Verificación y Validación

Errores, faltas y fallas

Un error humano produce una falta interna en el software que puede provocar una falla externa en el mismo. Un software falla cuando no hace lo requerido o hace algo que no debería.
Pueden existir múltiples razones:

1. Especificaciones que no estipulan lo que el cliente necesita o desea (faltan requerimientos o los que hay son incorrectos)
2. Uno o más requerimientos no se pueden implementar
3. Existen faltas en el diseño
4. Existen faltas en el código

La idea del proceso de Verificación y Validación (V&V) es detectar y corregir estas faltas antes de liberar el producto.

Objetivos:

1. Descubrir defectos para corregirlos
    a. Provocando fallas
    b. Revisando los productos
2. Evaluar la calidad de los productos
    a. El hecho de probar y/o revisar el software da una idea de la calidad del mismo

Existen dos procesos en V&V, la Identificación de defectos y la Corrección de defectos. En la identificación, se determina qué defecto o defectos causan una falla. En la corrección se realizan cambios al sistema para remover los defectos.

Debemos diferenciar el proceso de Verificación del de Validación. Distintos autores brindan sus definiciones de cada uno una misma idea de fondo.
La verificación es el proceso a través del cual se busca comprobar que se está construyendo el programa de forma correcta, es decir, siguiendo las especificaciones establecidas en los requerimientos (funcionales y no funcionales).
La validación es el proceso por el cual se busca comprobar que se está construyendo el producto correcto, es decir, lo que el cliente realmente necesita. La validación ocurre durante el final del proceso de desarrollo, ya que es ejecutada en presencia del cliente a quién se le muestra el producto en etapas finales.

Tipos de faltas existentes

Los distintos tipos de faltas que se suelen encontrar en el software se pueden clasificar en las

siguientes categorías:

1. En algoritmos
   Faltas típicas que se cometen en los algoritmos:
   a. Bifurcar a destiempo
   b. Preguntar por la condición equivocada
   c. No inicializar variables
   d. No evaluar una condición particular
   e. Comparar variables de tipos no adecuados
2. De sintaxis
   La mayoría son detectadas por los compiladores
3. De precisión y cálculo
   Incluyen fórmulas incorrectas, orden incorrecto de ejecución de cálculos, errores de precisión por truncamientos inesperados, etc.
4. De documentación
   Documentación incompleta o con errores; poco consistente con el software final.
5. De estrés o sobrecarga
   El software no se comporta bien ante situaciones de sobrecarga, por ejemplo, exceso de consumo de memoria a causa de no liberar correctamente la misma.
6. De capacidad o de borde
   El sistema colapsa al llegar a determinado volumen de datos.
7. De sincronización o coordinación
   No cumple con un requerimiento de tiempo o frecuencia.
8. De capacidad de procesamiento o desempeño
   Realizar una operación toma más tiempo de lo requerido o lleva un tiempo innecesario.
9. De recuperación
   El software no logra recuperarse correctamente después de una falla
10. De estándares y procedimientos
    El software no cumple con estándares.
11. Relativos al hardware o software del sistema
    Problemas de compatibilidad entre componentes.

Conviene categorizar y registrar los tipos de defectos de forma de obtener una guía para orientar la verificación. Si conozco los tipos de defectos más comunes en una organización, voy a poder optimizar el proceso de búsqueda y corrección de los mismos. A su vez, esto permite detectar las fases en las que se introducen más defectos, de manera de lograr una mejora del proceso en dichas fases.
Para esto conviene utilizar una “clasificación ortogonal”. Esto es, que cada defecto identificado pertenezca a una única categoría, lo cual es lógico, ya que si un mismo defecto pudiese quedar en más de una categoría, complicaría la elaboración de estrategias de corrección; terminaría complicando aún más el proceso.

Proceso de verificación y validación

Este se divide en varias partes:

1. Prueba unitaria
   Verifica las funciones de los componentes.
2. Prueba de integración
   Verifica el trabajo integrado de los componentes.
3. Prueba funcional
   Determina si el sistema ya integrado se comporta de acuerdo a los requerimientos.
4. Prueba de desempeño
   Se verifica que el sistema integrado en el ambiente cumple los requerimientos de tiempo de respuesta, capacidad de proceso y manejo de volúmenes.
5. Prueba de aceptación
   Se verifica que el sistema cumpla con los requerimientos del cliente bajo la supervición de este, y que además lo satisface (los requerimientos son correctos).
   Esto corresponde a la validación del sistema.
6. Prueba de Instalación
   Se verifica que el sistema quede instalado en el ambiente de trabajo del cliente y que además el mismo funciona correctamente.

Las pruebas unitarias normalmente son realizadas por los propios desarrolladores, en particular la misma persona que desarrolló el módulo, ya que aporta el hecho de conocer los detalles de cómo fue desarrollado.
Las de integración son realizadas por el equipo de desarrollo y es necesario que estos tengan en conocimiento las interfaces y funciones del sistema en general.

El resto de las pruebas son realizadas por un equipo especializado de verificadores que deben conocer los requerimientos y tener una visión global del funcionamiento del sistema.
Se requiere de un equipo especializado porque estos manejan mejor las técnicas de pruebas, conocen los errores más comunes realizados por el equipo de programadores (cuestiones de la organización) y carecen de problemas de psicología de pruebas. Estas últimas están basadas en que:

- El autor de un programa tiende a cometer los mismos errores al momento de probarlo.
- Debido a que es “su” programa, inconcientemente tiende a elaborar casos de prueba que no hagan que falle.
- Puede llegar a comparar mal el resultado obtenido con el esperado, debido al deseo de que pase las pruebas.

El hecho de verificar un programa, no nos permite asegurar la ausencia de fallas, sino solo la presencia de ellas. A medida que avanzamos en la corrección, vamos llevando al sistema a cumplir con los atributos de calidad deseados, que en definitiva es lo que buscamos, pero siendo conscientes de que no es posible asegurar la calidad en un 100% mediante este proceso.

Los programadores desarrollan una actitud de orgullo con respecto a su obra, lo que muchas veces deriva en un sentimiento de antagonismo para con los verificadores, desligándose de responsabilidad sobre los errores de su programa.

Se producen conflictos en el personal; por un lado, el encargado de verificación que pretende encontrar las faltas y por el otro, el desarrollador que pretende mostrar las bondades de su obra.
Las soluciones posibles:

1. Trabajo en equipo: roles distintos, pero con iguales objetivos. El desarrollador y el verificador trabajando en conjunto para obtener el producto.
2. Evaluar productos; no personas.
3. Fomentar la voluntad de mejora (tanto a nivel personal como del equipo)

Prueba unitaria

Existen tres tipos de técnicas:

1. Técnicas estáticas (o analíticas)
   Analizar un producto para deducir su correcta operación.
2. Técnicas dinámicas (o de pruebas)
   Experimentar con el comportamiento del producto para ver si actúa como es esperado.
3. Técnicas híbridas (o ejecución simbólica)
   Una combinación de las dos anteriores.

Técnicas estáticas

Dentro de estas técnicas tenemos distintos tipos:

1. Análisis de código
   Se revisa el código en busca de defectos y se puede llevar a cabo en grupos. Se buscan problemas en algoritmos y otras faltas.
   Técnicas conocidas
   - Revisión de escritorio (el programador revisa su propio código)
   - Recorridas e inspecciones
     Se critica al producto y no a la persona
     Permite unificar el estilo de programación entre el equipo de desarrollo, igualando hacia arriba la forma de programar.
     No debe usarse como forma de evaluación a los programadores.
     i. Recorridas
        Se simula la ejecución del código para descubrir faltas. Se hace con un número reducido de personas, quienes deben recibir antes el código fuente. Las reuniones no deben durar más de dos horas y estas deben enfocarse en detectar las fallas, no en corregirlas.
        Existen roles clave en el equipo:
        - Autor: quien escribió el código, lo presenta y lo explica. Se encarga de la “ejecución” del código (no olvidar que es estática).
        - Moderador: organiza la discusión.
        - Secretario: escribe el reporte de la reunión para entregarle al autor.

ii. Inspecciones
Se examina el código (pero no solo se aplican a código) buscando faltas comunes. Para esto se utiliza una lista de faltas comunes (una check-list). Estas listas dependen del lenguaje de programación y de la organización.
- Roles: Moderador o encargado de la inspección, secretario, lector, inspector y autor. Todos son inspectores. Es común que una persona tenga más de un rol. Algunos estudios indican que el rol del lector no es necesario
- Proceso de la inspección
    a. Planeación
    Seleccionar al equipo de inspección. Asegurar que el material esté completo.
    b. Visión de conjunto
    Presentación del programa y sus objetivos.
    c. Preparación individual
    Búsqueda de defectos de forma individual
    d. Reunión de inspección
    Dedicada solamente a detectar defectos
    e. Corrección
    Corrección de defectos del código por parte del autor
    f. Seguimiento
    Analizar si es necesaria otra inspección. Esto es tarea del moderador.

2. Análisis automatizado de código fuente
Se da como entrada el código fuente del programa y se despliega una lista de defectos detectados.
Se utilizan herramientas de software que recorren el código fuente detectando posibles anomalías y faltas. No requieren la ejecución del código a analizar. Más que nada es un análisis sintáctico, verificando instrucciones bien formadas e inferencias sobre el flujo de control. Funcionan como complemento al compilador.
3. Verificación formal
Se parte de una especificación formal y se busca demostrar que el programa cumple con la misma. Para esto se basa en que un programa es correcto si cumple con la especificación.
Presenta inconvenientes:
    - La demostración suele ser más larga (y compleja) que el propio programa.
    - La demostración puede ser incorrecta.
    - Demasiada matemática para el programador medio.
    - No tiene en cuenta limitaciones del hardware.
    - La especificación puede ser incorrecta. Esta es validada mediante pruebas, pero nada es 100% seguro.
    - Se debe construir una herramienta de software que lo demuestre o encuentre un contra-ejemplo. Sin embargo, esta herramienta no puede ser construida. Solo se

puede lograr una demostración asistida.

4. Ejecución simbólica
   Consiste en realizar una ejecución del código con valores simbólicos, realizando una interpretación abstracta. Se utiliza para razonar sobre las entradas que siguen un mismo camino en el código.
   Lo que hacemos es asignarle a cada variable un valor simbólico (una letra, por ejemplo) y luego ir ejecutando obteniendo fórmulas que representen los cálculos realizados con la letra que le fue asignada a la variable. Esto es, si tenemos una variable y a la cual le asignamos el valor simbólico Y, y luego ejecutamos una sentencia x = y * 2, el nuevo valor de x en la ejecución simbólica será Y * 2.
   Cuando se alcanza una bifurcación, se escribe la condición que debe cumplirse con los valores simbólicos y se estudia cada caso. Por ejemplo, si tenemos una consulta de tipo y == 12, en nuestro caso escribimos la condición Y * 2 == 12, y analizamos los casos en que esto se cumple y en los que no, continuando con el uso de valores simbólicos.

   Esta técnica tiene como ventaja que el resultado es más generico que realizar pruebas con valores directos. Sin embargo, es una técnica experimental, tiene claros problemas de escala (imaginarse aplicar esto a un código muy grande) y no aplica para interfaces gráficas o consultas SQL.

Técnicas dinámicas

Una prueba es el proceso de ejecutar un programa con el fin de encontrar fallas. Ejecutamos el producto para verificar que satisface los requerimientos e identificar diferencias entre el comportamiento real y el esperado.

Dos conceptos: Caso de prueba; Conjunto de pruebas. Un caso de prueba consiste de datos de entrada, condiciones de ejecución y un resultado esperado. Un conjunto de pruebas está formado por un conjunto de casos de prueba.

Hay dos tipos de técnicas dinámicas:

1. De caja negra
   El sistema se prueba como una “caja negra” de la que no se conoce su implementación. Se parte de la especificación para la elaboración de las pruebas y se verifica con las salidas que se obtienen. Pueden quedar porciones del código sin ejecutarse.
2. De caja blanca
   Se elaboran las pruebas en base a la implementación que es conocida. No tiene en cuenta la especificación por lo que puede ser que se pase por alto algún requerimiento del módulo.

Un ejemplo de proceso para un módulo puede ser la especificación del módulo y las pruebas de caja negra al mismo tiempo, seguido de la implementación y revisión de dicho módulo y concluyendo con las pruebas de caja blanca. Esto se debe a que durante las pruebas de caja negra, con el fin de corregir errores, se producen modificaciones al código que impactan sobre las pruebas de caja blanca.

Utilizamos estas técnicas porque es imposible realizar pruebas exhaustivas que verifiquen los resultados para todas las combinaciones posibles. Estas últimas son las únicas que asegurarían la correctitud.
Una prueba solo demuestra la presencia de fallas, nunca la ausencia de estas. Los tests deben ayudar no solo a detectar fallas sino también a localizarlas. Además, debe ser repetible, lo cual resulta difícil en programas concurrentes. El funcionamiento siempre consiste en ejecutar la prueba y comparar los resultados obtenidos con los esperados.

Necesitamos estrategias para seleccionar los subconjuntos de entradas más significativos. Debemos elaborar test sets de significancia, esto es, que tengan un alto potencial de descubrir faltas y que la ejecución de los mismos en forma correcta aumente la confiabilidad del producto. Nuestra meta debe ser correr un número suficiente de casos de prueba significativos que nos permitan reducir las probabilidades de que se produzca una falla.

Tests de caja blanca

Tipos:

1. Basadas en el flujo de control
   En términos del cubrimiento del grafo de flujo de control del programa.
2. Basadas en el flujo de datos
   En términos de cubrimiento de asociaciones definición-uso.
3. Mutation-testing
   En base a mutaciones del programa y comparaciones con el original.

Basadas en el flujo de control

1. Criterio de cubrimiento de sentencias
   Asegura que el conjunto de casos de prueba (CCP) ejecuta al menos una vez cada instrucción del código. Es un criterio sumamente débil.
2. Criterio de cubrimiento de decisión
   Asegura que todas las decisión del código evalúen al menos una vez verdadero y otra vez falso en el CCP. Es un poco más estricto que el anterior, pero no establece restricciones en cuanto a las posibles combinaciones que se pueden realizar, lo que deja afuera algunos casos en los que se podría detectar una falla.
3. Criterio de cubrimiento de condición
   En este criterio, el CCP debe hacer que para cada condición existentente en el código (es decir, las múltiples condiciones que puedan existir dentro de una decisión), estas deben evaluar al menos una vez verdadero y otra vez falso. Sigue careciendo de restricciones en cuanto a las combinaciones posibles.
4. Criterio de cubrimiento de decisión/condición
   Es una combinación de los dos anteriores. Además de que cada condición por separado debe evaluar verdadero y falso al menos una vez cada una, también cada condición por sí sola debe valer verdadero y falso al menos una vez.
5. Criterio de cubrimiento de condición múltiple
   Con este criterio se asegura que para cada decisión existente en el código se prueben

todas las combinaciones posibles de valores lógicos de las condiciones que estén adentro. Es un criterio aceptable pero no suficiente, ya que no asegura verificar todos los caminos por el hecho de que al igual que el criterio de decisión, no restringe la relación entre los valores de las decisiones.

6. Criterio de cubrimiento de arcos
   Este criterio se basa en los bloques de código de cada estructura de control, más que en las decisiones en sí. Establece que se debe pasar al menos una vez por cada arco del flujo de control. No se especifica si se debe trabajar con un grafo simple (cada nodo implica una decisión) o extendido (cada nodo es una condición), pero es fácil ver que en el primer caso se corresponde con el criterio de decisión visto antes.
7. Criterio de cubrimiento de trayectorias independientes
   Se obtiene el grafo del flujo de control y se hayan las trayectorias independientes. El criterio exige que el CCP ejecute al menos una vez cada trayectoria independiente. La cantidad de trayectorias independientes de un grafo se puede calcular con la fórmula: Arcos - Nodos + 2. Este es el número mínimo de casos de prueba necesarios para probar todas las trayectorias independientes.
   Si utilizamos una división en condiciones (y no en decisiones), es el criterio más fino hasta ahora.
8. Criterio de cubrimiento de caminos
   Se ejecuta al menos una vez cada camino posible. Es inviable para la mayoría de los grafos de flujo de control.

Basadas en el flujo de datos

Algunas definiciones previas:

- Una **definición** de una variable, es una sentencia en la cual la dicha variable se encuentra en el lado izquierdo.
- Un **uso computacional** o **c-uso** de una variable, es una sentencia en la cual dicha variable se encuentra en el lado derecho.
- Un **uso predicado** o **p-uso** de una variable, es una sentencia de un predicado booleano en la cual dicha variable está contenida.
- Un camino (i, n1, n2, …, nm, j) es un **camino libre-definición** para x desde la salida de i hasta la entrada a j si no contiene ninguna definición de x en los nodos de n1 a nm.
- Una **asociación definición-c-uso** (nd, nc-uso, x) está determinada por un camino libre-definición para x desde el nodo nd hasta el nodo nc-uso.
- Una **asociación definición-p-uso** (nd, (np-uso, t), x) y (nd, (np-uso, f), x) se define como el caso anterior, solo que se representa con dos ternas, una correspondiente a la evaluación verdadera y la otra a la falsa en el nodo np-uso.
- Un camino cualquiera es un **camino-du** para una variable x si se cumple que:
  - El camino tiene la forma (n1, n2, …, nk), donde n1 contiene una definición de x y nk contiene un c-uso, y (n1, …, nk) es limpio-definición para x, y además solo pueden ser iguales n1 y nk, es decir, no se repiten nodos en el medio.
  - Similar a lo anterior pero para un p-uso. La diferencia es que entre n1 y nk directamente no hay nodos iguales.

En base a estas definiciones podemos establecer los criterios basados en el flujo de control.

1. Todas las definiciones
   Para cada definición se ejecuta **al menos un** camino libre-definición para **algún** uso correspondiente. (c-uso o p-uso)
2. Todos los c-usos
   Para cada definición se ejecuta al menos un camino libre-definición hasta **todos** los c-usos correspondientes.
3. Todos los c-usos/algún p-uso
   Como el anterior pero en el caso de que no exista ningún c-uso se debe ir a algún p-uso.
4. Todos los p-usos
   Igual que para todos los c-usos, pero con los p-usos.
5. Todos los p-usos/algún c-uso
   Similar a todos los c-usos/algún p-uso pero al revés, viteh!
6. Todos los usos
   Se ejecuta para cada definición al menos un camino libre-definición hasta todos los c-usos y p-usos correspondientes a esa variable.
7. Todos los caminos-du
   Se ejecuta para cada definición todos los caminos-du. Esto implica que si hay varios caminos-du entre una definición y un uso, se deben ejecutar todos.

El testing basado en flujo de datos es mucho menos usado que el basado en flujo de control debido a la complejidad de la elaboración de las pruebas.

Tests de caja negra

Como ya dijimos, no tienen en cuenta el código. Se deben elaborar las pruebas con los resultados esperados para después compararlos. El problema clave es seleccionar los casos de prueba que sean realmente valiosos, es decir, con alta probabilidad de hacer fallar al software. Esto último requiere de experiencia en el área.

Veremos dos formas para guiarnos al elaborar pruebas: Particiones de equivalencia y análisis de valores límite.

Particiones de equivalencia

Un buen caso de prueba reduce significativamente la cantidad de los otros casos de prueba porque cubre un conjunto extenso de los posibles casos a probar. Una clase de equivalencia es justamente, el conjunto que contiene a los casos de prueba para los cuales suponemos que el programa se comporta igual.

El proceso consiste en identificar cuales son las clases de equivalencia de las entradas posibles y definir los casos de prueba para estas clases.
Se deben identificar tanto clases válidas como clases inválidas (aquellas de las que se espera que produzcan una falla). Si se cree que cierta entrada no es tratada de la misma forma por el

programa, dividir en clases de equivalencia más simples que identifique a esta nueva entrada con respecto al resto.

Para definir los casos de prueba:

1. Asignar un número único a cada clase de equivalencia
2. Mientras hayan clases de equivalencia sin probar, escribir un caso de prueba que abarque tantas clases de equivalencia válidas como sea posible.
3. Escribir uno y solo un caso de prueba para cada clase de equivalencia inválida, teniendo en cuenta que el caso solo debe abarcar esa única clase de equivalencia inválida y ninguna más. Esto es para evitar el cubrimiento de un error con otro error.

Valores límite

Las condiciones límite son aquellas que exploran el comportamiento del programa con valores por encima y debajo de los valores límite de las clases de equivalencia tanto de entrada como de salida (esto es aplicable a caja blanca).
La diferencia más notable es que se tienen en cuenta las clases de equivalencia de la salida, es decir, forzar los límites de los datos que el programa debe mostrar.

Comparación de las técnicas

Técnicas estáticas

1. Son efectivas en la detección temprana de defectos
2. Sirven para verificar CUALQUIER producto (requerimientos, diseño, código, casos de prueba, etc.)
3. Brindan conclusiones de validez general

Por otro lado...

1. Están sujetas a los errores de nuestro razonamiento
2. Normalmente están basadas en un modelo del producto y no en el producto
3. No se pueden utilizar para validación; solo para verificación
4. Dependen de la especificación
5. No consideran el hardware o software de base

Técnicas dinámicas

1. Consideran el ambiente en donde es ejecutado el software (es realista)
2. Sirven tanto para verificación como para validación
3. Sirven para probar otras características además de funcionalidad

Por otro lado...

1. Están atadas al contexto en donde se ejecutan
2. Su generalidad no es clara, solo se aseguran los casos ejecutados
3. Solo sirven para probar software que ya está construido
4. Normalmente solo se detecta un error por prueba

Prueba de integración

Durante este proceso probamos los módulos que componen al programa y aplicamos estrategias de integración para ir armando el sistema en su totalidad.

Prueba de módulos

Debemos manejar dos conceptos para la prueba de módulos: Driver y Stub.
Un Driver es un módulo que se crea para simular la utilización del módulo que deseamos probar. Un Stub es otro módulo que se crea para simular el comportamiento de un módulo que es utilizado por el módulo que deseamos probar. Una vez que tenemos lo necesario (Drivers y/o Stubs), se realizan pruebas utilizando los métodos vistos en pruebas unitarias.

Los stubs son costosos de realizar. Se debe tener cuidado de realizar un stub que agregue valor a la prueba, ya que uno que siempre devuelve los mismos valores no será de mucha utilidad para probar un módulo.
Los drivers son menos costosos de realizar que los stubs. Es el que suministra los casos de prueba al módulo que está siendo testeado.

Estrategias de integración

Podemos integrar de forma No Incremental (Big-Bang) o Incremental.
Dentro de las estrategias incrementales tenemos:

1. Bottom-Up
2. Top-Down
3. Sandwich
4. Por disponibilidad

El objetivo es combinar los módulos para que trabajen en forma conjunta.

En Big-Bang se prueban todos los módulos por separado y luego se integran todos a la vez. No hay integraciones parciales.
Lo bueno es que todos los módulos pueden ser probados a la vez, pero:

1. Se necesitan stubs y drivers para todos los módulos ya que estos se prueban de forma individual
2. Si hay una falla es difícil ver de dónde proviene ya que todo se integró a la vez
3. No se puede empezar la integración cuando ya hay algunos módulos disponibles
4. Los defectos en las interfaces no se distinguen claramente de otros defectos
5. No se recomienda para sistemas grandes

Bottom-Up por otro lado, comienza por los módulos que no requieren de otros módulos inferiores para funcionar. Se avanza hacia arriba en la jerarquía, por lo que solo se requieren drivers. Se aplican métodos de prueba unitaria.
Es importante planificar ya que los módulos críticos deben ser probados lo antes posible para

que estén disponibles.

El enfoque Top-Down comienza por arriba en la jerarquía. Requiere de stubs pero no de drivers.

Sandwich es una mezcla de Bottom-Up y Top-Down. Esta busca probar los módulos críticos primero ya que no siempre esto es posible con una técnica por separado.

Por disponibilidad implica simplemente que los módulos se integran a medida que están disponibles.

Comparación de estrategia no incremental con respecto a incremental

1. Requiere mayor trabajo. Se deben codificar muchos más drivers y stubs que en las incrementales.
2. Se detectan más tardíamente los errores en las interfaces ya que la integración se hace al final.
3. Se detectan más tardíamente las suposiciones con respecto a módulos ya que la integración se hace al final.
4. Es más difícil identificar el origen de una falla.
5. Los módulos se prueban menos ya que en las incrementales estoy probando indirectamente varias veces.

Aspectos de Top-Down focalizado en OO

- Resulta fácil la representación de casos reales ya que los módulos superiores tienden a ser de interfaces gráficas.
- Permite hacer demostraciones tempranas del producto
- Los stubs son muy complejos
- Las condiciones de prueba pueden ser muy difíciles o imposibles de probar

Aspectos de Bottom-Up focalizado en OO

- Las condiciones de las pruebas son fáciles de crear
- Al no usar stubs se simplifica el trabajo
- No existe el programa como entidad hasta no agregar el último módulo

Caso particular: Builds en Microsoft

Es un caso particular de técnica de integración. Separa en distintos equipos por características del sistema (y no por el sistema en su totalidad). Existe una autonomía de cada equipo sobre la especificación de las características.
Se establecen tres hitos según las características:

1. Hito 1: Características más críticas y componentes compartidos
2. Hito 2: Características deseables
3. Hito 3: Características menos críticas

En todas se diseñá, codifica, prototipa/verifica usabilidad, se realizan builds diarios y se integran con las otras características. La diferencia es que en el primer hito se agrega la eliminación de faltas severas, y en el tercer hito, se cambia la eliminación de faltas severas por la liberación a producción directamente.

Pruebas del sistema

Abarca:

1. Prueba funcional
2. Prueba de desempeño
3. Prueba de aceptación
4. Prueba de instalación

Prueba funcional

Pretende verificar que el sistema integrado realiza las funciones especificadas. Se prueban las distintas funcionalidades del sistema:

1. Cada funcionalidad por separado
2. Distintas combinaciones de funcionalidades

Es un enfoque de caja negra, basado en los requerimientos funcionales del sistema.

La prueba se realiza a partir de los casos de uso, identificando los distintos escenarios posibles y utilizándolos como base para las condiciones de prueba.

Prueba de desempeño

Se establecen **procedimientos de prueba**, **criterios de aceptación** y las **características del ambiente** de producción.

Las distintas pruebas de desempeño que se pueden realizar se listan a continuación:

1. Prueba de estrés (esfuerzo)
2. Prueba de volumen
3. Prueba de facilidad de uso
4. Prueba de seguridad
5. Prueba de rendimiento
6. Prueba de configuración
7. Prueba de compatibilidad
8. Prueba de facilidad de instalación
9. Prueba de recuperación
10. Prueba de confiabilidad
11. Prueba de documentación

Prueba de estrés (esfuerzo)

Implica someter al sistema a grandes cargas o esfuerzos. No se debe confundir con la prueba

de volumen ya que el estrés es un pico de datos durante un lapso corto de tiempo, a diferencia de volumen que es un aumento permanente de la cantidad de datos.
Se prueban volúmenes de datos, usuarios, dispositivos, etc.

Prueba de volumen

Se somete al sistema a grandes volúmenes de datos. Se busca probar que el sistema no puede trabajar con el volumen de datos especificado.

Prueba de facilidad de uso

Intenta encontrar problemas en la facilidad de uso al momento de que el usuario interactúe con el sistema. Se verifica que el lenguaje utilizado en los mensajes, botones, menues, etc. sea el adecuado. Algunas de estas pruebas pueden ser realizadas durante el prototipado. Son **sumamente** importantes ya que de no realizarse, el sistema podría resultar totalmente inutilizable.

Prueba de seguridad

Generar casos de prueba que burlen los controles de seguridad del sistema. Una forma de elaborar estas pruebas es investigar problemas típicos ya conocidos de sistemas similares.

Prueba de rendimiento

Casos de prueba que intentan mostrar que el sistema no cumple con los requerimientos de rendimiento especificados. Se observan los tiempos de respuesta y tiempos máximos de ejecución de tareas. Normalmente esto se especifica bajo ciertas condiciones de carga y configuración del sistema.

Prueba de configuración

Pruebas del sistema sobre distintas configuraciones de hardware y software. Se prueba con las configuraciones mínima y máxima posibles, como mínimo.

Prueba de compatibilidad

Se utilizan cuando el sistema tiene interfaces con otros componentes de hardware y/o software. Se busca probar que las interfaces no se comportan según lo especificado.

Prueba de facilidad de instalación

Se prueban distintas formas de instalar el sistema.

Prueba de recuperación

Se intenta probar que el sistema no queda estable luego de una falla. Se simulan o crean fallas y luego se verifica el estado del sistema mediante la recuperación.

Prueba de confiabilidad

Si bien toda prueba busca aumentar la confiabilidad, esta en particular intenta atacar los requerimientos específicos de confiabilidad. Se deben diseñar pruebas especiales para esto. Muchas veces se debe estimar la validez con modelos matemáticos ya que no es posible

probarlo en la realidad. Ejemplos de esto son requerimientos de confiabilidad que se miden en meses o años.

Prueba de documentación

Se debe revisar la documentación verificando la exactitud y claridad de la misma. Todos los ejemplos que se muestren deben funcionar correctamente en el sistema por lo que deben ser utilizados como casos de prueba.

Los métodos y los tipos de prueba dependen mucho del sistema del que se trate. Existen herramientas que automatizan algunas de estas pruebas, como por ejemplo las de esfuerzo.

Prueba de aceptación

Estas pruebas son realizadas por el cliente para ver si el sistema funciona de acuerdo a sus requerimientos y necesidades.

Prueba de instalación

Su propósito es encontrar errores de instalación. Es de mayor importancia si la prueba de aceptación no se realizó en el ambiente de instalación.

Otras pruebas

Cuando se desarrolla software para múltiples clientes se realizan otros tipos de pruebas conocidas como Pruebas Alfa y Beta. Las Alfa son realizadas por el equipo de desarrollo sobre el producto final. Las Beta las realizan clientes elegidos sobre el producto final. Un ejemplo de esto es Windows, que es un software que técnicamente siempre está en Beta Testing.

Otro caso de pruebas especiales se da cuando debemos desarrollar un sistema que sustituye a otro. Para esto se realizan pruebas en paralelo, dejando al sistema “viejo” funcionando mientras se integra el nuevo. Se realizan pruebas para ver que el sistema nuevo funciona correctamente.

Pruebas de regresión

Se realizan cuando se producen cambios para verificar que lo que ya estaba probado sigue funcionando.

Pruebas piloto

Se pone en funcionamiento el producto en áreas localizadas para reducir el impacto de fallas.

Herramientas para verificación

Cada vez se utilizan más. En algunos procesos es obligatorio utilizarlas.
Existen herramientas para distintos usos:

1. Verificación automatizada (centrada en el código)
    a. Estáticas (no se ejecuta el programa)

        i. Verifican la sintaxis
        ii. Generan grafos de flujo de control y pueden detectar problemas
    b. Dinámicas (se ejecuta el programa)
        i. Rastreo de la ejecución
        ii. Examina el contenido de memoria o instancias de datos
2. Ejecución de pruebas
    a. Automatización
        i. Elaboración del plan de pruebas
        ii. Ejecución del plan
    b. Captura y reproducción
        i. Digitación, clics y movimientos del mouse
        ii. Se guardan los datos y resultados esperados
        iii. Luego de detectar un defecto y corregirlo son utilizadas para repetir las pruebas
    c. Generación de drivers y stubs
    d. Evaluadores de cobertura
    e. Generadores de casos de prueba
        i. Generan a partir del código fuente
        ii. Generan casos de prueba para asegurar determinada cobertura
        iii. También pueden generar a partir de especificaciones
    f. Ambientes integrados (IDE)
        i. Ofrecen un conjunto de facilidades
        ii. Integran los tipos de herramientas mencionados y otros tipos

Clasificación de las herramientas de verificación

- Test Design Tools
  Herramientas que ayudan a decidir cuáles tests ejecutar.
- GUI Test Drivers
  Herramientas para automatizar tests de interfaces gráficas.
- Load and Performance Tools
  Herramientas para someter el sistema a altas cargas.
- Unit Test Tools
  Herramientas para testing unitario.
- Test Implementation Tools
  Generan stubs y por otro lado assertions para hacer las faltas más obvias.
- Test Evaluation Tools
  Ayudan a evaluar qué tan buenos son los test sets. Ejemplo de esto es el code coverage.
- Static Analysis Tools
  Herramientas que analizan programas sin correrlos.
- Defect Tracking Tools
  Herramientas que ayudan a manejar una base de datos con reportes de defectos.
- Test Managers
  Ayudan a gerenciar grandes cantidades de test sets.

Planificación de verificación y validación

El mayor error que se puede cometer al elaborar la planificación del proceso de prueba, es suponer al momento de armar el cronograma, que no se encontrarán fallas.
Esto trae consigo la subestimación del personal y tiempo necesario, así como también puede traer atada la entrega tardía del sistema y en el peor de los casos, directamente no entregar.

El proceso de V&V es caro, por lo que se requiere de una buena planificación para lograr el mayor aprovechamiento posible de dicho proceso y controlar sus costos.
Se ponen en la balanza los enfoques estático y dinámico para la verificación y la validación; se utilizan estándares y procedimientos para las revisiones y pruebas del software; se establecen listas de verificación para las inspecciones; se define el plan de pruebas del software.
A sistemas más críticos se les debe realizar una mayor verificación estática.

Un plan de pruebas comprende aplicar los estándares del proceso de prueba, más que describir las pruebas en sí. Esto permite que el personal de verificación obtenga una visión global de las pruebas del sistema y se ubiquen en ese contexto. También proveen información a los responsables para que se aseguren de que los requerimientos de software y hardware apropiados estén cubiertos al momento de realizar las pruebas.

Los planes de prueba no son estáticos y deben revisarse regularmente.

Terminación de las pruebas

Según un estudio de Myers, un aumento en la cantidad de fallas encontradas, implica un aumento en la cantidad de fallas que quedan por encontrar. Esto hace difícil saber cuándo detener la prueba. ¿Cómo saber cuándo terminar la prueba? Existen algunos criterios utilizados.

Criterios usados pero contraproducentes:

1. Terminar cuando el tiempo establecido para pruebas ha terminado
2. Terminar cuando se han ejecutado todos los casos de prueba y los mismos han sido infructuosos (no se encontraron errores)

El primero es inservible porque tal vez ni siquiera se haya llegado a la etapa de pruebas o haya transcurrido un tiempo insuficiente. No mide la calidad de las pruebas.
El segundo es malo porque es independiente de la calidad de las pruebas realizadas. Si estos casos son realizados por los propios desarrolladores, basándonos en el hecho de que tienden a crear pruebas que no hacen fallar su programa, estamos ante un claro ejemplo de inutilidad.

Categorías de criterios de detención (Myers):

1. Por criterios del diseño de los casos de pruebas
2. Por cantidad de defectos encontrados
3. Por cantidad de fallas por unidad de tiempo durante las pruebas

Criterio del diseño de las pruebas

Según el criterio de cubrimiento utilizado se terminan las pruebas cuando se ejecutan todos los casos de prueba sin provocar fallas. Por ejemplo, terminar cuando se cubran todas las trayectorias independientes.

Pueden ser apropiados para las pruebas unitarias, pero no son muy útiles en la prueba del sistema. Se fijan los método de diseño de las pruebas pero no se da una meta en cuanto a las fallas.

Criterio basado en la cantidad de defectos

Debemos tener una idea de la cantidad de defectos que hay. Debemos estimar.
Se presentan formas para estimar esto:

1. Siembra de defectos
2. Pruebas independientes

Siembra de defectos

Se agregan defectos y se ejecutan las pruebas elaboradas. Luego se establece una relación entre los defectos encontrados que fueron sembrados y los encontrados que ya estaban. Despejando se obtiene una estimación de cuántos defectos reales hay en el programa.

Es un enfoque simple y útil, pero asume que los defectos sembrados son representativos de los defectos reales. Esto depende del tipo y complejidad y es muy subjetivo.

Pruebas independientes

Se basa en utilizar dos grupos de prueba independientes a probar el mismo programa. Se establece la cantidad de defectos en base a una relación entre los defectos encontrados por un equipo y el otro.

Criterio basado en la cantidad de fallas por unidad de tiempo

Se grafica la cantidad de defectos detectados por unidad de tiempo. Observando la forma de la curva se decide cuando detenerse. Una baja en la curva puede deberse a una baja en la calidad de las pruebas, por lo que debemos asegurarnos de que este no sea el caso.

Lo más efectivo resulta ser una combinación de las últimas dos: “Se pararán las pruebas cuando se detecte un número predeterminado de defectos o cuando haya transcurrido el tiempo fijado para ello, eligiendo la que ocurra más tarde, siempre que un análisis del gráfico de cantidad de defectos por unidad de tiempo en función del tiempo transcurrido indique que la prueba se ha tornado improductiva”.