Unidad 9 - Técnicas Basadas En Estructura

Caja blanca

Daremos un paso adelante al sumergirnos en las técnicas de pruebas de software basadas en la estructura del software. Estas técnicas son fundamentales para evaluar la calidad y el rendimiento de los programas informáticos.
El conocimiento de las técnicas de pruebas de software es esencial para cualquier profesional dedicado al testing de software, estas técnicas nos brindan herramientas para identificar defectos, evaluar la funcionalidad y la seguridad del software, y garantizar que cumpla con los requisitos y estándares necesarios.

razones fundamentales por las cuales estos temas son cruciales:

- Las técnicas de pruebas permiten identificar y corregir defectos y errores en el software antes de su implementación o lanzamiento. Esto contribuye significativamente a mejorar la calidad y confiabilidad del software, evitando posibles problemas y fallos en el futuro.
- Las técnicas de pruebas son esenciales para verificar que el software cumple con los requisitos específicos del cliente o del proyecto, así como con los estándares de calidad establecidos en la industria.
- Al aplicar técnicas de pruebas de manera efectiva, se pueden identificar y resolver problemas de manera temprana, lo que ahorra tiempo y recursos en comparación con la detección de problemas en etapas más avanzadas del desarrollo.
- En el contexto actual de la ciberseguridad, las técnicas de pruebas de seguridad son esenciales para identificar vulnerabilidades y brechas de seguridad en el software. Esto es crítico para proteger los datos y la privacidad de los usuarios.
- Las pruebas de usabilidad y pruebas de aceptación del usuario son técnicas que permiten evaluar la experiencia del usuario final. Esto asegura que el software sea intuitivo, eficiente y cumpla con las expectativas de los usuarios.
- Detectar y solucionar problemas de software en etapas tempranas del desarrollo es más económico que hacerlo después de que el software está en producción. Las técnicas de pruebas ayudan a evitar costosos retrabajos y reparaciones posteriores.

Durante esta unidad, exploraremos a fondo las técnicas de pruebas de software, centrándonos en las metodologías basadas en la especificación y en la estructura. Estas técnicas son fundamentales para evaluar la calidad, confiabilidad y funcionalidad de un software.

Explorando las técnicas de caja blanca en pruebas de software

Las técnicas de caja blanca, también conocidas como pruebas de caja blanca o pruebas estructurales, se centran en examinar el código fuente y la estructura interna de un programa.

Paso 1: Identificar el Código a Probar

El primer paso en las técnicas de caja blanca es identificar qué partes del código se deben probar. Esto implica examinar el diseño del programa y seleccionar las unidades o módulos que requieren pruebas exhaustivas. Por ejemplo, en una aplicación web, podríamos centrarnos en las funciones que manejan el inicio de sesión de usuario.

Paso 2: Diseñar Casos de Prueba Basados en el Código

Diseñamos casos de prueba que se basen en la estructura interna del código. Esto implica crear pruebas específicas para verificar el comportamiento de las funciones, las rutas de ejecución y las condiciones lógicas. Por ejemplo, podríamos diseñar un caso de prueba que verifique si un usuario puede iniciar sesión con una contraseña válida.

Paso 3: Ejecutar Pruebas y Analizar Cobertura de Código

ejecutamos las pruebas diseñadas y analizamos la cobertura de código. La cobertura de código nos indica qué partes del código se han ejecutado durante las pruebas y cuáles no. El objetivo es alcanzar una cobertura cercana al 100% para garantizar que todas las rutas de ejecución se hayan probado. Si encontramos partes del código que no se ejecutan durante las pruebas, puede ser un indicio de áreas no probadas o posibles problemas.

Paso 4: Identificar y Corregir Defectos

Durante la ejecución de las pruebas de caja blanca, es posible que identifiquemos defectos, errores de lógica o problemas de rendimiento en el código. Estos deben documentarse y corregirse de manera oportuna. El proceso de identificación y corrección de defectos es esencial para mejorar la calidad del software.

Paso 5: Repetir el Proceso

Las técnicas de caja blanca son iterativas. Después de identificar y corregir los defectos, repetimos el proceso de diseño y ejecución de pruebas para asegurarnos de que el código esté en su mejor forma. Este ciclo puede repetirse varias veces hasta que estemos seguros de que el software es robusto y confiable.

Ejemplo Práctico: Pruebas de una Calculadora

Imaginemos que estamos trabajando en el desarrollo de una aplicación de calculadora para dispositivos móviles. Esta calculadora tiene funciones básicas como suma, resta, multiplicación y división. Queremos asegurarnos de que nuestra calculadora funcione correctamente antes de lanzarla al mercado.

Paso 1: Identificación del Código a Probar

En este caso, las áreas clave a probar son las funciones de cálculo, es decir, las funciones de suma, resta, multiplicación y división. Estas funciones son críticas para el funcionamiento de la calculadora.

Paso 2: Diseño de Casos de Prueba Basados en el Código

Para la función de suma, diseñamos casos de prueba que incluyen diferentes escenarios, como:

- Sumar números positivos.
- Sumar números negativos.
- Sumar cero a un número.
- Sumar números decimales.

Cada caso de prueba se basa en la estructura interna de la función de suma y verifica que los resultados sean correctos.

Paso 3: Ejecución de Pruebas y Análisis de Cobertura de Código

Ejecutamos nuestras pruebas de caja blanca para la función de suma y analizamos la cobertura de código. Durante las pruebas, ingresamos diferentes valores y verificamos que los resultados sean coherentes con las expectativas. Si encontramos algún error, lo documentamos y lo corregimos.

Paso 4: Identificación y Corrección de Defectos

Durante la ejecución de pruebas, podríamos identificar un defecto en la función de suma que hace que la calculadora arroje resultados incorrectos cuando se suman números grandes. Identificamos este defecto y trabajamos en su corrección.

Paso 5: Repetición del Proceso

Repetimos este proceso para las funciones de resta, multiplicación y división de nuestra calculadora. Cada función se somete a pruebas exhaustivas de caja blanca para garantizar su precisión.

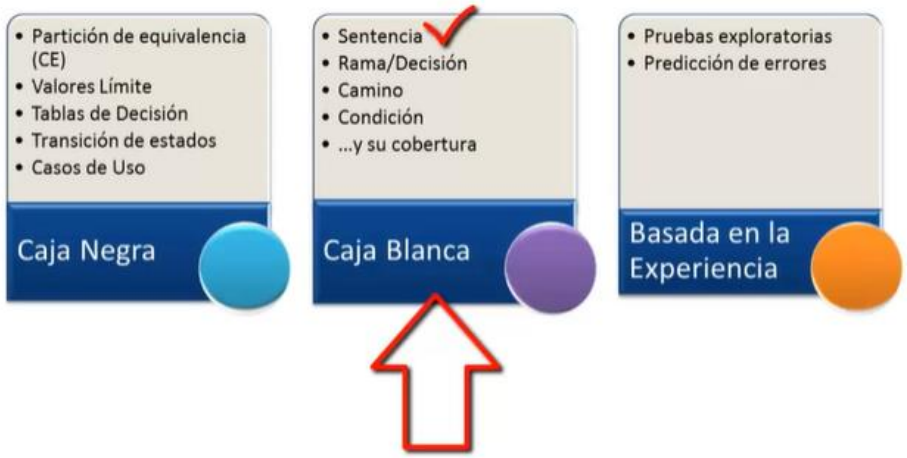
Aprender técnicas de caja blanca es crucial en el desarrollo de software por varias razones:

- Identificación Temprana de Defectos: Las técnicas de caja blanca permiten identificar y corregir defectos en el código fuente antes de que lleguen a los usuarios finales. Esto reduce los costos y problemas asociados con la corrección de errores en etapas avanzadas del desarrollo.
- Mejora de la Calidad: Al comprender cómo funciona internamente el software, podemos diseñar pruebas más efectivas que cubran todas las rutas de ejecución posibles, lo que conduce a un software de mayor calidad y confiabilidad.
- Optimización del Rendimiento: Las pruebas de caja blanca también pueden ayudar a identificar áreas de código que pueden optimizarse para mejorar el rendimiento del software.
- Cumplimiento de Requisitos: Al realizar pruebas de caja blanca, podemos verificar que el software cumpla con los requisitos funcionales y las especificaciones de diseño.

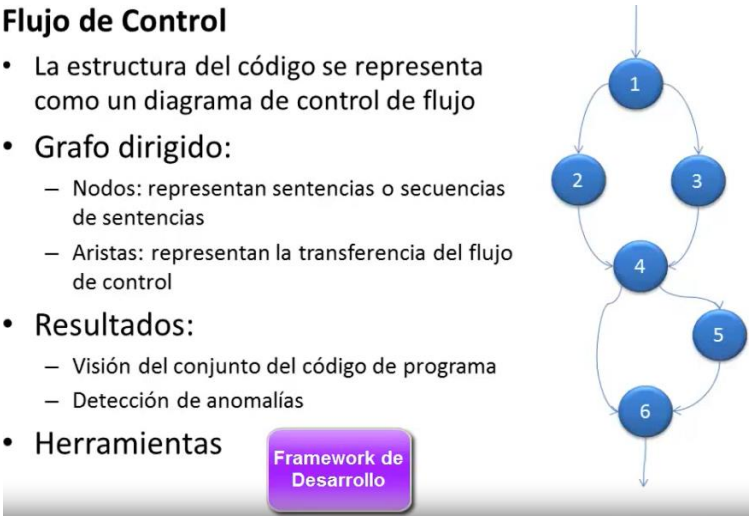
Las técnicas de caja blanca son una parte fundamental de las pruebas de software, ya que nos permiten profundizar en la estructura interna del código y garantizar su calidad. Al comprender cómo funcionan las técnicas de caja blanca y seguir un proceso sólido, podemos identificar y corregir defectos de manera efectiva, lo que resulta en software más confiable y robusto.

En resumen, aprender y aplicar técnicas de caja blanca en pruebas de software es esencial para garantizar que los programas sean confiables, eficientes y cumplan con los requisitos del cliente. Además, ayuda a evitar problemas costosos en etapas posteriores del desarrollo.

Técnica de diseños de pruebas **VIDEO 01**



- **La caja blanca se aplica cuando el probador, tiene acceso al código (desarrollador).**
- **Los casos de prueba están centrados en probar la estructura interna del código.**
- **Se debe medir la cobertura del código (Medida en % que indica cuanto del código ha sido ejecutado por los casos de prueba)**
- **Tanto la cobertura de sentencia y decisión están basadas en el flujo de control**



El flujo de control: es la representación gráfica de los segmentos de programa Está compuesto por los nodos y aristas (%). Resultados: que se logran a través de las herramientas de software. Mide la cobertura y detecta anomalías.
El flujo de control se obtiene a través de las mismas herramientas que utiliza el desarrollador para lograr su código.

COBERTURA DE SENTENCIA

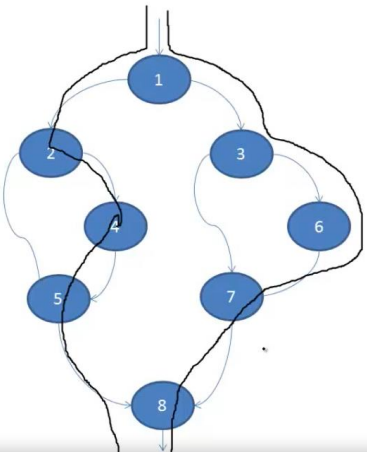
- Centrado en la cobertura de sentencias del código
  - Qué casos de prueba son necesarios con el objeto de ejecutar todas o un % de las sentencias de un código existente?
- Basados en el flujo de control
  - Las sentencias están representadas por nodos
  - El flujo de control está representado por flechas

- Cobertura de Sentencia ➔ Cs
  - Cs = Número de sentencias visitadas * 100%
    Número total de sentencias

El objetivo de esta técnica (criterio de salida) es lograr una cobertura específica de todas las sentencias

ej =

Cuántos casos de prueba se requieren para lograr una cobertura de sentencia del 100%?



Se requieren 2 casos de pruebas, recordar que el objetivo de cobertura de sentencia es visitar todos los nodos por lo menos 1 vez

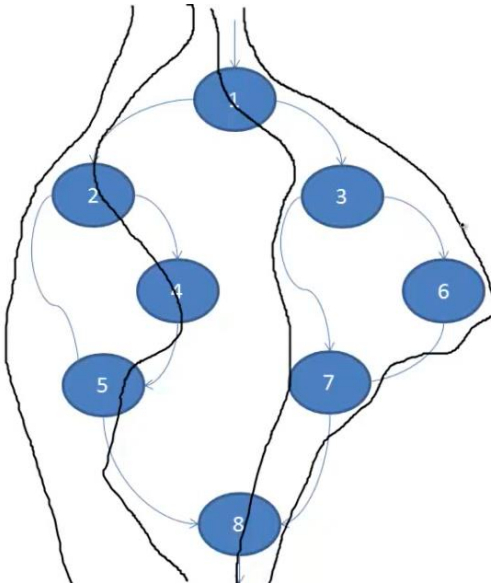
COBERTURA DE DECISIÓN

- En lugar de los nodos, las aristas
  - Todas las aristas del diagrama de flujo de control tienen que ser visitadas por lo menos una vez
  - Cobertura de Rama
- Basados en el flujo de control
- Cobertura de Decisión/Rama ➔ Cr
  - Cr = Número de ramas visitadas * 100%
    Número total de ramas

El objetivo de esta técnica (criterio de salida) es lograr la cobertura de un % específico de todas las decisiones

Ej=

Cuántos casos de prueba se requieren para lograr una cobertura de decisión del 100%?

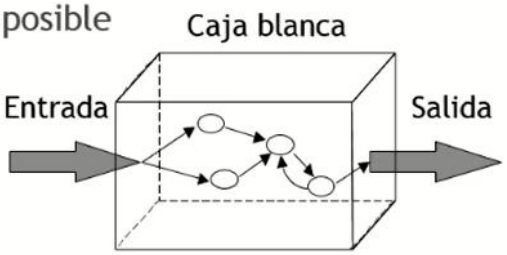


Se requieren 4 casos de prueba para cubrir el 100%, esta cobertura es más exhaustiva que la de sentencia.

Prueba de caja Blanca **video 02**

Técnica del camino básico

En las pruebas de **caja blanca** se dispone del código fuente y se intentan analizar tantos fragmentos como sea posible



A priori **se puede pensar** que es viable la verificación de la totalidad de los caminos/alternativas del código, pero realmente esto **no es posible**

EJ=

```
while ((a<100) and (b<50))
{
  hacer algo que modifique "a" o "b"
}
```

¡Considerando dominios de números enteros 1 y 100 tenemos 100^2=10000 posibilidades!

No es posible probar **todos** los caminos de ejecución distintos

Pero **sí** lo es verificar al menos los **caminos independientes** y **confeccionar** casos de prueba que garanticen que se verifican dichos caminos de ejecución

Tecnica del camino básico

Consiste en derivar **casos de prueba** a partir de un conjunto dado de **caminos independientes** por los que circula el flujo del programa

*"Un camino independiente es el que **introduce por lo menos** una **nueva secuencia** (**arista** en el grafo de flujo) que no estaba considerada en el conjunto de caminos independientes ya calculados"*

El número de caminos independientes viene **acotado** por la **complejidad ciclomática** (V(G)) del grafo de flujo asociado al código

Diseño de casos de prueba

Entrada → Proceso → Salida

Entrada: fragmento de **código**, su **grafo de flujo** asociado y **V(G)**

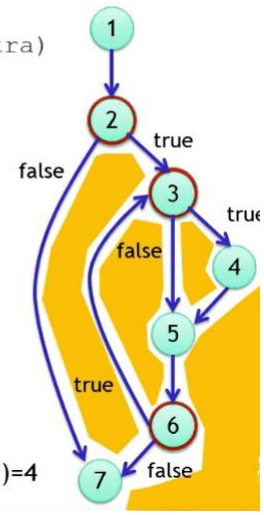
Proceso: se cubrirán **todos** los caminos básicos, **garantizando** el paso por todas las sentencias (**nodos**), secuencias/pasos (**aristas**) y prueba de condiciones en sus dos posibilidades booleanas (**nodos predicado**)

Salida: conjunto de **casos de prueba, uno por cada** camino generado

Ej=

Técnica del camino básico. Ejemplo

```
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;
}
```



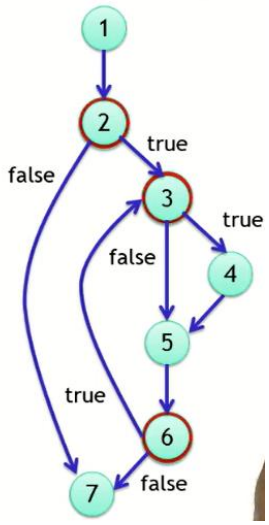
V(G)= complejidad ciclomátca

Los números pintados en rojos son las aristas.

V(G) = 4 (7 nodos, 3 nodos predicado y 9 aristas), luego habrá un **máximo de 4 caminos independientes**

1. 1-2-7
2. 1-2-3-4-5-6-7
3. 1-2-3-5-6-7
4. 1-2-3-4-5-6-3-5-6-7 (**no es único**, ya que 1-2-3-5-6-3-4-5-6-7 también añade la arista 6-3)

¡**No hace falta seguir**, pues ya tenemos 4 caminos!



Casos de prueba

Número	Camino independiente	*cadena*	*letra*	*n*
1	1-2-7	""	'a'	0
2	1-2-3-4-5-6-7	"a"	'a'	1
3	1-2-3-5-6-7	"b"	'a'	0
4	1-2-3-4-5-6-3-5-6-7	"ab"	'a'	1

Si al ejecutar estos casos de prueba el valor resultante de *n* (nº de ocurrencias de *letra* en *cadena*) no coincide con este habremos descubierto un error

Conclusiones

Gracias a la **técnica del camino básico** podemos garantizar que estamos recorriendo y verificando:

- ☐ todas las **aristas** del grafo (caminos independientes)
- ☐ todas las **condiciones**, tanto en su evaluación true como false - nótese que **no se garantiza** una cobertura de condición múltiple (ej. si hay condiciones múltiples **no** se **analiza toda** la tabla de verdad)

Aun así, **no podemos garantizar** que el código esté **libre de errores**

Cobertura de sentencia **Video 04**

Cobertura de sentencia

- También conocido como Line coverage o Segment coverage.
- Prueba las sentencias potencialmente ejecutables.
- La cobertura se mide en base a la cantida de de sentencias probadas vs el total de sentencias.

Estas pruebas se enfocan en el código que ya existe.

Ejemplo

```
READ X
READ Y
I F X>Y THEN Z = 0
ENDIF
```

En este escenario solo necesitaría un caso de prueba. Asegurarse de usar datos correctos de entrada en la cobertura de pruebas.

Ejemplo 2

```
READ X
READ Y
Z = X + 2 * Y
IF Z > 50 THEN
PRINT large Z
ENDIF
```

- Casos de prueba:
- Test A: X= 2, Y = 3 ? El valor de Z = 8
- Test B: X =0, Y = 25 ? El valor de Z = 50
- Test C: X =47, Y = 1 ? El valor de Z = 49
- Cobertura: 83% (5/6 statements cubiertos)
- Test D: X = 20, Y = 25 ? El valor de Z = 70
- Cobertura: 100% (6/6 statements cubiertos)

Ejemplo 3

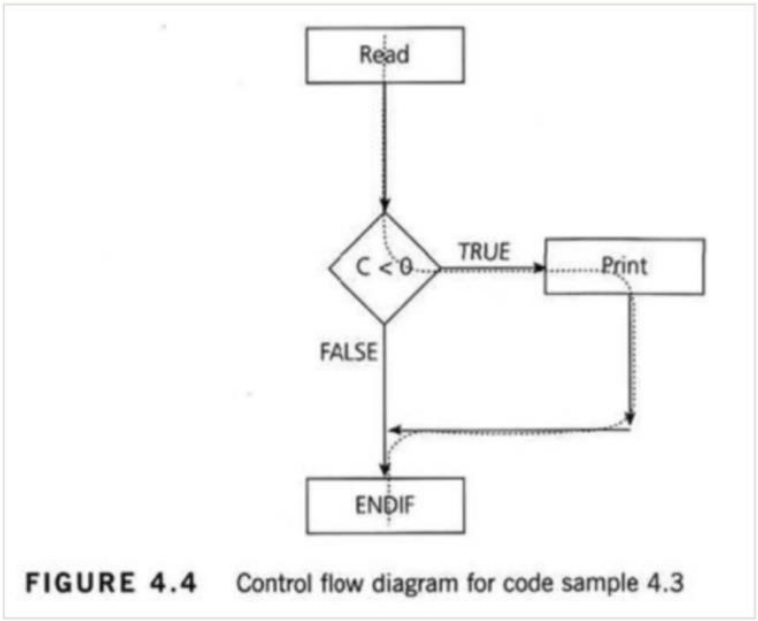


FIGURE 4.4 Control flow diagram for code sample 4.3

En este caso con un solo caso de prueba, la sentencia es de lectura, condicional, imprimir y de fin.
El caso de minios de prueba a cubrir del 100% es 1. Con un solo caso de prueba podemos cubrir todas las sentencias.

Una particularidad de la cobertura de sentencias, no prueba en las condicionales la bifurcación si es false si no hay sentencias.

Cobertura de decisión **Video 05**

Cobertura de decisión

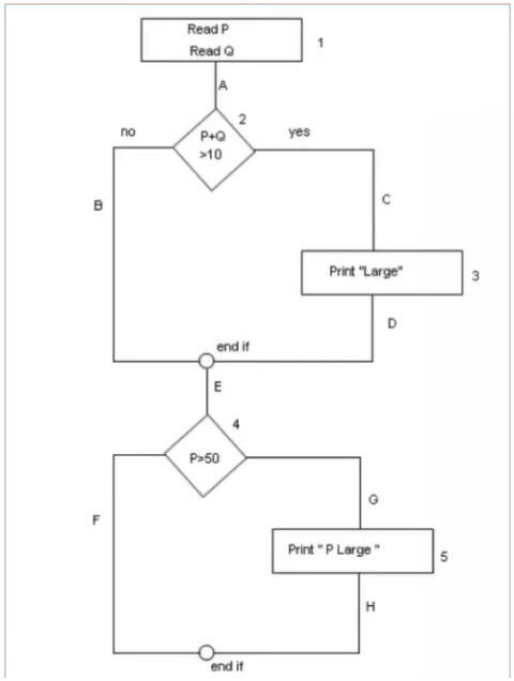
- Tambien conocido como branch coverage (cobertura de ramas)
- Cubre tanto las condiciones verdaderas como las falsas.
- Una decisión puede ser una condicional o bucle.

EN este ejemplo para alcanzar un 100% de cobertura de decision necesitaria

Ejemplo

```
READ X
READ Y
I F X>Y THEN Z = 0
ENDIF
```

minimo 2 casos de prueba, donde el primer caso necesito una variable X > a Y y se ejecute la sentencia y se le asigne el 0 a la variable Z.
Y un segundo caso donde la variable X sera < que la variable Y, no se ejecuta la condicion y la variable Z. Va a pasar por la parte falsa de la condicion. Minimo 2 pruebas para alcanzar 2 pruebas de decision.
Si fuera de condicional se necesitaria 1 caso de prueba.



PAra cubrir el 100% de cobertura de decision hay que hacer 2 rutas, es decir que devuelva los 2 verdaderos en las condicional otra ruta que siga el flujo de la izquierda. En esta técnica se toma los 2 lados siempre.

Cobertura de sentencia solamente prueba ambos lados si hay en ambas sentencias.
Cobertura de decisión siempre prueba las condicionales verdaderas y falsas independientemente si tienen o no sentencias anidadas.

Unidad 10 - Técnicas Basadas En Experiencia y Selección De Técnica

profundizaremos en las técnicas que se basan en la experiencia acumulada en la industria del desarrollo de software. Estas técnicas se apoyan en la sabiduría y el conocimiento previo de profesionales con experiencia para evaluar y probar el software de manera efectiva. Aprenderán cómo estas técnicas pueden proporcionar una perspectiva única para identificar defectos y mejorar la calidad del software.

razones fundamentales de estudiar las técnicas avanzadas:

- Optimización del ciclo de desarrollo: Las técnicas de pruebas eficientes permiten identificar y resolver problemas de manera rápida y efectiva, lo que

acelera el ciclo de desarrollo de software. Esto es esencial en un entorno competitivo donde la rapidez en la entrega de productos de calidad es fundamental.

- Minimización de riesgos empresariales: El software defectuoso puede tener un impacto significativo en la reputación de una empresa y en su capacidad para competir en el mercado. Estas técnicas de pruebas ayudan a reducir los riesgos empresariales al garantizar la fiabilidad y seguridad del software.
- Adaptación a las tendencias tecnológicas: En un mundo en constante evolución tecnológica, las técnicas de pruebas actualizadas permiten a las empresas mantenerse al día con las últimas tendencias y tecnologías. Esto es esencial para la competitividad y la relevancia en el mercado.
- Satisfacción del cliente: Al garantizar que el software funcione correctamente y cumpla con las expectativas del cliente, se mejora la satisfacción del cliente. Los clientes satisfechos son más propensos a recomendar y seguir utilizando los productos y servicios de la empresa.
- Certificaciones y credenciales profesionales: El conocimiento de técnicas de pruebas de software es altamente valorado en la industria. Obtener certificaciones y credenciales en este campo puede abrir oportunidades laborales y aumentar el potencial de ingresos de los profesionales de testing.
- Responsabilidad ética y legal: Las técnicas de pruebas de software son fundamentales para garantizar que el software no cause daños a los usuarios o incumpla con regulaciones y leyes aplicables. Cumplir con estándares éticos y legales es esencial en el desarrollo de software.

En conclusión, el estudio de las técnicas de pruebas de software es esencial no solo para el éxito profesional, sino también para la competitividad de las empresas en el mercado actual. Ayuda a minimizar riesgos, mejorar la calidad, mantenerse al día con la tecnología y, en última instancia, satisfacer las necesidades de los clientes y usuarios finales de manera ética y legal.

Técnicas basadas en la experiencia

¿Qué son las Técnicas Basadas en la Experiencia?

son un enfoque de prueba de software que depende en gran medida de la pericia, la intuición y el conocimiento previo de los probadores. En lugar de basarse en una rigurosa documentación de casos de prueba o especificaciones detalladas, los probadores confían en su experiencia en aplicaciones y tecnologías similares para identificar posibles áreas problemáticas y defectos en el software.

¿Cómo utilizar Técnicas Basadas en la Experiencia?

Utiles en situaciones donde las especificaciones son deficientes o inapropiadas, o cuando existe una presión significativa de tiempo. También pueden complementar pruebas más formales y estructuradas, actuando como una capa adicional de detección de defectos. Algunas de las formas comunes en las que se utilizan las Técnicas Basadas en la Experiencia incluyen:

- Predicción de Errores: Los probadores utilizan su experiencia para predecir posibles defectos en el software y diseñan pruebas específicas para descubrirlos. Esto a menudo implica hacer suposiciones sobre las áreas más propensas a errores.
- Pruebas Exploratorias: En lugar de seguir casos de prueba predefinidos, los probadores diseñan y ejecutan pruebas sobre la marcha. Esto es particularmente útil cuando el tiempo es limitado y se necesita una cobertura de prueba rápida.

Ventajas y Limitaciones de las Técnicas Basadas en la Experiencia

Ofrecen flexibilidad y la capacidad de descubrir defectos que pueden pasar desapercibidos en pruebas más estructuradas, también tienen limitaciones. Dependiendo en exceso de la experiencia puede llevar a sesgos y omisiones, y las pruebas basadas en la intuición no siempre son reproducibles ni documentales.

Las Técnicas Basadas en la Experiencia son un valioso complemento para las estrategias de prueba de software. Al aprovechar la intuición y la pericia de los probadores, pueden identificar defectos de manera efectiva y proporcionar una capa adicional de seguridad en el proceso de desarrollo de software. Sin embargo, es esencial equilibrar estas técnicas con pruebas más estructuradas y documentadas para garantizar una cobertura completa y confiable del software.

Basados en la experiencia **VIDEO 01**

Complementan a las técnicas de caja negra y blanca.

Predicción de errores

Casos de prueba basados en:

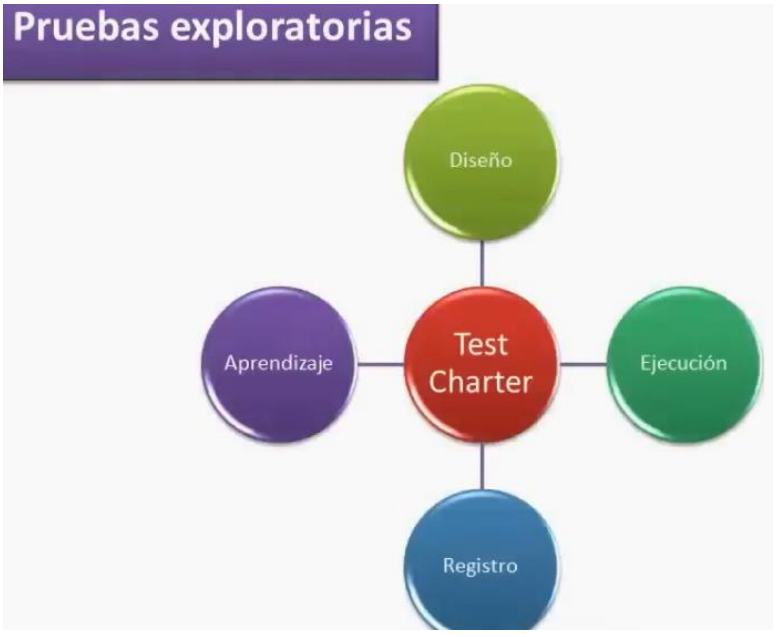
- Experiencia en otros proyectos
- Fallos detectados
- Predicciones del probador por conocimiento
- Lista de posibles defectos ➔ fault attack

attack-based testing: *An experience-based testing technique that uses software attacks to induce failures, particularly security related failures. See also attack.*
Glossary - Standard Glossary of Terms used in Software Testing – ISTQB - Version 2.4

Pruebas exploratorias

- Aplican
  - No existe documentación o la misma es muy pobre
  - El tiempo para pruebas es corto
- Partes pequeñas del software
- Un modelo mental del programa se construye
  - Cómo funciona el programa?
  - Cómo se comporta o debería comportar?

Pruebas exploratorias

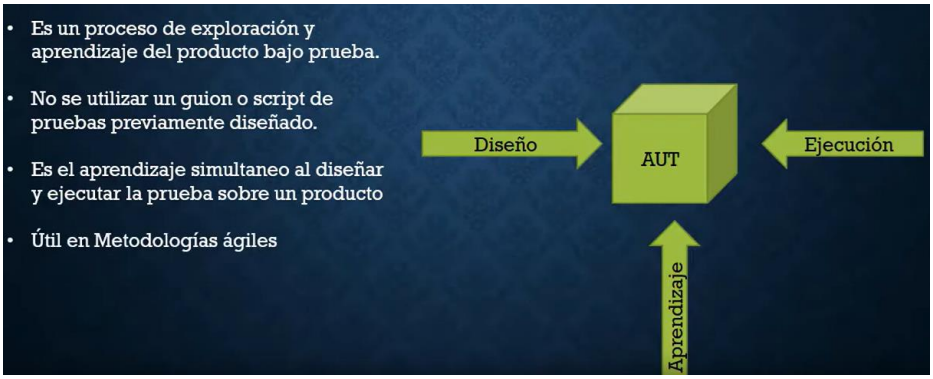


Las pruebas exploratorias no tienen un proceso definido, su característica fundamental es que las actividades de diseño de casos de prueba, ejecución de los mismos, registro de resultados y el aprendizaje de como funciona el software se realizan de manera simultánea.
Esta rutina no se realiza de manera aleatoria, responde a objetivos y a una estrategia básica de pruebas que se define de manera anticipada, se conoce como Test Charter. Esta exploracion genera nueva informacion que permite la generacion de nuevos

escenarios.

Testing exploratorio

Video 02



Es la combinación del diseño, ejecución y aprendizaje al mismo tiempo sobre el aplicativo que esta bajo prueba.