Tema 3
Testing

JORGE BARÓN ABAD

JORGE.BARON@ADAITS.ES

3.1
Diseño y Realización de Pruebas

Testing

Testing

- Proceso que permite verificar y revelar la calidad de un producto software. Se utilizan para identificar posibles fallos de implementación, calidad o usabilidad de un programa

Testing

- En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades, donde es muy fácil que se produzca un error humano:
  - Incorrecta interpretación de los requisitos del cliente.
  - Imprecisión en la especificación de objetivos.
  - Errores en el diseño de la página web.
  - Fallos en el diseño de la base de datos.
  - Bugs en la fase de desarrollo.

Testing: Verificación y Validación

Verificación

Testing: Verificación y Validación

- **Verificación**: Proceso por el que se comprueba que el software o parte del software cumple las condiciones impuestas, los requisitos del sistema.
- ¿Estamos construyendo el producto correctamente?
- Se comprueba que el software cumple los requisitos funcionales y no funcionales de su especificación.
- El código que estamos construyendo debe estar en armonía con la especificación que hemos tomado del cliente.

Testing: Verificación y Validación

Validación

Testing: Verificación y Validación

- **Validación**: Proceso de evaluación del sistema o de uno de sus componentes, para determinar si satisface las necesidades del cliente o usuarios.

- ¿Estamos desarrollando el software correcto?

- El resultado final del desarrollo software se debe ajustar a lo que el usuario quería (sus necesidades).En la mayoría de las ocasiones el producto desarrollado no casa con la ideas del cliente, normalmente porque a éste suele faltarle capacidad técnica de expresión.

Testing: Verificación y Validación

- Un sistema puede pasar la validación, sin embargo, no pasa la verificación.
- Cumple con la especificación del usuario, con lo que él quería, cubre sus necesidades pero internamente puede adolecer de graves detalles como:
  - un precario diseño en la base de datos;
  - uso de un excesivo e innecesario número de líneas de código, por desconocer las potencialidades del lenguaje de desarrollo o de técnicas avanzadas de programación como la POO
  - uso incorrecto en la BD de instrucciones propias del lenguaje de desarrollo, en lugar de las sentencias adecuadas de SQL.

Testing: Verificación y validación

- Nunca se va a poder demostrar que el software está completamente libre de defectos.
- Mediante el testing podemos demostrar la presencia de errores, pero no la ausencia de los mismos

Testing: Verificación y validación

- Los objetivos de estas fases son:
  - Facilitar la detección temprana y corrección de los posibles errores que se hayan cometido en el proceso.
  - Fomentar y favorecer la intervención de las partes implicadas en todas las fases del proceso.
  - Proporcionar medidas de apoyo hacia el proceso para mejorar el cumplimiento de los requisitos establecidos.

Testing: Estrategia

- Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas.

- Seguiremos el Modelo en Espiral (Modelo de ciclo de vida de ciclo de vida del software, en el que las actividades se conforman en una espiral. Cada bucle o iteración representa un conjunto de actividades)

Testing: Modelo Espiral

- Las pruebas empezarían con la prueba de unidad, donde se analizaría el código implementado.
- Seguiríamos en la prueba de integración, donde se prestan atención al diseño y la construcción de la arquitectura del software.
- El siguiente paso sería la prueba de validación, donde se comprueba que el sistema construido demuestra la conformidad con las necesidades de los clientes.
- Finalmente se alcanza la prueba de sistema que verifica el funcionamiento total del software y otros elementos del sistema.

Testing: Tipos de pruebas

- En realidad no existe una clasificación concreta de pruebas. Pero si podemos determinar dos tipos de enfoques a la hora de realizar pruebas:
  - **Pruebas de caja Negra**
  - **Pruebas de caja Blanca**

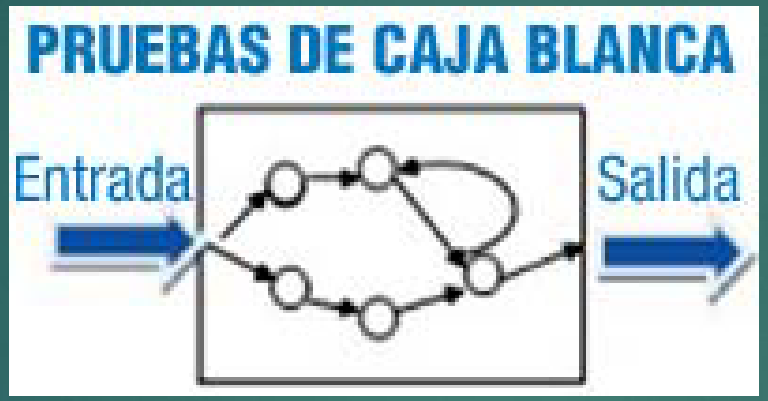
Testing: Pruebas de Caja Negra

- **Pruebas de caja Negra**: cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.

- Una prueba de tipo **Caja Negra** se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento interno del sistema. Cuando se realiza este tipo de pruebas, solo se conocen las entradas adecuadas que deberá recibir la aplicación, así como las salidas que les correspondan, pero no se conoce el proceso mediante el cual la aplicación obtiene esos resultados
- Se usa principalmente en el proceso de validación.

Testing: Pruebas de Caja Blanca

- **Prueba de la Caja Blanca**: Se prueba la aplicación desde dentro, usando su lógica de aplicación.



- Una prueba de **Caja Blanca**, va a analizar y probar directamente el código de la aplicación. Como se deriva de lo anterior, para llevar a cabo una prueba de Caja Blanca, es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas

- Se usa principalmente en el proceso de verificación.

Testing: Clasificación de pruebas

Clasificación de Pruebas	
Pruebas de Unidad o Prueba unitarias	Pruebas estructurales o de caja blanca
Pruebas de carga	Prueba funcional o de caja negra
Pruebas de estrés	Pruebas de regresión
Pruebas de estabilidad	Pruebas de seguridad
Pruebas de picos	Pruebas Alfa y Beta.
Pruebas de portabilidad	Prueba aleatorias
Pruebas de integración	Pruebas de interfaz

Testing: Pruebas de Carga

- Este es el tipo más sencillo de pruebas de rendimiento. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada.
- Esta carga puede ser el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga.
- Esta prueba puede mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación.
- Debe monitorizarse la base de datos y el servidor de aplicaciones.
- Usamos informes y gráficos para analizar la situación.
- Esta prueba puede mostrar el cuello de botella en la aplicación.

Testing: Pruebas de Carga

- Ejemplo : Analizar si el sistema soporta 1000 usuarios concurrentes(al mismo tiempo en la aplicación):
  - Primera prueba: 1 usuario sin concurrencia (esto puede servir como baseline, para comparar luego, puede ser con 1, 5, 10 o más, pero tiene que ser algo sumamente reducido para lo esperado en el sistema).
  - Segunda prueba: 200 usuarios concurrentes (o sea, el 20% de la carga esperada). Aquí ya se puede obtener muchísima información sobre qué tan difícil la vamos a tener para completar la prueba en tiempo y forma.

Testing: Pruebas de Carga

- Herramientas:
  - The grinder
  - GATLING
  - TSUNG
  - Apache Jmeter

Testing: Pruebas de estrés

- Esta prueba se utiliza normalmente para romper la aplicación.
- Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe.
- Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.
- ¿Cuál es la cantidad máxima de usuarios concurrentes que soporta el sistema con una experiencia de usuario aceptable? O sea, ¿cuál es el punto de quiebre?

Testing: Pruebas de estrés

- Ejemplo: Para este caso debemos ejecutar distintas pruebas, con distintas cantidades de usuarios, analizando si al aumentar la concurrencia no aumenta las transacciones por segundo, eso indica que llegamos al punto de quiebre, ya que se está saturando el sistema en algún punto, sin escalar (**throughput** => **tasa de transferencia efectiva**).

- Se realiza ejecutando pruebas incrementales de 0 a X, dónde X es un número de usuarios grandes, y con un intervalo grande entre cada prueba para ir acotando el límite. Por ejemple de 0, a 1000, y vamos realizando pruebas de 0-100 , 0-200, 0-300, etc.. Hasta que detectemos que haya disminuido el **throughput.**

Testing: Pruebas de estrés

- Herramientas:
  - WebLoad
  - LoadView
  - Qengine(ManageEngine)
  - CloudTest

Testing: Pruebas de estabilidad

- Esta prueba normalmente se hace para determinar si la aplicación puede aguantar una carga esperada continuada.
- Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación
- ¿Cómo funcionará el sistema luego de estar cierto tiempo ejecutando, digamos luego de estar un día entero?
- También se le llama prueba de endurance.

Testing: Pruebas de estabilidad

- Se podría ejecutar una carga constante que esté entre el 50 y el 70% de la carga soportada por el sistema en condiciones aceptables.
- Podría servir una carga menor también, todo dependerá de qué tan complejo sea preparar los datos de prueba para poder ejecutar durante muchas horas.
- Una vez terminadas las pruebas, debemos analizar para identificar otros tipos de problemas, como fuga de memoria, conexiones colgadas, etc..

Testing: Pruebas de estabilidad

- Herramientas:
  - Performance Tester
  - Httperf
  - OpenSTA
  - WAPT

Testing: Pruebas de picos

- La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan, como cuando tiene cambios drásticos en su carga.

- Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados .

- Si mi sistema en régimen normal funciona adecuadamente y viene un pico de estrés (la casuística hace que en un mismo momento coincidan muchas más peticiones que lo normal, ante lo cual sé que los tiempos de respuesta quizá empeoran por debajo de lo aceptable), entonces ¿qué tan rápido se recupera el sistema?

Testinsg: Pruebas de picos

- La idea es ver qué pasa cuando hay un pico, qué tanto le cuesta al sistema recuperarse. Si hay un pico entonces ¿el sistema me queda colgado? ¿a los 10 segundos se recupera? ¿a las dos horas se recupera? ¿o qué?

- Es necesario ya conocer cuál es el punto de quiebre del sistema, para poder preparar una prueba que esté por debajo de ese umbral, y generar un pico subiendo la carga por un período de un minuto por ejemplo, y luego bajarla.

- El enfoque incremental que se puede aplicar aquí es en el pico en sí. Se podría comenzar experimentando con picos pequeños (de corta duración o poca carga) y luego estudiar cómo reacciona el sistema ante picos mayores.

- En cualquier caso, esto es algo que se debe modelar en base a un estudio de los comportamientos de los usuarios, especialmente en base a los access logs que se posean.

Testinsg: Pruebas de picos

- Herramientas:
  - Agile Load
  - Acutest

Testing:
Pruebas de Portabilidad

- Las pruebas de portabilidad sirven para conocer la facilidad que tiene un software de adaptarse de un entorno a otro.
- Un software bien diseño debe ser capaz de funcionar en la gran mayoría de plataformas, independientemente de su versión.
- Estas pruebas deben aplicarse de forma iterativa e incremental, porque cada paso del tiempo, el número de plataformas y versiones de las mismas aumentas.
- Por esa razón debemos hacer estás pruebas lo más automatizada posibles.

Testing:
Pruebas de Portabilidad

- Los objetivos de estas pruebas son:
  - Determinar los requisitos mínimos:
    - Ram y espacio de disco
    - Procesador
    - Resolución del monitor
    - Sistema operativo
    - Navegador
  - Ayuda a encontrar problemas de integración, no detectados al aplicarse las pruebas de integración en un solo entorno.
  - Ayuda a determinar cuando lanzar un software o el alcance del mismo.

Testing:
Pruebas de Portabilidad

- Ejemplos:
  - Nuestra aplicación Web debe usarse en una empresa dónde se utiliza, distintos dispositivos de diversa resolución con los siguientes sistemas operativos: Android, iOS, MacOS, Windows y Ubuntu.
  - Antes de lanzar nuestro producto en dicha empresa debemos probar el correcto funcionamiento en todas esas plataformas de nuestra aplicación.
  - No obstante debemos tener en cuenta, que dentro de unos meses todas esas plataformas pueden actualizar su versión, por esa razón debemos, tener las pruebas preparadas para verificar el correcto funcionamiento de nuestro software en las nuevas versiones.

Testing:
Pruebas de Portabilidad

- Herramientas:
  - Selenium
  - Docker

Testing: Pruebas de seguridad

- Intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán de accesos impropios por parte de piratas informáticos.
- Deben seguirse los siguientes pasos:
  - Recoger tanta información como sea posible sobre la aplicación.
  - Análisis sobre la infraestructura o la topología de la arquitectura.
  - Comprobar el sistema de autenticación.
  - Inspeccionar la gestión de sesiones.
  - Entender como funciona los accesos a los recursos de la plataforma.
  - Realizar fallos en la lógica de negocio del sistema, para ver como se comporta. Por ejemplo saltarte un paso.
  - Detectar la falta de validación de todos los datos de entrada.
  - Planificar ataques DoS y DDoS.

Testing: Pruebas de seguridad

- Herramientas:
  - Vega
  - ZED Attack Proxy (ZAP)
  - Wapiti
  - W3af
  - Iron Was
  - Google Nogotofail
  - BeEF (Browser Exploitation Framework)

Testing: Pruebas Alfa y Beta.

- En estas pruebas, se realizan después de haber realizado todas las demás pruebas e involucramos a personas externas, ajenas al desarrollo de la aplicación, para usar la aplicación y probarla.
- Las pruebas alfa se llevan a cabo en un entorno controlado, pero por un cliente.
- Las pruebas beta se realizar en una aplicación en vivo del software en un entorno que no puede ser controlado por el desarrollador
- Son pruebas de validación, porque sirven para evaluar la satisfacción del cliente o usuario. En este caso importa la funcionalidad de la aplicación, no como esta hecha.

Testing: Pruebas de Interfaz

- Estas pruebas se revisan los aspectos gráficos de la web, para determinar si el despliegue de la aplicación es correcto.
- Se comprueba que la página web se adapte en todos los dispositivos correctamente (Responsive).
- Se realizan pruebas de usabilidad para comprobar si es fácil navegar por ella y realizar las acciones que deseamos.

Testing: Pruebas de Interfaz

- En estas pruebas debemos usar diversas herramientas para validar:
  - Colores usados.
  - Fuente de texto.
  - Claridad del texto.
  - Imágenes usadas.
  - Posición de los botones.
  - Navegación por la web.
  - Accesibilidad.

Testing: Pruebas de Interfaz

- Herramientas:
  - Validador HTML, CSS y Accesibilidad.
  - Selenium

Testing: Pruebas unitarias

- Con ella se va a probar el correcto funcionamiento de un módulo de código .
- Características:
  - **Automatizable** :No debería requerirse una intervención manual. Esto es especialmente útil para integración continua.
  - **Completas** :Deben cubrir la mayor cantidad de código.
  - **Repetibles o Reutilizables:** No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para integración continua.
  - **Independientes** :La ejecución de una prueba no debe afectar a la ejecución de otra.
  - **Profesionales**: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.
  - **Rápidas de crear:** no debe llevar más de 5 minutos en ser creadas, están diseñados para hacer que el trabajo sea más rápido

Testing: Pruebas unitarias

- Ventajas:
  - **Proporciona un trabajo ágil**: Como procedimiento ágil que es, te permite poder detectar los errores a tiempo, de forma que puedas reescribir el código o corregir errores sin necesidad de tener que volver al principio y rehacer el trabajo. Puesto que las pequeñas se van haciendo periódicamente y en pequeños packs. Disminuyendo el tiempo y el coste.
  - **Calidad del código:** Al realizar pruebas continuamente y detectar los errores, cuando el código está terminado, es un código limpio y de calidad.
  - **Detectar errores rápido:** A diferencias de otros procesos, los tests unitarios nos permiten detectar los errores rápidamente, analizamos el código por partes, haciendo pequeñas pruebas y de manera periódica, además, las pruebas se pueden realizar las veces que hagan falta hasta obtener el resultado óptimo.

Testing: Pruebas unitarias

- Ventajas:
  - **Facilita los cambios y favorece la integración**: Los tests unitarios, nos permiten modificar partes del código sin afectar al conjunto, simplemente para poder solucionar bugs que nos encontramos por el camino. Los tests unitarios, al estar desglosados en bloques individuales permiten la integración de nuevas aportaciones para hacer un código más complejo o actualizarlo en función de lo que el cliente demande.
  - **Proporciona información**: Gracias al continuo flujo de información y la superación de errores, se puede recopilar gran cantidad de información para evitar bugs futuros.
  - **Proceso debugging**: Los Tests unitarios ayudan en el proceso de debugging. Cuando se encuentra un error o bug en el código, solo es necesario desglosar el trozo de código testeado. Esta es uno de los motivos principales por los que los tests unitarios se hacen en pequeños trozos de código, simplifica mucho la tarea de resolver problemas.

Testing: Pruebas unitarias

- Ventajas:
  - **El diseño**: Si primero se crean los tests, es mucho más fácil saber con anterioridad cómo debemos enfocar el diseño y ver qué necesidades debemos cumplir. Testeando una pieza del código, también puedes saber que requisitos debe cumplir, y por eso mismo te será mucho más fácil llegar a una cohesión entre el código y el diseño.
  - **Reduce el coste**: Partiendo de la base que los errores se detectan a tiempo, lo cual implica tener que escribir menos código, poder diseñar a la vez que se crea y optimizar los tiempos de entrega, vemos una clara relación con una reducción económica.

Testing: Pruebas unitarias

- Desventaja:
  - Las pruebas unitarias por sí solas, no son perfectas, puesto que comprueban el código en pequeños grupos, pero no la integración total del mismo. Para ver si hay errores de integración es necesario realizar otro tipo de pruebas de software conjuntas y de esta manera comprobar la efectividad total del código.

Testing: Pruebas integración

- Esta prueba, se realiza tras haber realizado y superado todas las pruebas unitarias.
- El siguiente paso es integrar todos los componentes, módulos de código probados de forma independiente, en un entorno dónde se interconecten entre sí.
- Se utiliza para construir la arquitectura del software y determinar el diseño.

Testing: Pruebas de regresión

- Estas pruebas se utilizan para un bug que ya ha sido detectado y se ha corregido, con el objetivo de verificar que no vuelve a ocurrir dicho fallo.
- Debe aplicarse dichas pruebas tanto en el componente modificado, como en aquellos con los que se integran.
- Se utilizan otros tipos de pruebas, como funcionales y no funcionales.
- Es conveniente que dichas pruebas seán automatizadas, porque deben repetirse constantemente.

Testing: Pruebas funcionales (Caja Negra)

- Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.

- Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?

Testing: Tipos de Prueba funcionales

- Destacamos tres tipos de pruebas funcionales:
  - Particiones Equivalentes
  - Análisis de valores límites
  - Pruebas aleatorias

Testing: Particiones equivalentes

- La idea es considerar el menor número posible de casos de pruebas, para ello, cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes.

- Lo que se pretende, es crear un conjunto de clases de equivalencia, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.

Testing: Particiones equivalentes

- Primeros debemos identificar las clases de equivalencia. Para ello tomamos las condiciones de valores de entrada y salida y se divide en 1 o más grupo. Estos grupos se clasifican en válidos y no válidos.
- **Ejemplos**:
- Rango de Valores: Si una condición de entrada especifica un rango de valores (rango entre 1 y 99).

Clases validas	Clase invalidas
1<número <99	numero < 1 número > 99

Testing: Particiones equivalentes

- **Número de Valores:** Si una condición de entrada especifica el numero de valores (1 a 6 propietarios por automóvil)

Clases validas	Clase invalidas
1<propietarios<6	no hay propietarios
	hay mas de 6 propietario

- **Conjunto de Valores**: Si una condición de entrada especifica un conjunto de valores y existen razones para creer que el programa los trata distintos (vehículo puede ser: ómnibus, camión, taxi, moto)

Clases validas	Clase invalidas
Uno por cada uno	Todos los que no son esos, por ejemplo: trailer

Testing: Particiones equivalentes

- **Debe ser**: Si una condición de entrada especifica un "debe ser" (el primer carácter debe ser un dígito)

Clases validas	Clase invalidas
Primer carácter un dígito	primer carácter distinto de dígito

Testing: Particiones equivalentes

- Creamos casos de Prueba:

- Por cada dato de entrada y salida elegimos un valor perteneciente a una clase de equivalencia.

- Un caso de prueba está formado por un dato de prueba para cada dato de entrada y salida (resultado esperado).

Testing: Casos de Prueba

- Ejemplo: resultado sumarParImpar(NumeroPar, NumeroImpar);
- Casos de entrada NumeroPar:
  - Validos => 2,4,6
  - No Válidos => 1,3,5
- Casos de entrada NumeroImpar:
  - Validos => 1,3,5
  - No Válidos => 8,10,6
- Casos de la salida resultado:
  - Validos => 3, 9, 12
  - Invalidos => 2, 0 , -1

Testing: Casos de Prueba

- El objetivo es probar todas las clases válidas y no válidas al menos una vez cada una de ellas
- Cada caso debe cubrir el máximo numero de entradas
- En un caso de prueba pueden aparecer varias clases de entrada válidas
- En un caso de prueba solamente puede aparecer una entrada no válida. El criterio para las clases no válidas impide que los errores se enmascaren mutuamente
- Se debe especificar el resultado esperado.

Testing: Análisis de valores límites

- Esta técnica se basa en que la mayoría de fallos se producen en los extremos de los posibles valores de entrada.
- Por lo tanto debemos hacer un análisis de los resultados de la operación cuando se usan estos valores límites.
- Se suele aplicar junto a la técnica de clase de equivalencia.

Testing: Análisis de valores límites

- Suelen aplicarse mínimo 4 pruebas:
  - El valor en el límite superior
  - El valor en el límite superior +1
  - El valor en el límite inferior
  - El valor en el límite inferior -1

Testing: Análisis de valores límites

- Un ejemplo:
  - Función para ayuda de alquiler entre 18 y 35 años.
  - Clases de equivalencia:
    - Inválidas:
      - Personas Menores a 18 años
      - Personas Mayores a 35 años
    - Válidas:
      - Personas Entre 18 y 35 años.

Testing: Análisis de valores límites

- Debemos probar lo siguiente:
  - Personas entre 18 y 35 años:
    - Probamos con 18 => Deberá dar un resultado correcto
    - Probamos con 17 => Deberá dar un resultado erróneo
    - Probamos con 35 => Deberá dar un resultado correcto
    - Probamos con 36 => Deberá dar un resultado erróneo

Testing: Pruebas Aleatorias

- Esta prueba consiste en generar entradas aleatorias.
- Se suelen usar herramientas automatizadas que generen valores al azar, tanto valores con resultado erróneos como correctos.

Testing: Pruebas estructurales (Caja Blanca)

- Estas pruebas se centran en todos los caminos que puede recorrer nuestro software.
- Estás pruebas están ligadas a nuestro código fuente.
- Se verifica la estructura interna de cada uno de los componentes de la aplicación, independientemente de la funcionalidad establecida para ese componente.
- El objetivo principal es comprobar que se van a ejecutar todas las instrucciones del software, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer,etc..

Testing: Pruebas estructurales (Caja Blanca)

- Al estar diseñadas este tipo de pruebas a una implementación concreta, si el código se modifica, la prueba debe modificarse.

- Normalmente se aplican a pruebas unitarias.

Testing: Pruebas estructurales (Caja Blanca)

- Técnicas:
  - **Cobertura de sentencias**
  - **Cobertura de decisiones**
  - **Cobertura de condiciones**
  - **Cobertura de caminos**
  - **Cobertura del camino de prueba**

Testing: Cobertura de sentencias

- Cada sentencia del programa debe ejecutarse al menos una vez.
- Por ejemplo:
  - **funcion comparaNumeros(x, condicion1, condicion2,condicion3){**

```
lf(condicion1){
        x++;
}
if(condicion2){
        x++;
}
if(condicion3){
        x++;
}
return x;
```

}

- Debemos hacer una prueba en la que la condición1, condicion2 y condicion3 sean verdad.

Testing: Cobertura de decisiones

- Se trata de crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez a cierto y otra a falso.
- **funcion esMayorDeEdad(a){**

```
    if(a >= 18){
        return true;
    }else{
        return false;
    }
}
```

- Debemos hacer dos pruebas:
  - Una en la que a es menor a 18
  - Otra prueba en la que a es mayor que 18.
- Con esta técnica siempre aplicaremos 2 Pruebas por cada condición.

Testing: Cobertura de condiciones

- Se trata de crear los suficientes casos de prueba para que cada elemento de una condición, se evalúe al menos una vez a falso y otra a verdadero.
- **funcion sumarPares(a,b){**

```
if(a %2 == 0 && b %2 == 0){
    return a+b;
}else{
    return false;
}
}
```

- Debemos hacer 4 pruebas:
  - A es par y B es par => a+b
  - A es par y B es impar => 0
  - A es impar y B es par => 0
  - A es impar y B es impar => 0

Testing: Cobertura de caminos

- Establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final.
- La ejecución de este conjunto de sentencias, se conoce como camino. Como el número de caminos que puede tener una aplicación, puede ser muy grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.

Testing: Cobertura de caminos

```
Funcion calcular(a,b){
    int resultado = 0;
    If(a > b){
        If(a % 2 == 0 || b %2 == 0){
            If(b % 2 == 0 ){
                resultado = b;
            }
        }
    }else{
        resultado =  a;
    }
    return resultado;
}
```

Testing: Cobertura de caminos

- Pruebas a hacer:
  - Prueba a mayor que b
  - Prueba a par mayor que b par
  - Prueba a menor que b.

Testing: Cobertura de caminos de prueba

- Se pueden realizar dos variantes, una indica que cada bucle se debe ejecutar sólo una vez, ya que hacerlo más veces no aumenta la efectividad de la prueba y otra que recomienda que se pruebe cada bucle tres veces: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces.

Testing: Cobertura de caminos de prueba

```
funcion prueba(a){
    int resultado = 0;
    for(int i = 0;i < a;i++){
        resultado +=a;
     }
    return resultado;
}
```

Debemos hacer 3 pruebas:

- a vale 0
- a vale 1
- a vale 5


THE
END