

Testing en el contexto:

Asegurar la calidad Vs controlar la calidad

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que la calidad no puede inyectarse al final, sino que depende de las tareas realizadas durante todo el proceso.

El testing no puede asegurar ni la calidad en el SW ni el SW de calidad. El testing es una disciplina en la que contemplamos requerimientos de calidad pero no la asegura. Esto es por un lado porque la calidad no abarca solamente aspectos del producto sino también del proceso, y como estos se pueden mejorar, que a su vez evita defectos recurrentes; y por otro lado porque hay otras disciplinas o actividades que se ejecutan antes que el testing (en término del ciclo de vida) y que nos permiten asegurar la calidad y que están fuera del testing.

Detectar errores en forma temprana ahorra esfuerzos, tiempo, y recursos.

¿Por qué es necesario el testing?

Quick Quiz:

Act. marcar las siguientes afirmaciones como correctas o incorrectas:

- Porque la existencia de defectos en el SW es inevitable. CORRECTA
- Para llenar el tiempo entre fin del desarrollo y el día del release. INCORRECTA
- Para probar que no hay defectos. INCORRECTA
- Porque el testing está incluido en el plan del proyecto. INCORRECTA
- Porque debuggear mi código es rápido y simple. INCORRECTA
- Para evitar ser demandado por mi cliente. CORRECTA
- Para reducir riesgos. CORRECTA
- Para construir confianza en mi producto. CORRECTA
- Porque las fallas son muy costosas. CORRECTA
- Para verificar que el SW se ajusta a los requerimientos y validar que las funciones se implementan correctamente. CORRECTA

Rompiendo mitos

Testing no es igual a calidad de producto ni calidad de proceso.

El testing NO empieza cuando se termina de codificar, puede empezar mucho antes, incluso cuando estoy definiendo requerimientos.

El testing NO es probar que el SW funciona, ya que el testing es un proceso destructivo cuyo objetivo es encontrar defectos (no probar que el SW funciona).

El tester NO es enemigo del programador, lo que pasa es que si esto no está planteado desde un lugar de sinergia o trabajo colaborativo, por ahí se puede entender que las actividades de ambos roles están enfrentadas.

Definición de prueba de software

El testing es un proceso destructivo, su objetivo es encontrar defectos ("romper el sw") esto tiene que ver con que se asume que la presencia de defectos en sw es inevitable.

Es destructivo porque la actitud del testing es una actitud negativa (no voy a decir "voy a probar que el sw funciona", lo que voy a decir es "tengo que encontrar los defectos que ya se que están en el código").

Un testing exitoso es aquel que encuentra defectos.

Mundialmente cuando hablamos de sw confiable, el 30% y 50% de un sw confiable se nos va en testing. El testing se lleva esa parte del esfuerzo de costo de sw de construir un sw confiable.

Principios del Testing

- El testing muestra presencia de defecto
- El testing exhaustivo es imposible (no es viable en termino de costo, tiempo y esfuerzo) (No se puede hacer! entonces hay que abarcar la mayor cant de casos posibles con el menor esfuerzo) (nunca voy a abarcar el 100%)
- Testing temprano (Empiezo con testing desde el momento en el que empiezo a escribir el caso de prueba, planificarlo y definir el nivel de cobertura)
- Agrupamiento de defectos (Agrupar defectos en función del universo que voy probando y los casos de prueba que voy definiendo)
- Paradoja del pesticida (Ej: cuando usas un pesticida, probablemente no tengas problema con la plaga que querés matar, pero sí con otras que no sean esa plaga) (quiero armar casos de prueba que se ejecuten automáticamente para eliminar defectos, y dejo de lado cosas que no estoy viendo por hacer ese testing automatizado)
- El testing es dependiente del contexto
- Falacia de la ausencia de errores
- Un programador debería evitar probar su propio código (Hay una excepcion que es el primer nivel de testing, el testing unitario)
- Una unidad de programación no debería probar sus propios desarrollos
- Examinar el software para robar que no hace lo que se supone que debería hacer es la mitad de la batalla, la otra mitad es ver que hace lo que no se supone que debería hacer
- NUNCA planificar el esfuerzo de testing sobre la suposición de que no se van a encontrar defectos

(Se dice defectos no errores. Encontrar defectos, corregirlos y volver a testear es todo parte del testing, porque a veces arreglo algo y se rompe otra cosa.)

¿A dónde apuntamos para encontrar los defectos?

Buscarlos pensando en que el SW haga lo que se supone lo que debería hacer (lado positivo) y también que el SW NO haga lo que no debería hacer (lado negativo)

¿Cómo se cuando dejo de testear?

Para saber que hemos llegado a hacer la cantidad de ciclos de prueba suficientes, es cuando yo tengo la confianza de que mi SW funciona correctamente. Sin embargo se que puede haber más defectos pero tengo la confianza de que mi SW funciona correctamente.

Esto también depende del riesgo de mi sistema, ej: no es lo mismo un SW que permite insertar drogas a pacientes internados que un sistema de pedidos de productos.

Entonces, una variable a tener en cuenta para saber cuando dejar de testear es el nivel de exposición al riesgo. Cuanto mayor es el riesgo que manejo más voy a tener que testear, más esfuerzo y dinero tengo que poner.

Otra variable son los costos asociados al proyecto. Va en conjunto con el riesgo. En el caso de los riesgos vamos a determinar en función del riesgo, las funcionalidades que vamos a testear primero, a qué le vamos a dedicar esfuerzo y que es lo que por ahora podemos no testear.

¿Cómo medimos cuando dejamos de testear?

El criterio de aceptación es lo que comúnmente se usa para resolver el problema de determinar cuando determinada fase de testing ha sido completada.

¿Cómo lo definió el criterio de aceptación?

Debemos definir el criterio de aceptación en base a una variable que pueda medirse. (necesitamos algo concreto para saber si se alcanzó o no el criterio de aceptación)

Por ejemplo: voy a testear hasta que un cierto porcentaje de test no fallen, voy a testear hasta llegar a cierto costo (70 hs testing), testear hasta que no haya defectos de SW de cierta severidad (ej critica o alta).

Estas son diferentes formas de enunciar el criterio de aceptación.

Los criterios de aceptación se deben acordar con el cliente. (Nos ponemos de acuerdo con el cliente en cómo cuantificar el criterio de aceptación). Lo importante es que el cliente los conozca.

Pero por más de que lleguemos al criterio de aceptación, hay que saber que siempre que entreguemos el SW este va a tener defectos, porque el testing no es exhaustivo, y eso hay que hacérselo saber al cliente (que no tenemos manera de probar el SW completo).

Conclusiones:

- El testing exhaustivo es imposible
- Las variables a tener en cuenta son: evaluación del nivel de riesgo y los costos asociados al proyecto
- Depende del riesgo: usamos los riesgos para determinar: que testear primero, a que dedicarle más esfuerzo de testing, que cosas podemos no testear (por ahora)

Afirmaciones FALSAS:

- El testing es suficiente cuando se termino todo lo planificado
- Nunca es suficiente
- Cuando se ha probado que el sistema funciona correctamente

Quick Quiz:

Cuando el testing es suficiente?

Act. maque con verdadero y falso

- Cuando se terminó todo lo planificado FALSO
- Nunca es suficiente FALSO
- Cuando se ha probado que el sistema funciona correctamente FALSO
- Cuando se tiene la confianza de que el sistema funciona correctamente VERDADERO
- Depende del riesgo de tu sistema VERDADERO

Psicología del testing:

La búsqueda de fallas puede ser visto como una crítica al producto y/o su autor, pero esto NO es cierto, el tester no “critica” al programador, sino que es una tarea o act. orientada a la calidad del producto que estamos construyendo.

En frameworks ágiles se trata de romper con esta rivalidad de estos roles, ya que no es así, trabajan en conjunto, hay que verlo como un trabajo colaborativo.

La construcción del SW requiere otra mentalidad a la hora de testear el Sw.

ERROR VS DEFECTO

En testing se habla de defectos, esos defectos encontrados en esa etapa de testing fueron generados en una etapa anterior (cuando se escribió el código se generó el error) y luego pasaron a la etapa de testing y ese error se transformó en defecto.

El error se detecta en la misma etapa en la que se genera, en cambio el defecto se encuentra en una etapa posterior.

Clasificación de los defectos

Se los puede clasificar por severidad o prioridad.

Severidad, tiene que ver con que pasa si el defecto permanece en el SW.

La clasificación según severidad es:

- 1) **Bloqueante** (Si el defecto existe no puedo seguir ejecutando el caso de prueba)
- 2) **Crítico** (a nivel de la funcionalidad que esto ejecutando es necesario corregir el defecto porque sino la funcionalidad va a tener un funcionamiento no adecuado)
- 3) **Mayor**
- 4) **Menor**
- 5) **Cosmetico** (más visual)

La prioridad tiene que ver con la prioridad de la funcionalidad y del caso de prueba en el contexto de esa funcionalidad DENTRO del negocio, de acuerdo a eso yo determino la prioridad de tratamiento. (Pienso en qué significa ese defecto para el negocio). No hay una relación directa entre severidad y prioridad (Si es severidad bloqueante no necesariamente son de urgencia en prioridad).

La clasificación según prioridad es:

- 1) **Urgencia**
- 2) **Alta**
- 3) **Media**
- 4) **Baja**

¿Cómo se hacen las pruebas?

Niveles de pruebas:

Se refiere a cómo voy escalando en términos de los que voy probando desde la menor granularidad a la mayor granularidad.

- 1) **Testing unitario:** terminé de construir un componente y lo pruebo de forma individual. Lo testeó de manera independiente, sin vincularlo a otros componentes. Se produce con acceso al código bajo pruebas y con el apoyo del entorno de desarrollo, tales como un framework de pruebas unitarias o herramientas de depuración. Se pueden usar herramientas como TDD para hacer SW automatizado. Se ejecuta el testing unitario, se encuentran errores y se corrigen y se sigue adelante.
Los errores se suelen reparar tan pronto como se encuentran, sin constancia oficial de los incidentes.
Eso lo hace el mismo programador comúnmente. Es el primer testing candidato a ser automatizado.

- 2) **Testing de integración:** Testear orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto. (Probar los componentes interrelacionándose con otros componentes, esta integración se va haciendo de a poco, no pruebo todo junto integrado de una)

Cualquier estrategia de prueba de versión o de integración debe ser incremental, para lo que existen dos esquemas principales:

- integración de arriba hacia abajo (top-down)
- Integración de abajo hacia arriba (bottom-up)

Lo ideal es una combinación de ambos esquemas.

Hay que elegir primero para probar los módulos críticos deben ser probados lo más temprano posible.

Los puntos clave del test de integración son simples:

- Conectar de a poco las partes más complejas
- Minimizar las necesidades de programas auxiliares

3) **Testing de sistema:**

Se testea el sistema completo, la aplicación funcionando como un todo, de una manera completa (prueba de la construcción final).

Trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc)

El entorno donde se realizan estas pruebas debe ser lo más parecido posible al entorno de producción, para encontrarnos en un escenario similar a cuando despleguemos el SW en producción. Y también para minimizar el riesgo de incidentes debido al ambiente específicamente y que no se encontraron en las pruebas.

Acá es importante plantear el foco no solo en las pruebas funcionales sino también en los no funcionales (seguridad, carga, performance). Deben investigar tanto requerimientos funcionales y no funcionales del sistema. (Testing funcional y no funcional)

4) **Testing de aceptación**

Lo hace el usuario para determinar si el SW se ajusta a sus necesidades. Encontrar defectos no es el foco principal en las pruebas de aceptación. (Cambia el foco de encontrar defectos a verificar si el sistema cumple con las necesidades)

Se busca verificar que el sistema se comporte de acuerdo a lo que el usuario espera. La meta en las pruebas de aceptación es el de establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema.

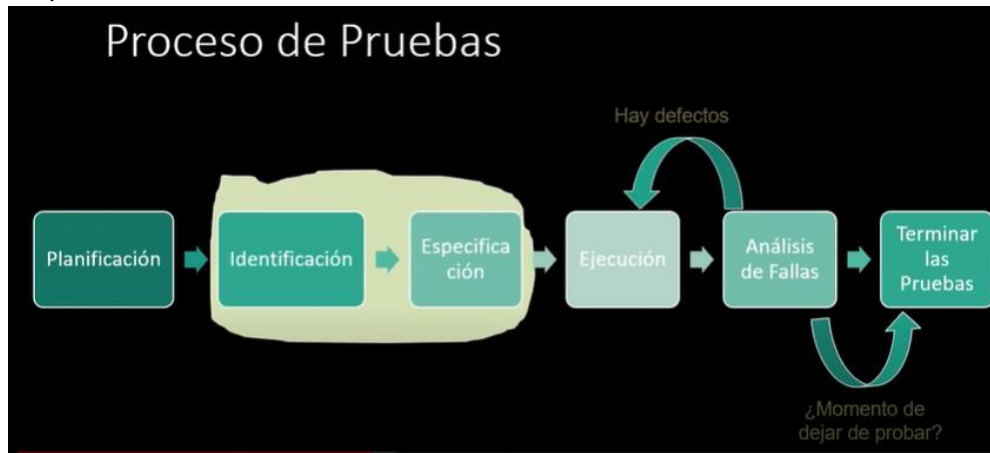
El ambiente de trabajo realizado por el usuario debe ser lo más parecido al entorno de producción.

Este testing comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa) como la prueba en ambientes de trabajo reales (pruebas beta).

¿En qué momento de scrum se haría el testing de aceptación? En la review. (demo)

PROCESO DE PRUEBA

En métodos tradicionales o procesos definidos, la prueba es también un proceso. Dentro de las metodologías tradicionales, el proceso de prueba tiene distintas etapas.



En scrum es diferente (no ocurre este proceso).

Etapas del proceso de pruebas en metodologías tradicionales

1. Planificación: armar plan de pruebas, definimos los criterios de aceptación. Es un plan subsidiario al plan de proyecto. (El plan de prueba es una partecita del plan de proyecto)

En esta etapa se

- Construye el plan de pruebas
- Acordamos metas y objetivos con nuestro cliente. criterios de aceptación.
- Definimos cuál va a ser la estrategia de testing.
- Como vamos a hacer las pruebas. Cuales manuales y cuales automatizadas.
- Cuales van a ser nuestras coberturas.

La planificación es algo VIVO, se va retroalimentando permanentemente, y en cualquier momento se puede modificar.

2. Identificación y especificación: de los casos de pruebas, qué es lo que vamos a probar. (La identificación y la especificación van de la mano)

- Tiene que ver con determinar qué vamos a probar.
- Definir los casos de prueba
- Identificar los datos para esos casos de prueba y priorizarlos
- Definir y diseñar el entorno de prueba sobre el cual vamos a ejecutar los casos.

A esta etapa se la puede ejecutar aunque no tengamos escrita una sola línea de código. Acá necesitamos tener definidos los req. funcionales y no funcionales para poder definir los casos de prueba. El código no es necesario tenerlo. Es más, es mejor que se haga esta actividad de manera paralela con el desarrollo.

3. Ejecución: ejecutamos los casos de prueba y vemos cuáles son los resultados. Si hay defectos o si no llegamos a los criterios de aceptación, volvemos a ejecutar y

volvemos a analizar las fallas, repetimos esto... HASTA CUANDO? hasta cumplir con los criterios de aceptación.

Ejecutar los casos de pruebas definidos. ¿QUÉ IMPLICA? ejecutarlos y automatizar los casos de prueba que se puedan si es que esperamos automatizar.. se empezaría a automatizar los casos de prueba unitarios.

Y además durante esta etapa también vamos a establecer la prioridad de los casos de prueba y definir cuales constituyen un ciclo de prueba.

Una vez terminada la ejecución lo más importante es (teniendo en cuenta que en la etapa de identificación y especificación definimos para cada caso de prueba los RESULTADOS ESPERADOS entonces...) lo más importante una vez terminada la ejecución es --> **COMPROBAR QUE LOS RESULTADOS QUE OBTENEMOS SEAN O NO LOS RESULTADOS QUE HABÍAMOS DEFINIDO COMO ESPERADO.** Si los resultados son los esperados, el Caso de Prueba pasa. Si no es esperado, FALLA y hay que corregir y volver a ejecutar. A su vez definiremos el defecto que se haya encontrado definiendo su severidad y prioridad de corrección.

4. Evaluación y reporte: tiene que ver con, una vez que terminamos el ciclo de prueba decimos, cuales son los defectos que encontramos? qué severidad tiene? llegamos a cumplir los criterios de aceptación o no llegamos a cumplir? seguimos probando o no seguimos probando??

Quick Quiz:

¿Cuántas líneas de código necesito para empezar a hacer testing?

CERO. Porque todas las actividades de planificación, identificación, de preparar el ambiente no necesito escribir ninguna línea de código.

¿En Scrum ? ¿Cómo hacemos este proceso?

La actividad de testing queda embebida dentro de las actividades que se ejecutan en el contexto del sprint. El testing se hace.

Tiene en términos de CP y de ejecución tiene las mismas características que cuando trabajamos en metodología tradicional, solamente que no vamos a tener un proceso definido con estas características vistas recién. Si no que queda embebida dentro del sprint y tratando de **ENCONTRAR LOS DEFECTOS LO ANTES POSIBLE Y CORREGIRLOS DE LA MANERA MÁS RÁPIDA POSIBLE.**

¿Que sería un Caso de prueba (CP)?

Tiene que ver con escribir una secuencia de pasos que tienen condiciones y variables con las que yo voy a ejecutar el sw y me van a permitir saber si el sw está funcionando correctamente o no.

La buena definición del CP nos ayuda a reproducir defectos!

EJEMPLO: Supongamos que estamos probando un sw para un GPS, y el caso de prueba es probar es buscar una calle.

Primero voy a definir una serie de condiciones o variables que yo defino a la hora de pensar en hacer el caso de prueba: voy a buscar una calle que ya esté cargada en el gps.

Entonces si vamos a buscar la calle san lorenzo, yo me voy a asegurar que cuando ejecute el CP esa calle esté cargada en el gps con una altura determinada

Ingreso en el GPS la calle SL, ingreso la altura 720 y pongo BUSCAR y cual es el resultado esperado? que el sistema me ubique cual es la calle y altura que ingrese.

Qué es lo más importante de la definición del Caso de prueba?

Primero, que yo tengo esas condiciones y variables determinadas claramente especificadas y tengo bien definidas cual es la secuencia de pasos que tengo que ejecutar con esas variables para lograr el resultado esperado.

Después si al ejecutarlo da o no el resultado esperado es otro asunto.

Todo esto es fundamental porque si el estado esperado no se logra, quien tiene que reproducir un defecto para corregirlo no tiene más que seguir los pasos enunciados en el Caso de Prueba.

Si esto no esta bien definido, y me pongo a probar lo que me parezca, cuando el defecto aparezca puede que no sea facil de reproducir el defecto. Puedo tener suerte, o si no es facil de reproducir, porque yo no me guie con un CP que estaba definido.

ENTONCES: el cp es fundamental para que el tester sepa que es lo que esta testeando, y sepa que es lo que ejecuta, sobre todo para validar si el resultado que obtengo es el esperado y si no lo es, es fundamental para saber cómo hice para llegar a ese defecto y poder corregirlo.

CONDICIONES DE PRUEBA

En el caso de prueba del ejemplo, una condición seria "que la calle san lorenzo esté cargada en el gps"

Estas son fundamentales, porque no es lo mismo que la calle esté o no cargada. Porque me cambiaría el resultado esperado!

CASO DE PRUEBA

Un caso de prueba es la unidad de la actividad de la prueba.

Consta de 3 partes:

- 1) **Objetivo:** la característica del sistema a comprobar. Tiene que ver con que es lo que yo voy a probar concretamente

- 2) **Datos de entrada y de ambiente:** datos a introducir al sistema que se encuentra en condiciones preestablecidas. (Datos o condiciones previas o precondiciones que voy a tener que contemplar)
- 3) **Comportamiento esperado:** la salida o la acción esperada en el sistema de acuerdo a los requerimientos del mismo. (Una vez que ejecuto el caso, tiene siempre un comportamiento esperado)

Con el ejemplo que dimos tenemos como mínimo 2 ejemplos: probar con una calle que exista y después una que no exista "no se puede localizar la ubicación ingresada"

LOS BUGS SE ESCONDEN EN LAS ESQUINAS Y SE CONGREGAN EN LOS LIMITES:

Uno de los aspectos a tener en cuenta (que vamos a ver en el práctico) dado que el testing exhaustivo es imposible la idea es probar lo más que podíamos del sistema con el menor esfuerzo posible. **Pensar como hago para ejecutar la menor cantidad de casos de prueba pero que me aseguren la mayor cobertura que me permita cubrir varios aspectos de la funcionalidad.**

Una de las premisas con el planteo de casos de prueba es que los bugs y los errores se concentran en los lugares límites. Esta premisa es a la hora de definir casos de prueba. Hablamos de lugares límites cuando trabajamos con rangos.

EJEMPLO DEL GPS: si estamos hablando de que tienen las calles alturas. y la calle va del 0 al 790 entonces se va a probar con el límite, que sea 791, el 0 y el 1. Para contemplar la mayor cantidad de universos posibles. Por qué? porque si pruebo el 790 y anda, es muy probable que el 500 también me funcione. Entonces probemos donde es mas probable que haya errores.

Hay una técnica, prueba de caja negra por valores límites, lo vamos a ver en profundidad en el práctico.

Derivación de los Casos de prueba

Desde donde derivamos los CP? Como enuncio los CP? Como me los imagino?



Si tenemos requerimientos planteados o US podemos derivar los CP desde ahí. Si no tenemos la US pero tenemos los Casos de uso podemos usar eso.

Si tenemos US los casos de prueba NO SON LAS PRUEBAS DE USUARIOS ESCRITAS EN LA USER. Las pruebas en las US tienen que ver con los criterios de aceptación y pruebas de aceptación que se hacen en la review que tiene que ver con a donde testeó!! Pero no son los casos de prueba que estamos hablando ahora, **estos casos de prueba no están en la user story, estos casos de prueba son del equipo para adentro. (Las pruebas de aceptación y los criterios de aceptación de la US nos van a ayudar a escribir el Caso de Prueba)** Obvio las pruebas de aceptación nos van a ayudar a definir los nuestros porque es lo que se espera para ver si se acepta o no! PERO SE HACEN EN ESE ÚLTIMO NIVEL DE ACEPTACIÓN.

Estos casos de prueba que estamos viendo ahora, son de los niveles de pruebas unitarias, de integración y de sistema.

Mientras mejor estén especificados los requerimientos, más fácil deshacer los casos de prueba.

Hay algunas formas diferentes de escribir casos de prueba, por medio de técnicas por ejemplo el diseño conducido por casos de prueba, son técnicas en las que uno en vez de escribir los req. y después hacer los casos de prueba, uno escribe los req con forma de caso de prueba. Es otra manera de derivar los casos de prueba.

Conclusiones sobre la generación de casos de prueba

Vamos a tener distintas técnicas! (caja negra, caja blanca, las vamos a usar combinadas) ninguna técnica es completa! Ninguna técnica me permite abarcar todos los casos de prueba. Lo que nosotros buscamos es usar distintas técnicas complementarias entre si para tratar de definir o de abarcar la prueba de la mayor cantidad de sw posible con la menor cantidad de CP posible.

Cómo lo vamos a lograr? Usando estas técnicas que nos ayudan a enunciar los CP de tal manera que estos nos abarque la mayor cantidad de escenarios posibles.

Si no tenemos requerimientos definidos, es mucho mas difícil. Si estuvieran bien especificados, el trabajo se nos simplifica de manera exponencial.

Todas las técnicas tienen ventajas y desventajas que apuntan a cuestiones diferentes. En el práctico vemos caja blanca y negra. Apuntan a cosas distintas.

CICLO DE TEST

Un ciclo de pruebas abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una misma versión del sistema a probar.

Supongamos que tenemos un conjunto de casos de pruebas con distintas técnicas ya definidas, con 10 CP. Ejecutamos los 10. Algunas veces obtenemos resultado esperado y otros con defectos de distinta severidad.

Si terminamos la ejecución y encontramos 20 defectos bloqueantes, **y ahí que hacemos?** tenemos que corregirlos, y hay que volver a probar para ver si se corrigieron o no. Esa vuelta a ejecutar los casos de prueba es un nuevo ciclo de test. Cada vez que yo ejecuto todos los CP definidos es a una misma versión del sistema, un ciclo de test. Cada ciclo de prueba es la ejecución de la totalidad de los CP a una misma versión.

Cuantos ciclos se van a ejecutar? Depende de los resultados de la ejecución de los CP. Si dijimos en los criterios de aceptación que vamos a ejecutar hasta que no haya más defectos bloqueantes, entonces se van a ejecutar tantos ciclos de test como sean necesarios hasta que no haya más bloqueantes.

TEST DE REGRESIÓN

Al concluir un ciclo de prueba, y reemplazarse la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión a fin de prevenir la introducción de nuevos defectos al intentar solucionar los defectos detectados.

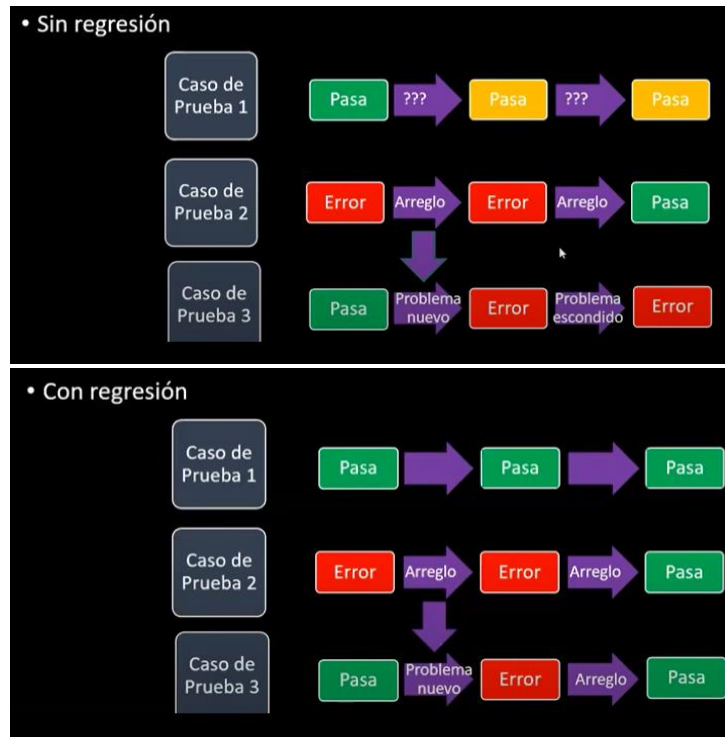
La regresión está directamente relacionada con el ciclo de prueba.

Si la regresión forma parte del testing.

La regresión nos dice que cuando terminamos un ciclo de prueba y reemplazamos la versión habiendo corregido todos los defectos, hay que volver a verificar no solamente los CP donde encontramos los defectos, sino que hay que volver a ejecutar TODOS los CP de manera completa, porque probablemente cuando intentamos corregir un defecto normalmente se produce un nuevo defecto y si nos

limitamos solamente a probar los CP en los que habíamos detectado los defectos, se nos pueden pasar por alto algunos defectos que saltaron con la corrección.

Probablemente arreglando el caso de prueba 1 se rompe el 2.
En la regresión hay que probar TODO de nuevo.



SMOKE TEST

Los test de humo tienen que ver con hacer una primera corrida del sistema en términos generales, en términos básicos, no exhaustivo, (una pasada rapidita de todo el sistema) para ver que no haya ninguna falla catastrófica.

Para ver que mas o menos anda. Para no abocarme a la ejecución detallada de los CP cuando hay alguna cosa más grosera que se me esta pasando.

DISTINTOS TIPO DE PRUEBA (FUNCIONAL Y NO FUNCIONAL)

- **Testing funcional:** se basan en funciones y características (descrita en los documentos o entendidas por los testers) y su interoperabilidad con sistemas

específicos. Está basado en los requerimientos y basado en los procesos de negocio.

- **No funcional:** Es la prueba de como funciona el sistema.No hay que olvidarlas! Los requerimientos no funcionales osn tan importantes como los no funcionales. Ej performance del Testing, pruebas de carga, prueba de stress, prueba de usabilidad, pruebas de mantenimiento, pruebas de fiabilidad, y pruebas de portabilidad.

El no funcional también es importante. Tiene vital importancia para las pruebas pre productivas! Porque tiene que ser prácticamente un ambiente igual al entorno para ver si soporta todo esto el sistema porque sino no tendrían sentido.

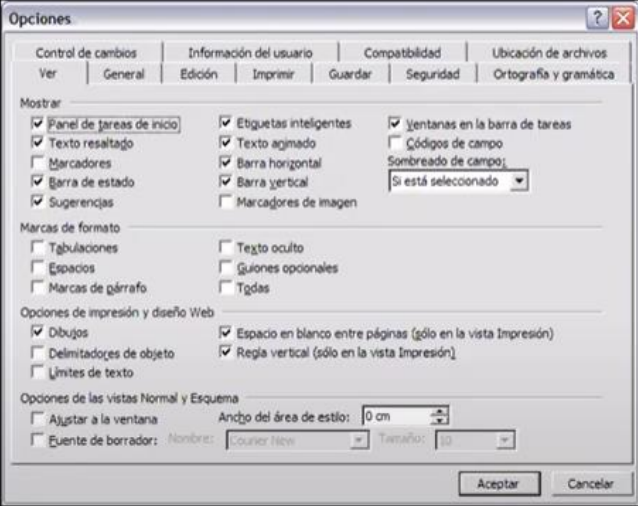
EJEMPLOS DE PRUEBAS NO FUNCIONALES

PRUEBAS DE INTERFACES DE USUARIOS

Cada vez son más complejas. Son pruebas más específicas hoy en día.

Usuario en control
+
Muchas combinaciones
=
Más pruebas

- Funciones de negocios
- Interfaces de usuarios**
- Performance
- Carga
- Estrés
- Volumen
- Configuración
- Instalación



Las GUIs, son mucho más complejas que las interfaces basadas en caracteres

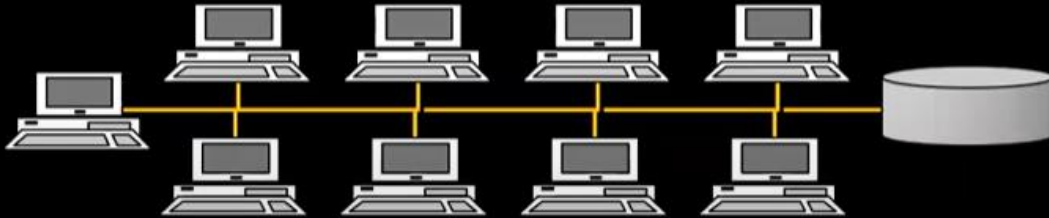
PRUEBAS DE PERFORMANCE

Tienen que ver con el tiempo de respuesta y la concurrencia.

Prueba de Performance

- Prueba de performance
 - Tiempo de respuesta
 - Concurrencia

- Funciones de negocios
- Interfaces de usuarios
- **Performance**
- Carga
- Estrés
- Volumen
- Configuración
- Instalación



PRUEBA DE CONFIGURACIÓN

Prueba de mi entorno con una determinada configuración de cada uno de sus componentes.

□ Prueba de Configuración

- Compatible con X video drivers
- Conexiones de red
- Software de terceras partes

Prueba de Configuración

Aplicación Cliente



Servidor de Base de Datos



Cliente OS

TP monitor

- Funciones de negocios
- Interfaces de usuarios
- Performance
- Carga
- Estrés
- Volumen
- **Configuración**
- Instalación

Runtime

**Mainframe
conectividad**

Server OS

Middleware

Network

E-mail

TESTING Y CICLO DE VIDA

Existen distintos tipos de CICLO DE VIDA.

Un ciclo de vida puede ser iterativo e incremental u otro cascada. Dependiendo del ciclo de vida, el testing se inserta de una manera distinta. EJ: en el ciclo de vida espiral el testing se plantea de una manera diferente que en el cascada.

Lo que sí es importante es tratar de usar aquellos ciclos de vida en donde el testing se comience a ejecutar lo más temprano posible. POR QUE? Porque nos da visibilidad de que tenemos que probar, hasta nos ayuda a ver si los req funcionales y no funcionales están correctamente especificados, y además cuanto más temprano lo introducimos, los costos de correcciones de defectos disminuyen.

VERIFICACION Y VALIDACION

Una de las actividades importantes en el ciclo de vida son estas. El testing me permite cubrir una parte pero la revisión técnica también.

- Verificación: estamos construyendo el sistema correctamente?
- Validación: estamos construyendo el sistema correcto? el que nos estan pidiendo?

Es importante usar aquellos ciclos de vida donde el testing se comience a ejecutar lo mas temprano posible. POR QUE? porque nos da visibilidad de que tenemos que probar, hasta nos ayuda a ver si los req func y no func estan correctamente especificados, y ademas cuanto mas temprano lo introducimos, los costos de correcciones de defectos disminuyen.

Existen 2 estrategias para el diseño de casos de prueba: Caja blanca y negra

En las de caja blanca: nosotros podemos ver el detalle de la implementación de la funcionalidad, vamos a poder ver el código y vamos a poder diseñar nuestro CP para poder garantizar la COBERTURA (si quiero cubrir todos los if, todos los else, si quiero cubrir todas las ramas de ejecución de cierta funcionalidad, como yo veo el codigo puedo guiar la ejecución de cp por donde yo quiera. Conozco la estructura interna)

En la de caja negra: yo no conozco la estructura interna de la implementación, solo voy a analizar en términos de entradas y salidas. Voy a identificar cuales son las diferentes entradas que una funcionalidad puede tener y voy a elegir los valores con los cuales voy a ejecutar esa funcionalidad y finalmente voy a comparar las salidas obtenidas contra las que esperaba tener.

Dentro de las estrategias tenemos distintos métodos.

(Implementación de una estrategia en particular) ¿Para qué usamos esos métodos?
Para maximizar la cantidad de defectos encontrados.

CAJA NEGRA

Dentro de caja negra tenemos 2 tipos de métodos en particular:

- **Métodos basados en las especificaciones:**

- 1) **Partición de equivalencias o clases de equivalencias:** analiza primero cuales son las diferentes condiciones externas (tanto entradas como salidas posibles) que van a estar involucradas en el desarrollo de una funcionalidad. Me planto frente a la funcionalidad e identificar las entradas y salidas posibles, las entradas pueden ser cualquier tipo de variable que puede estar en juego tanto sea un campo de texto en el cual yo escribo o un combo de selección en el cual elijo alguna opción posible, o también pueden ser las coordenadas en las que yo me encuentro actualmente, o la medición de un sensor de temperatura, etc. Puede ser cualquier condición que va a guiar la ejecución de la funcionalidad. La salida puede ser mostrar un mensaje en pantalla, o la luz en un control remoto, o la emisión de un mensaje a través de frecuencia modulada. (nos tenemos que abstraer a cualquier producto de sw. por eso esos ejemplos, no pensemos solo en una interfaz grafica)

Una vez que identifique estas condiciones externas (entrada/salida) para cada condición externa yo voy a analizar cuales son los subconjuntos de valores posibles que pueden tomar cada una de esas condiciones externas que produce un resultado equivalente.

DOS PASOS

1. Identificar las clases de equivalencias (válidas y no válidas):

Son subconjuntos que cualquier valor que yo tome de subconjunto ante la ejecución de la funcionalidad produce un resultado equivalente.

Por ejemplo: una app que vende bebidas alcohólicas lo primero que te va a pedir es la edad, si tienes 18. En ese caso tienes como variable de entrada la edad, y como variable de salida si entras o no dependiendo la edad.

Entonces un subconjunto equivalente sería todos los números menores a 18 van a producir el mismo resultado, es decir no poder ingresar. Es lo mismo si pones que tienes 12 a si tienes 15.

Entonces a simple vista nos surgen a simple vista al menos 2 clases de equivalencia: "números enteros mayores o iguales a 18" o "números enteros menores a 18" "números enteros menores a 0" --> esta última sería una clase

de equivalencia no valida. (Aca también pueden entrar valores no numéricos, valores fraccionarios y que no ingrese nada)

- **Rango de valores continuos**
- **Valores discretos**
- **Selección simple: un combo con ciudades para seleccionar**
- **Selección múltiple: opción múltiple para elegir varias opciones**

2. Identificar casos de prueba

Una vez que identifique esas clases de equivalencia, identificamos los casos de prueba, acá es donde el método tiene su resultado. (los métodos solo sirven para poder diseñar los casos de prueba). Yo lo que voy a hacer es tomar de cada una de esas condiciones externas, una clase de equivalencia en particular y Voy a elegir un valor representativo de esa clase de que para ese caso de prueba. El CP debe especificar valores que pertenecen a alguna de las clases de equivalencia, se elige un solo valor de cada clase, porque a tomar uno solo, los demás de la misma clase producen un resultado equivalente.

¿El caso de prueba que es? es una receta, es un conjunto de pasos ordenados que yo debo seguir para ejecutar la funcionalidad, con una especificación de los valores que yo voy a ingresar.

La idea de este método es que cada caso de prueba tenga UN REPRESENTANTE de cada clase de equivalencia. Osea es un método que claramente va a depender de qué tan bien hagas las clases...

PRÁCTICO DE CAJA NEGRA - VER VIDEO TESTING CAJA NEGRA PARTE 2 4K4 MINUTO 10:10

Por ejemplo cuando tengamos que ponerle prioridad al caso de prueba-- LOS DE PRIORIDAD ALTA VAN A SER LOS DE CAMINO FELIZ. Los de prioridad baja van a estar relacionados con validaciones, valores no ingresados, valores que no se corresponden con el formato esperado. Prioridad MEDIA, todo lo que esta al medio.... PRACTICÁ. Combinaciones de valores que puedan ejecutarse bajo ciertas condiciones que quiza produzcan una falla o camino no feliz.

PRECONDICIONES - Es todo el conjunto de valores o características que tiene que tener mi contexto para que yo pueda llevar adelante este CP particular. Si estuvieramos hablando de una funcionalidad en la que se requiere que el usuario esté logueado o tener ciertos permisos. EJ: El usuario "Juan" está logueado con permisos de administrador.

PASOS - Conjunto de operaciones ordenadas que debe ser totalmente claro sin ambigüedades que el tester debe ejecutar para conseguir el resultado esperado - 1."El ROL ingresa a la opc "ingresar" RESULTADO ESPERADO: "BIENVENIDO AL SITIO WEB"

2. El cliente ingresa "18" en el campo de "edad"
3. El cliente selecciona a la opc "ingresar"

- **ANALISIS DE VALORES LIMITES** - una implementacion o particularidad de particion de equivalencias.

-> los basados en la experiencia

- ADIVINANZA DE DEFECTOS
- TESTING EXPLORATORIO

Esta es la variante al metodo de particion de equivalencias.

En vez de tomar cualquier valor de la clase de equivalencia, vamos a tomar los bordes de una clase. Aca vamos a poder llegar a tomar un dupliciado.

CAJA BLANCA

En este método si disponemos de los detalles de implementación, es decir, que podemos ver el código y en base a eso poder diseñar nuestros casos de prueba.

Se basa en el análisis de la estructura interna del sw o componente de sw. Se puede garantizar el testing coverage.

Nos permiten diseñar casos de prueba maximizando la cantidad de defectos encontrados con la menor cantidad de pruebas posibles disponiendo de los detalles de la implementación. Existen diferentes coberturas que permite garantizar. Esas coberturas no son las únicas existentes y son en las que nos basaremos para la parte práctica.

Definicion de cobertura: forma de recorrer los distintos caminos que nuestro código nos provea para desarrollar una funcionalidad.

- 1) **Cobertura de enunciados o caminos básicos:** busca poder garantizar que a todos los caminos independientes que tiene nuestra funcionalidad los vamos a recorrer al menos una vez. Podemos decir que nuestro codigo ha sido ejecutado pasando por todos los caminos independientes. Nos va a dar una idea de cuantos Casos de Prueba debemos hacer para poder recorrer todo los caminos indep.

Es una métrica de SW que provee una medición cuantitativa de la complejidad lógica de un programa. Usada en el contexto de testing, define el número de caminos independientes en el conjunto básico y entrega un límite inferior para el número de casos necesarios para ejecutar todas las instrucciones al menos una vez.

Se requiere poder representar la funcionalidad a través de un diagrama de grafos.

Pasos del diseño de pruebas mediante el camino básico

- Obtener el grafo de flujo
- Obtener la complejidad ciclomática del grafo de flujo
- Definir el conjunto básico de caminos independientes
- Determinar los casos de prueba que permitan la ejecución de cada uno de los caminos anteriores
- Ejecutar cada caso de prueba y comprobar que los resultados son los esperados

Como se calcula la complejidad ciclomática:

M=Complejidad ciclomática

E=Número de aristas del grafo

F= Números de nodos del grafo

P=Numero de componentes conexos, nodos de salida

$M = E - N + 2 * P$ --> Nos da la medida de complejidad ciclomática, es decir, la medida de caminos independientes.

$M = \text{Número de regiones} + 1$

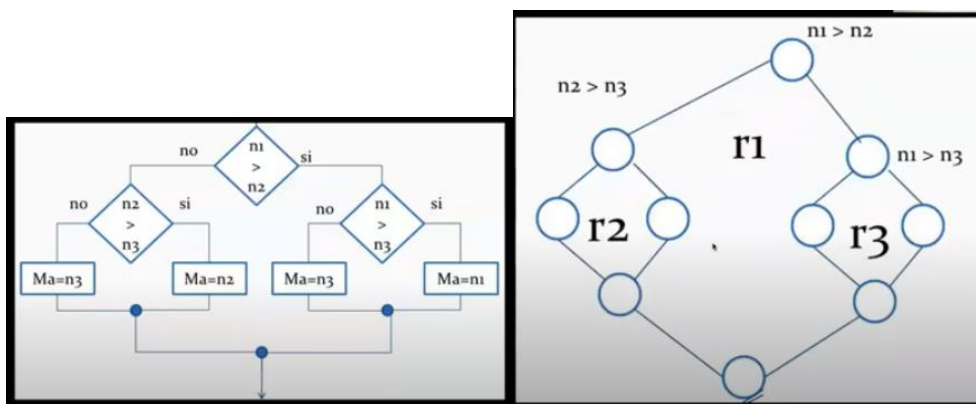
Dado un diagrama de flujo se pueden generar casos de pruebas. Por ejemplo:

-Bucle while do while

-if else

-secuencial

EJEMPLO: <https://www.youtube.com/watch?v=6lrH0k-2KQo> MINUTO 8:19



Tenemos 3 Decisiones que son nuestros IF. (los rombos)

Los caminos independientes serían: desde la raíz tendríamos:

derecha - derecha

derecha - izquierda

izquierda - izquierda

izquierda - derecha

Hay 4 caminos independientes. Y si vamos al diagrama de grafos, hay 3 estructuras cerradas entonces y según la fórmula $M = \text{Numero de regiones} + 1$ podemos comprobar que en este caso sería $3 + 1 = 4$ caminos indep.

Para poder garantizar que nosotros cubrimos todos los caminos independientes, ¿cuál es la cantidad mínima de casos de prueba que necesitamos hacer? Con solo 4 garantizamos la cobertura o de caminos básicos o de caminos independientes.

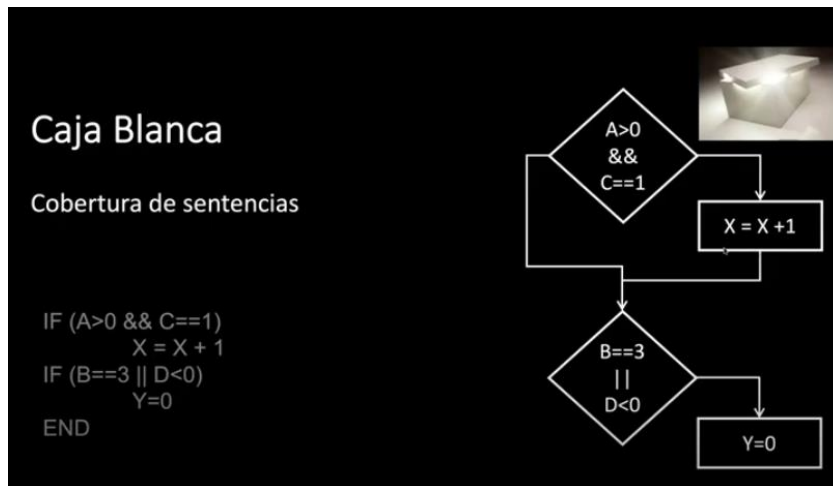
¿Podemos escribir más de 4 ?

Por más que estuviera cubierta la cobertura en sí, con los 4 CP, no quita que yo pudiera escribir 100 o 1000 CP. Solo que no es la cantidad mínima. Y lo que busca esto es escribir la menor cantidad de CP. Durante todo el práctico vamos a intentar esto, buscar descubrir cuál es la cantidad mínima de casos de prueba para garantizar la cobertura determinada.

Según la complejidad ciclomática (o cantidad de caminos independientes) podemos establecer más o menor la evaluación del riesgo de testear eso

Complejidad Ciclométrica	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

- 2) Cobertura de sentencias:
if --> decisiones



¿Cuántas sentencias tenemos en esta funcionalidad del ejemplo?

Hay dos sentencias. Por ejemplo: $x = x + 1$

Una sentencia es cualquier instrucción como asignación de variables, invocación de metodos, mostrar un mensaje, lanzar una excepcion. Cualquier instruccion que nosotros demos. ¡Mientras no sean estructura de control! Las estructuras de control (los if) son decisiones.

¿Cuál es el objetivo de la cobertura de sentencias?

Darle cobertura a todas las sentencias.

Buscar cual es la cantidad mínima de CP que me permiten pasar o ejecutar o evaluar o recorrer todas las sentencias.

En el ejemplo, con un solo caso de prueba, cubrimos todas las sentencias ya que no son excluyentes. Con elegir los correctos valores para las 4 variables distintas, con un solo caso de prueba puedo cubrir todas las sentencias.

3) Cobertura de decisión

Una decisión es una estructura de control completa. Cada una de las estructuras de control que nosotros tenemos va a ser cada uno de esos rombos en los cuales podamos tomar un camino o el otro según si el resultado de esa estructura puede dar true or false.

Por ejemplo: en un if hay que probar cada una de las ramas.

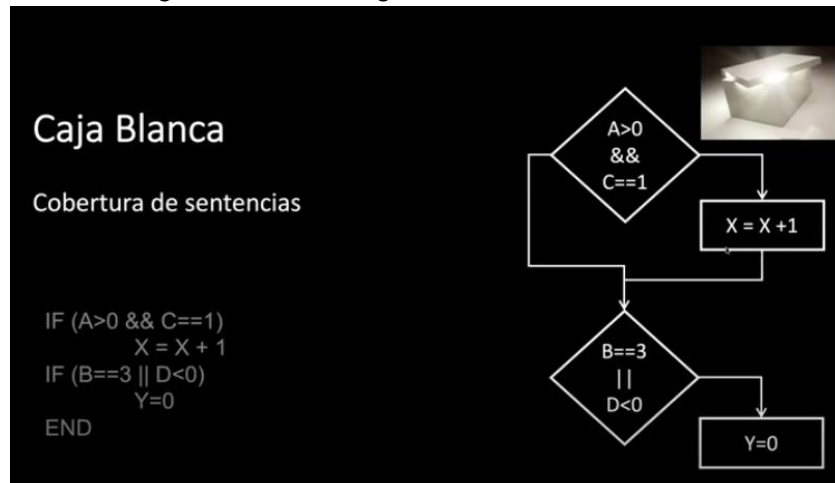
Probar todas las decisiones. buscar la mínima cantidad de CP que pruebe todas las decisiones cubriendo las dos ramas. Lo que queremos ver es si cada una de nuestras decisiones funcionan correctamente y para ver si funcionan bien tengo que ver que si se vaya por la rama del true cuando necesito que sea true y que si se vaya para la rama del false, cuando sea false.

Voy a buscar la min cantidad de Casos de Prueba para poder probar todas las decisiones que tengo en mi trozo de código, ya sea forzarlo en las 2 ramas True O False. Aca no nos interesa probar las sentencias sino nuestras decisiones.

¿Cuál creemos que es la cantidad de casos de prueba para este ejemplo?

Cada rombo es una decisión, dentro de los rombos hay combinaciones de condiciones que se encuentran unidos por operadores booleanos. Por ejemplo en el primer rombo tengo $A > 0 \ \&\& \ C == 1$, entonces, $A > 0$ es una condición, $\&\&$ el operador booleano y $C == 1$ es otra condición.

Cuando estamos en la cobertura de decisión no nos importa la combinación de condiciones de adentro, lo único que le importa es que de alguna manera salga un true o de alguna manera salga un false!



Hay dos decisiones, 4 condiciones, dos operadores lógicos .

En el ejemplo, la cantidad de casos de prueba que se van a necesitar son 2 para recorrer todos los caminos posibles.

Valores que le daremos a las variables de los casos de prueba:

TC1	TC2
A=5	A=5
C=1	C=5
B=3	B=5
D=-3	D=5

4) Cobertura de condición

Las condiciones son cada una de las evaluaciones lógicas unidas por operadores lógicos dentro de una decisión.

Busca encontrar la menor cantidad de casos de prueba que nos permita la menor cantidad de CP que nos permiten valuar tanto las condiciones en su valor V o F independientemente de por donde salga la decisión.

Evaluó todas las condiciones en su valor verdadero o falso, sin importar si el if sale por el true o el false.

Acá es al revés que antes, no nos interesa si sale T O F aca nos interesa hacer todas las combinaciones! de lo que esta adentro de las decisiones.

Para los objetivos del testing de caja blanca no le interesa hacer la salvedad con respecto al cortocircuito de los operadores logicos (esto quiere decir que puedo valuar en el ejemplo, por ejemplo la A Y C en falso al mismo tiempo) Solo le interesa

evaluar cada una de las condiciones en su valor V F minimizando la cant.de CP. En este caso al ser todos independientes se puede hacer como min 2 casos de prueba en donde hago todos los valores en false en un CP y todos en true en otro CP.

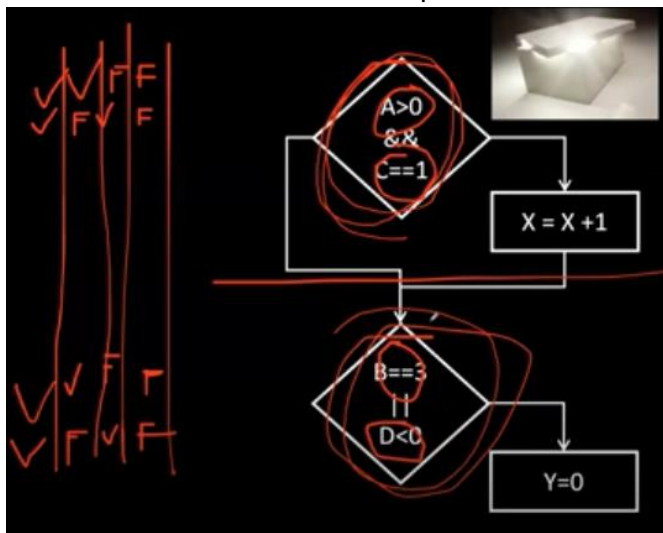
5) Cobertura de decisión/condición

La unica variante con las anteriores es no solamente busca valuar las decisiones en su valor V o F sino tambien valuar todas las condiciones en su valor V y F.

Para el ejemplo: sería como minimo 2 CP. Si hago que las 2 condiciones sean verdad entonces como es un & salgo por V. Si hago que las 2 condiciones posteriores sean verdaderas tmb al ser un OR sale V. Ahi pude cubrir todas las condiciones verdaderas y a todas las decisiones V. Lo mismo con la falsa.

6) Cobertura múltiple

Busca valuar el combinatorio de todas las condiciones en todos sus valores de verdad posible. Plantear el combinatorio de los valores de verdad para toda la combinación de condiciones disponibles.



El combinatorio seria:

VV
VF
FV
FF

VV
VF
FV
FF

Considerando que estas variables son independientes se puede hacer como minimo 4 Casos de prueba.

Si estuvieran anidadas uso: El mismo CP donde me da VV lo uso para el resto de las 4 combinaciones! osea que no necesariamente es que necesitamos 8 CP.

Para que el proceso sea destructivo tiene que haber defectos.
Un sw es exitoso cuando encontramos defectos en el sistema.

Cuanto esfuerzo se va en el testing? se va el
El costo que lleva lograr un sw coonfiable del total es el 30 a 50%
Esa es la cantidad de esfuerzo qe hay que dedicarle al testing.

Error, un hallazgo
El error es una falla que se detecta en la misma etapa que se produce, cuando vemos un mal caso de uso, en la etapa de requerimiento.

Defecto ocurre en una etapa posterior de cuando se produce. Luego del desarrollo.

Testing unitario -> se hace foco sobre un componente, una funcionalidad en particular. Las hace el desarrollador mientras construye ese componente. en el workflow de implementacion

Pruebas de integración -> Cómo funcionan todos estos componentes juntos. Implica integrar un conjunto de componentes pero no el sistema como un todo. workflow de prueba

Pruebas de sistemas -> Esto es probar como un todo. workflow de prueba

Pruebas de aceptación (Pruebas de usuario) -> El que hace el cliente. En el workflow de despliegue.

Severidad --> una perspectiva desde el lado de quien construye el sw
Prioridad --> una perspectiva del negocio. Si importa o no.

Ambientes para el desarrollo de Sw

Desarrollo

Prueba

Pre Produccion

Produccion

Los casos de prueba son para probar un escenario. Son unidades atómicas.
Es un escenario concreto y tambien condiciones y variables completas tiene que ver con los ambientes en los que hacemos testing. descubrir defecto y reproducir el defecto para saber como corregirlo. Elemento fundamental para hacer testing.

Verificación y validación -> relacionados al testing. La diferencia es validación que sea lo que pide el cliente (requerimientos) y la verificación es que funcione correctamente el producto y que no tiene defectos.

Es posible hacer un testing exhaustivo? No, porque el tiempo no alcanzaría (no se puede probar todos los escenarios posibles con todas las variables posibles) No terminaríamos nunca. Entonces con los casos de prueba tenemos que tratar de cubrir la mayor cantidad de escenarios posibles o de aspectos posibles. Cuando escribimos los casos de prueba tenemos que empezar a mirar por las esquinas y los límites porque ahí es donde se concentran los bugs.

Los casos de prueba deberían derivarse desde diferentes lados, ej: desde requerimientos, desde documentos del cliente, desde especificaciones de programación, desde el código... etc

Estrategias:

- Caja negra: se basa en las especificaciones y en la experiencia, solo tiene en cuenta las entradas y salidas (no veo lo que hay adentro, defino las entradas y las salidas que se deben esperar)

- Caja blanca: analizan la estructura interna del SW o de un componente, para ello necesito el código porque son estrategias que se basan en la estructura interna del código

Por ejemplo el testing exploratorio, se explora para buscar bugs (caja negra) esta basada en la experiencia

Objetivo del testing hay que tratar de probar lo que mas se pueda (la mayor cantidad posible de funcionalidades) con el menor esfuerzo.

Ciclo de prueba o de test: caso de prueba que voy a ejecutar para una misma versión del sistema. Cuantos ciclos de test hago para una versión del producto? Hasta que se cumpla el criterio de aceptación (dependiendo de lo que acorde con el cliente, puedo acordar que mientras no haya defectos bloqueantes)

Una regresión: volver a probar para ver si no rompieron algo cuando desarrollaron otras cosas. implica volver a testear lo que corregí. (Principio: cuando se arregla un caso de prueba que fallaba se pueden producir fallas en otros casos de prueba que antes no fallaban)

Proceso de pruebas: Planificación -> Identificación -> especificación -> ejecución -> análisis de fallas (si hay defectos vuelves a ejecución) -> fin de las pruebas

1) Planificación: defino un plan de pruebas con los lineamientos de que voy a probar, y por lo general buscas un acuerdo con el cliente. Defino niveles de pruebas y tipos de prueba.

- 2) Identificación y especificación: definir un escenario con variables concretas!
(DATOS CONCRETOS A LAS VARIABLES)
- 4) Ejecución: ejecutamos las pruebas
- 5) Análisis de fallas: analizamos y volvemos a ejecutar

Principios de testing:

Cuanto antes arranque el testing mejor, desde el momento que puedo empezar a identificar casos de prueba y trabajar..... etc

Tipos de prueba

- 1) Smoke test: primera corrida de los test de sistema que provee cierto aseguramiento de que el SW que está siendo probado no provoca una falla catastrófica de todo el sistema.
- 2) Sanity test
- 3) Testing funcional: basado en las funciones o feature del sistema, relacionado a los requerimientos
- 4) Testing no funcional: a veces se pasa por alto y produce dolores de cabeza en producción. Ej: pruebas de estrés o de carga, prueba de fiabilidad, mantenibilidad (Son tan importantes como las funcionales)

TDD: test driven development, es un mecanismo de desarrollo, si el desarrollador sabe de antemano cómo se va a probar el producto que se está construyendo probablemente lo va a construir con menos errores. Es desarrollo guiado por pruebas de SW. Tiene que ver con la integración continua y el desarrollador realiza pruebas unitarias para construir un código limpio.

LIBRO: El arte de probar el SW