flujo.

❖ Caja negra

- Son pruebas funcionales o inducidas por datos.
- Prueba lo que el software debería hacer.
- Toma como punto de partida la definición del módulo a probar.
- Las pruebas exhaustivas de caja negra, son imposibles de realizar en la mayoría de los sistemas.
- ¿La solución?: Acotar las pruebas.
- ¿Cómo?: Seleccionando subconjuntos de ellas que permitan cubrir un conjunto extenso de otros casos de prueba posibles.
- Técnicas de derivación de casos de prueba:



- **Caja gris**
  - Combina elementos de la caja negra y caja blanca.
  - Se conoce parte de la implementación o estructura interna y se generan condiciones y casos que no genera una prueba de caja negra.
  - El conocimiento es “parcial”, no “total” (lo que sería caja blanca).
  - Diferencias entre los tipos de pruebas de caja:

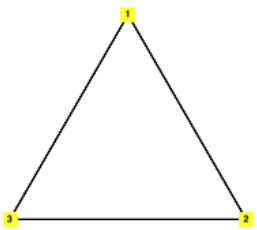


5.- DESARROLLO DE PRUEBAS DE CAJA BLANCA:

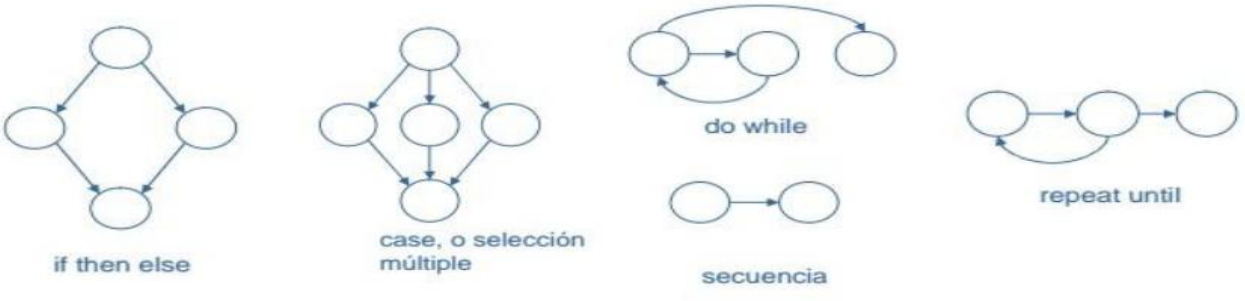
- Intentan garantizar que todos los caminos de ejecución del programa quedan cubiertos.
- Usa la estructura de control para obtener los casos de prueba:
  - *De condición*: Diseñar casos de prueba para que todas las condiciones del programa se evalúen a cierto/falso.
  - *De bucles*: Diseñar casos de prueba para que se intente ejecutar un bucle 0,1,…,n-1,n y n+1 veces (siendo n el número máximo).
- Está considerada como la fase final del proceso para crear una confianza en que el producto es el apropiado para su uso en explotación.
- Conducidas por la estructura lógica (interna) del programa.
- Ejercitar todas las posibles condiciones que se pueden dar.

5.1.- Prueba del camino básico (caja blanca).

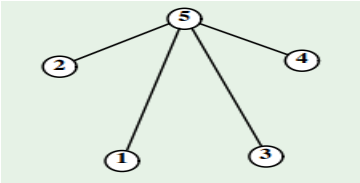
- Es una técnica que permite al desarrollador obtener la medida de la complejidad de nuestro sistema.
- Puede usar esta medida para la definición de un conjunto de caminos de ejecución.
- Para obtener esta medida de complejidad utilizaremos la técnica de representación de grafo de flujo.
- Un nodo es cualquier tipo de sentencia en programación. Por ejemplo: cont=con+1; → nodo 1, cont=0 → nodo 2. Aunque se pueden representar en un mismo nodo ya que son sentencias secuenciales (bucles no) para optimizar estos gráficos.
- Es una técnica que permite al desarrollador obtener la medida de la complejidad de nuestro sistema.
- Puede usar esta medida para la definición de un conjunto de caminos de ejecución.
- Para obtener esta medida de complejidad utilizaremos la técnica de representación de grafo de flujo.
- **Grafos**: $G = (V, E)$.
  - **Definición**: Un grafo es un conjunto de vértices o nodos [V], (con una relación entre ellos), unidos por aristas, arcos o caminos [E], (un conjunto de pares pertenecientes a V). Gráficamente, los vértices son puntos y las aristas son líneas que los unen. Ejemplo de Grafo con 3 vértices y 3 aristas:



- Representaciones de flujos de un grafo:



- ***Grafo dirigido***: La relación sobre V no es simétrica, por lo tanto, las aristas no tienen dirección.
- ***Grado de un vértice***: La relación sobre V sí es simétrica, por lo tanto, las aristas si tienen dirección.
- ***Grado de un vértice***: El grado de un vértice es el número de aristas de las que es extremo. Se dice que un vértice es 'par' o 'impar' según lo sea su grado. Ejemplo de grafo con el vértice (o nodo) número 5 que tiene un ***grado cuatro*** y los demás nodos son de grado uno.



- ***Camino simple***: Camino en el que todos sus vértices, excepto, tal vez, el primero y el ultimo, son distintos.
- ***Longitud de un camino***: Número de arcos del camino.
- ***Complejidad ciclomática***: La Complejidad Ciclomática es una métrica del software en ingeniería del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Nos indicará el número máximo de caminos independientes que va a tener un grafo, (los caminos no tienen por qué ser los mismos), lo importante es que entre los caminos independientes hallados se haya pasado por todas las aristas almeno una vez. Su cálculo se puede realizar de tres formas:
  - V(G) = Número de regiones del grafo. Los dos triángulos y el área externa, el cuadrado que forma el conjunto
  - V(G) = Aristas – Nodos + 2
  - V(G) = Nodos predicado + 1 → (Nodo predicado es aquel del que salen 2 o más flechas).

Cuadro para la Evaluación del riesgo según la métrica de la complejidad ciclomática

Complejidad ciclomática	Evaluación del riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado
Entre 21 y 50	Programas o métodos complejos, alto riesgo
Mayor que 50	Programas o métodos no testeables, muy alto riesgo

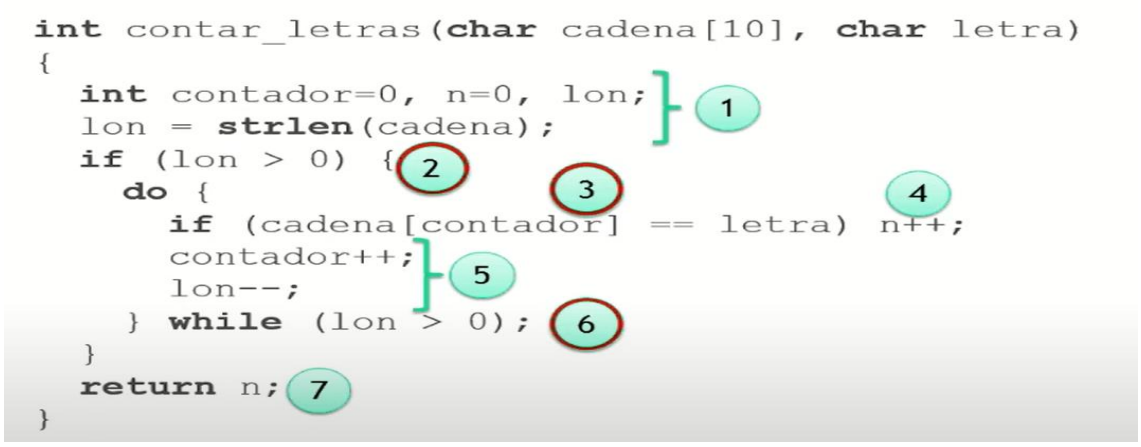
5.2.- Ejemplo del camino básico (caja blanca).

Dado el siguiente fragmento de código, calcular su Grafo Asociado. Además, especifica los nodos y aristas que tiene. Una vez hallado esto calcula el número máximo de caminos básicos que tiene, (poniendo los caminos independientes hallados).

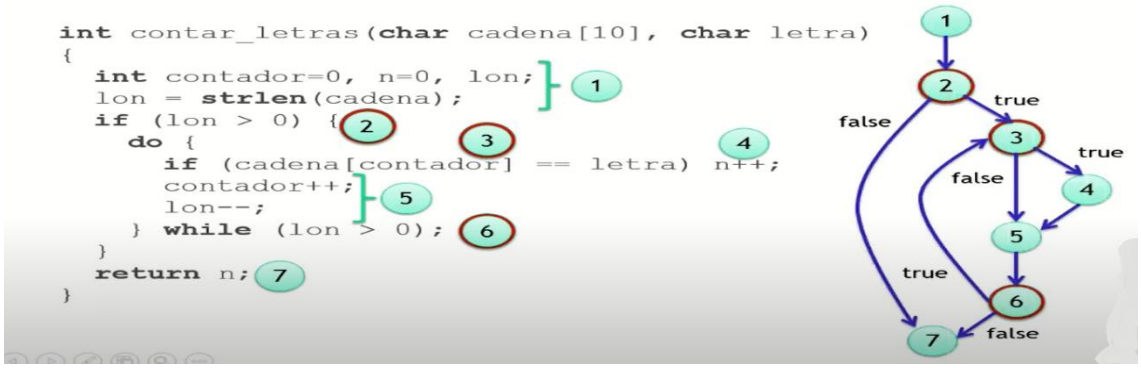
```
int contar_letras(char cadena[10], char letra)
{
  int contador=0, n=0, lon;
  lon = strlen(cadena);
  if (lon > 0) {
    do {
      if (cadena[contador] == letra) n++;
      contador++;
      lon--;
    } while (lon > 0);
  }
  return n;
}
```

Este código calcula el número de ocurrencias de una letra dentro de una cadena, que se ha pasado como parámetro de entrada

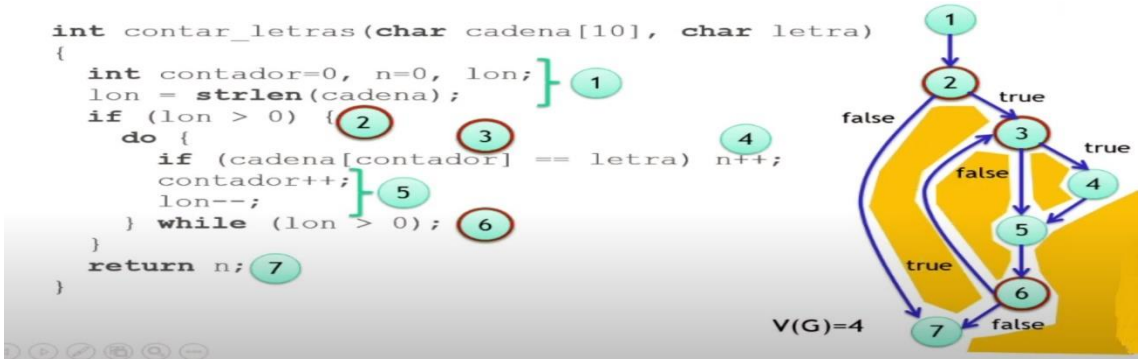
- Paso-01: Etiquetar nuestro Grafo desde el código fuente. Hemos etiquetado, como bloque atómico, de conjunto de instrucciones, ya que no hay condiciones como 1. Como 2 la primera condición, como 3 la siguiente instrucción y así sucesivamente.



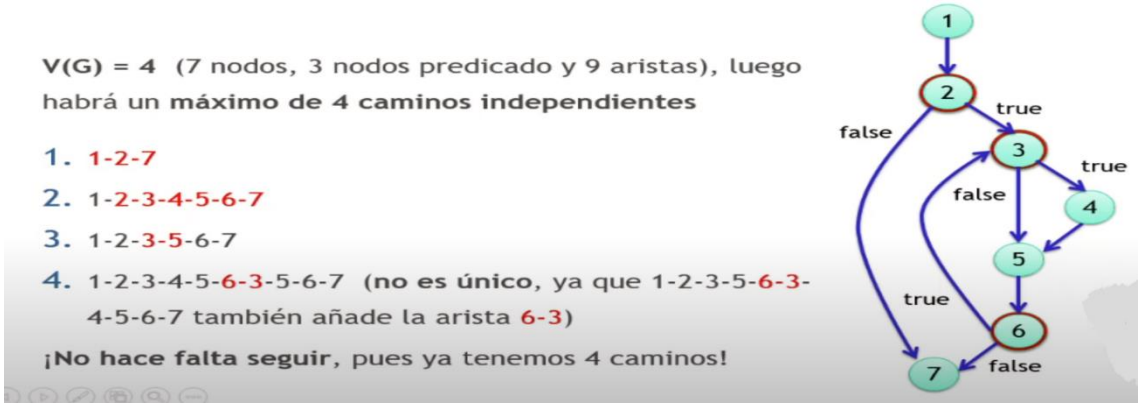
- Paso-02: Este es nuestro Grafo asociado que hay en función de las directrices que hay para movernos a partir del código fuente al Grafo, (con las condiciones, con los nodos y con los nodos predicados… Y a partir de este Grafo podemos calcular su complejidad ciclomática, (que van a ser en este caso es cuatro).



- Paso-03: (Cálculo del número máximos de caminos básicos): Para el cálculo de los caminos independientes de nuestro Grafo, hay varias formas. La primera forma es mirando nuestro Grafo Asociado calculado y pistar las áreas que tiene, y de cada área saldrá un camino independiente, (en nuestro caso hay 4 áreas). Segunda forma de calcularlo es: V(G) = (Aristas – Nodos + 2). → V(G) = (9 – 7 + 2) = 4.



- Paso-04: Una vez hallado el número máximo de caminos, (caminos independientes), en este caso 4. El número de caminos independientes es fijo, pero los caminos no tienen por qué ser los mismos. Ya que el único criterio que hay que satisfacer es que cada camino independiente añada al menos una nueva arista. Como ya hemos llegado a los 4 caminos independientes que marca la Complejidad Ciclomática, no hace falta calcular más caminos. Y hemos visitado todas las Aristas del grafo, no hace falta construir más.



6.- DESARROLLO DE PRUEBAS DE CAJA NEGRA:

- También denominadas pruebas de comportamiento.
- Consideran la función específica para la cual fue creado el producto (qué es lo que hace).
- Las pruebas se llevan a cabo sobre la interfaz del sistema.
- No es una alternativa a la prueba de caja blanca, sino que las complementa. Por lo tanto, se suelen usar en conjunto los dos tipos de técnicas.
- Conducidas por las entradas y salidas
- Prueba exhaustiva de las entradas y salidas.

6.1.- Prueba de caja negra mediante la técnica de partición equivalente y técnica de análisis de valores límites con ejemplos.

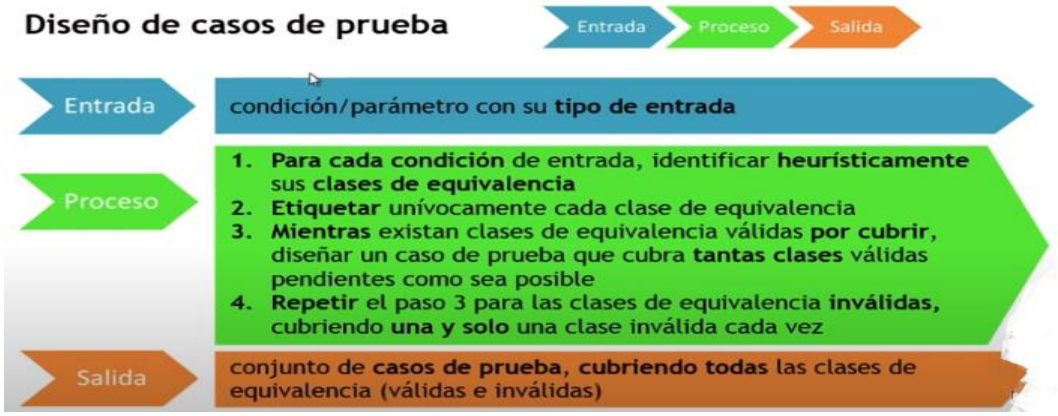
- Las pruebas de caja negra se basan en la comprobación de la funcionalidad esperada en base a su interfaz (parámetros de Entrada/Salida) del componente software.
- Probar la funcionalidad del módulo sin disponer del código fuente.
- Intentar encontrar errores de estructuras de datos, iniciación y finalización, rendimiento, interfaces de E/S, etc..
- **Técnica de partición equivalentes**:
  - Consiste en derivar casos de prueba mediante la división del dominio de entrada en clases de equivalencia, evaluando su comportamiento de un valor cualquiera representativo de dicha clase.
  - Una clase de equivalencia representa al conjunto de estados válidos o inválidos para todas las condiciones de entrada que satisfagan dicha clase.
  - Suponemos razonablemente que el resultado de éxito/fallo será el mismo para cualquier valor de esa clase.
  - Condiciones de entrada (restricciones de contenido o formato) de datos válidos y de datos inválidos.
  - Reduciendo el conjunto de casos de prueba a uno más pequeño (arriesgado), es decir, se busca descubrir clases de errores para cada clase de equivalencia identificada.

  - **Heurística de identificación de clases de equivalencia**:

Tipo de entrada	Nº clases válidas	Nº clases inválidas
Rango de valores Ej. [20..30]	1: valor en rango (25)	2: valor por debajo y otro por encima del rango (15 y 40)
Conjunto finito de valores Ej. {2,4,6,8}	1: valor en el conjunto (4)	2: valor fuera del conjunto, por debajo y por encima (1 y 10)
Condición booleana (T/F) Ej. "debe ser una letra"	1: valor evaluado a cierto ('J')	1: valor evaluado a falso ('#')
Conjunto de valores admitidos Ej. {opción1, opción2, opción3}	3 en este ejemplo: tantos como valores admitidos (opción1, opción2, opción3)	1: valor no admitido (opción4)
Clases menores Ej. "2 rangos: 0<valor<5; 10<=valor<20"	*Se divide en distintas subclases que se manejan exactamente igual que en los puntos de arriba (una para "0<valor<5" y otra para "10<=valor<20")*	

- ***Ejemplos de tipos de entrada***:
  - *Tipos de entrada de Rango:* Supongamos que disponemos de un rango de valores, por ejemplo, de 20 a 30; en este caso tendremos una clase válida que tendrá un valor válido dentro de ese rango, (por ejemplo 25, 26 o 27, también serían posibles valores). Y por otro lado tendremos dos clases inválidas, una con un valor por debajo de 20 y otra con un valor por encima de 30, por ejemplo 15 y 40, aunque también 10 y 50 podían haber sido valores válidos.
  - *Tipos de entrada de Conjunto finito de valores*: Supongamos que disponemos de un conjunto de valores, por ejemplo, 2, 4, 6 y 8; en este caso tendremos una clase válida que tendrá un valor válido dentro de ese rango, (por ejemplo 4). Y por otro lado tendremos dos clases inválidas, una con un valor por debajo de este conjunto y otra con un valor por encima de este conjunto, por ejemplo 1 y 10.
  - *Tipos de entrada de Condición Booleana:* Supongamos que disponemos de datos de entrada que es una condición booleana, que se valora a cierto (T o V) o a falso (F) en nuestro ejemplo debe ser obligatoriamente una letra, tendremos una clase válida, que es el valor con el que se evalúa a cierto, por ejemplo, la letra (‘ J ’). Y por otro lado tendremos una clase inválida, con un valor que se valora a Falso, por lo tanto, será algo que no sea una letra, por ejemplo, el símbolo (‘ # ‘).
  - *Tipos de entrada de Conjunto de Datos Admitidos:* Supongamos que los datos de entrada son un conjunto de valores admitidos, es decir, un conjunto de opciones, como, por ejemplo, la 1, la 2 y la 3; en este caso tendremos tantas clases válidas como valores admitidos, en este caso habrá 3, opc1, opc2 y opc3. Y por otro lado tendremos como clase inválida tendremos un valor no admitido, por ejemplo, una cuarta opción, opc4.
  - *Tipos de entrada de Clases Menores:* Supongamos que disponemos de unos datos de entrada que son una combinación de los valores que acabamos de ver, en este caso estamos hablando de clases menores o subclases, por ejemplo, puede ocurrir que existan dos rangos de valores, que vaya entre 0 y 5 y entre 10 y 20; en este caso tendremos que dividir en rango o clases menores, tratarlas como subclases y manejarlas exactamente como se ha visto en el resto de la tabla. Es este caso, primero trabajaríamos con el primer rango (‘ 0 < valor < 5 ‘), obteniendo un valor válido que estará dentro del rango y dos valores inválidos un valor por debajo y otro valor por encima del rango. Y a continuación haríamos exactamente lo mismo para trabajar con la otra clase, el segundo rango (‘ 10 < valor < 20 ‘), obteniendo un valor válido que estará dentro del rango y dos valores inválidos un valor por debajo y otro valor por encima del rango.

- **Diseños de caso de prueba**: Una vez que se han identificado estas clases, para construir los casos de pruebas, lo que tenemos que hacer es lo siguiente:



❖ **Técnica de Análisis de los valores límites** :

- Esta técnica sirve para extender los casos de prueba los casos de prueba que se han generado con la técnica anterior.
- Consiste en hacer hincapié de los datos o de los valores justo que está en los límites.
- La idea subyacente es que los errores suelen concentrarse no en la mitad de un rango, sino justamente en los límites operacionales (en los extremos).
- Para ello, la forma de construir las clases válidas o inválidas se calculamos de una forma muy similar, trabajando con los datos de entrada.

○ **Ejemplo** : Si tenemos en el primer tipo de entradas un rango de valores de 20 – 30, con esta técnica vamos a generar 4 clases válidas: Dos de ellas con los valores límites del rango, (en este ejemplo 20 y 30), y dos más con los valores dentro del rango, pero justo adyacentes a estos límites, (en este caso 21 y 29). Por otro lado, el número de clases inválidas es de 2, es decir con un valor justo por debajo del rango y otro justo por encima del rango, (en este ejemplo 19 y 31).
En el segundo tipo de entrada es un conjunto de valores finitos, (2, 4, 6 y 8), tendremos cuatro clases válidas. En primer lugar, con el valor mínimo y máximo, es decir, justo el que está en el límite. Y dos más con los valores adyacentes a este límite, en este caso (4 y 6). Por otro lado, tendremos dos clases de valores inválidas será de dos, con un valor justo por debajo y justo por encima del límite, (en este caso 1 y 9).

Complemento a la técnica de partición equivalente que busca explotar los **errores que se pueden producir** en los **extremos tanto de entrada como de salida** (ej. límites de rangos, estructuras, bucles, clases equivalentes, etc.)

En lugar de diseñar una clase válida con un **elemento representativo** de la partición equivalente, se **escogen** los **valores en** los **límites de la clase**

Tipo de entrada	Nº clases válidas	Nº clases inválidas
Rango de valores Ej. [20..30]	4: valor en los **límites del rango** (20, 21, 29 y 30)	2: valor **justo** por debajo y **justo** por encima del rango (19 y 31)
Conjunto finito de valores Ej. {2,4,6,8}	4: valores **mínimo y máximo** en el conjunto (2, 4, 6 y 8)	2: valor fuera del conjunto, **justo** por debajo y por encima (1 y 9)

Partiendo de los datos de entrada, como acabamos de ver, lo que hacemos es identificar heurísticamente las clases de equivalencia, tantos válidas como inválidas, las etiquetamos unívocamente y se empieza a construir casos de prueba, para cubrir todas las clases válidas. Una vez se ha acabado esto y hemos construido esos casos de pruebas se repite el mismo proceso para las clases de equivalencia inválidas, es decir, se construirá un caso de prueba para cubrir las clases inválidas. El resultado, por tanto, será el conjunto de casos de prueba que nos cubra absolutamente todas las clases de equivalencia, tantos válidas como inválidas.

8.- BIBLIOGRAFÍA:

- **Libro de Texto de Entornos de desarrollo Ed. Paraninfo**
- **Cursos de entornos de desarrollo de la web fp-informática**
- **Wikipedia**