Fundamentos de Pruebas de software

Error (Bug)

Una equivocación cometida por el desarrollador
Esta puede ser de requisitos, diseño y/o implementación.

Defectos (Defect)

Es el resultado de una deficiencia o error durante la construcción del software

- Es cualquier termino que esta fuera de rangos normales (anomalías)
- Parte de algo más grande
- No corresponde a sus especificaciones

Fallo (Failure)

Es la incapacidad del software o una de sus partes de realizar sus funciones especificadas

- Resultado no esperado
- Hay o no hay un mensaje de error
- Interrupción

Relación entre Error, Defecto y Fallo

Error: lo introduce el programador
Defecto: el error plasmado en el software
Fallo: el sistema no se comporta como debería
Efectos Negativos: Generado por el fallo

Errores Comunes

- División entre cero
- Ciclo infinito
- Desbordamientos (overflow)
- Condición de carrera
- Variable no inicializada
- Memoria no permitida (Violación de acceso)
- Pérdida de memoria (memory leak)
- Desbordamiento del arreglo
- Desbordamiento de búfer (buffer overflow)
- Bloqueo mutuo (deadlock)
- Indizado inadecuado de tablas en bases de datos

Diferencia entre Validación y Verificación

Validación ~ *¿Construimos el producto correcto?*

Proceso de evaluación del SW para determinar si cumple lo esperado por el cliente.
Lo haces en todo el proceso de desarrollo

Ejemplo: Hago el diseño, lo valido, hago los requisitos, lo valido. Para ir sabiendo si lo que he avanzado es lo que el cliente quiere y necesita. Es como un feedback por parte del cliente.

Verificación ~ ¿Construimos correctamente el producto?

Proceso de análisis y revisión de que el SW cumple los objetivos establecidos al inicio.
Se basa en los objetivos ya establecidos con el cliente, como verificar los requisitos ya hablados con el cliente.

Pruebas en el ciclo de vida de software

Inspección

Proceso de revisión de código (u otros elementos) para encontrar defectos

- Revisión y enfocarse en código fuente
- Es menos costoso, ve los defectos es forma temprana
- Trata de no encubrir o minimizar errores
- Verificar el cumplimiento de estándares
- No reemplaza una prueba

Prueba

Proceso que busca demostrar que el sistema hace lo que debe hacer, localizando errores y/o anomalías.

- Técnicas de descubrir anomalías
- Diseñadas y planificadas
- Diferentes clases y en distintos niveles
- Ingreso de datos de prueba

¿Quienes realizan las inspección y pruebas?

En si depende, En cualquiera de los casos pueden automatizarse

Equipo de desarrollo

- Conoce el funcionamiento del sistema
- Sesgo al probar
- Económico a corto plazo

Equipo de pruebas

- Personas especializadas en calidad

- Mas costoso, pero más objetivo
- Paralelizable con desarrollo

Depuración

Es el proceso para corregir los errores y problemas encontrados por las pruebas

- Eliminar los bugs encontrados
- Usar las salidas de las pruebas (mensajes, resultados esperados, datos introducidos) para brindar soluciones
- Realizadas por el equipo desarrollo
- Se debe dar un tiempo estimado (explicito) en el cronograma

Pase a producción

Proceso de instalación, implantación o puesta a disposición de un SW en el ambiente donde realizar sus funciones

- Una vez culminadas las pruebas
- Pasar del entorno de desarrollo al entorno final
- Puede haber entornos de preproducción intermedios
- Largo o directo dependiendo de la complejidad

Aseguramiento de calidad (QA) y Control de calidad (QC)

	QA	QC
Definición	Aseguran la calidad en los procesos que desarrollan el producto.	Aseguran la calidad del producto.
Enfoque	Prevenir defectos. Es proactivo.	Identificar defectos. Es reactivo.
Objetivo	Mejorar el desarrollo y pruebas para que no surjan defectos al desarrollar.	Identificar defectos al desarrollar pero antes de su entrega.
Cómo	Establecer un sistema de gestión de calidad y revisar que sea adecuado. Auditorías pe riódicas.	Encontrar y eliminar problemas de calidad para asegurar que se cumplan los requerimientos.
Responsable	Todo el equipo encargado del de sarrollo.	Es el equipo encargado de las pruebas.
Tipo de Herramienta	Es una herramienta de gestión	Es una herramienta correctiva / de control

QA está enfocado el proceso
QC está enfocado al producto

Gestión de la Configuración

Gestión de elementos (y versiones) de un Software.

Incluye:
◎ Código fuente, Scripts de pruebas, SW de terceros
◎ Hardware, datos, documentación
◎ Parámetros de configuración, entre otros.

Ejemplo: Tengo la Versión 1, que contiene MySQL 8.2, JQuery 5.4.2 pero en mi Versión 2 uso MySQL 8.2, JQuery 5.4.3 pero falló, así q el posible defecto está en la librería.

Entonces la gestión de configuración guarda la relación de las versiones para:

- Asegurar entrega de la versión correcta de elementos a probar.
- Especificar la configuración de dichos elementos a probar.
- Identificar qué defectos se encuentran en qué versión.

Documentación de pruebas

Plan de pruebas

Especifica cómo se deben ejecutar las pruebas. Puede incluir:

- Contexto de las pruebas: Uso principal del sistema, Elementos a probar, Asunciones, restricciones
- Comunicación: Interesados, forma de comunicación de resultados (A quien se tiene que informar)
- Riesgos en el desarrollo de las pruebas
- Calendarización e identificación de responsables en cada actividad de las pruebas
- Casos de prueba especificados (es el detalle de todas las pruebas que voy hacer)

Otros documentos

Dependiendo de la organización y/o proyecto se pueden pedir cierta documentación adicional. Algunos documentos según el estándar ISO/IEC/IEEE 29119-3:

ISO/IEC/IEEE 29119 es una serie de cinco estándares internacionales para pruebas de software

◉ Plan de pruebas
◉ Política de prueba
◉ Especificación de diseño de pruebas

◉ Especificación de casos de pruebas
◉ Informe de preparación de datos de prueba / entorno de prueba
◉ Resultados actuales
◉ Informe de incidentes de prueba

Principios de Pruebas

Los 7 Principios de Pruebas

- **Principio 01: Las pruebas sirven para demostrar defectos**
Cuando probamos, sólo estamos mostrando los defectos que existen, pero no aseguramos que ellos no existan. Pueden existir en algo que no se probó.

- **Principio 02: Las pruebas exhaustivas son imposibles**
No es posible realizar todas las pruebas con todos los valores posibles. Se requeriría una cantidad astronómica de casos de pruebas. Los casos son sólo una muestra, por lo que los esfuerzos se deben controlar y priorizar.

- **Principio 03: Las pruebas deben empezar lo más pronto posible**
Se debe empezar a probar lo más pronto que se pueda en el ciclo de vida del Software. Eso ayuda a detectar los defectos antes que se distribuyan en más elementos.
No hay necesidad de tener código, podemos iniciar desde los requerimientos.

- **Principio 04: Los defectos se aglutinan**
Los defectos no suelen distribuirse homogéneamente en el sistema. Si se encuentran varios defectos en una parte del Software, es probable que haya más en partes cercanas. Suele suceder que, cuando hay más complejidad, se generan "clústers" de defectos.

- **Principio 05: La paradoja del pesticida**
Así como los insectos y bacterias se hacen resistentes, el software se hace resistente a las pruebas que no son mantenidas. Si se prueba siempre lo mismo, eventualmente ya no se encuentran defectos. Hay que ir nutriendo la batería con nuevos casos (o actualizar los antiguos), según se van desarrollando nuevas funcionalidades.

- **Principio 06: Las pruebas dependen del contexto**
No se pueden probar las mismas cosas en diferentes sistemas. Cada sistema tendrá sus propios criterios de pruebas, de exhaustividad de acuerdo a su criticidad y uso. El contexto puede incluir los riesgos, los tipos de usuario, los tipos de dispositivos, el uso lúdico o profesional, el uso para atención a otras personas, entre otros.

- **Principio 07: La ausencia de errores es una falacia**
Encontrar errores y repararlos no indica que el sistema cumplirá todas las expectativas del cliente. Tampoco que no habrá ningún defecto u error en el software. El proceso de pruebas disminuye la posibilidad de que suceda, pero no lo evita.

Mini - Casos

Pregunta 1

¿En qué consistiría el pase a producción de una app móvil? (Asuma que es auto-contenida, es decir, no utiliza servicios externos y/o de un servidor propio)

◉ A: En realizar pruebas de aceptación.
◉ B: En ponerla a disposición en el Marketplace correspondiente.
◉ C: En depurar sus errores.
◉ D: En integrar sus módulos.

¿Qué etapa de desarrollo debo haber terminado antes de que se realice?
B, Ya es ponerla en mano del consumidor final. Y después de la depuración es el pase a producción así que en depuración es la etapa de desarrollo que debo haber terminado

Pregunta 2

Un equipo de desarrollo acaba de terminar de implementar un sistema para una cadena de pastelería, que permite manejar el inventario de productos para elaborar la cobertura de chocolate. Al terminar la demo de forma satisfactoria, los clientes, sin embargo, les comentan que el sistema que buscaban era el que manejara la cobertura de los seguros que tienen contratados en sus tiendas. ¿Qué proceso falló?

◉ A: La Verificación
◉ **B: La Validación**
◉ C: El Pase a Producción
◉ D: La Depuración

Pregunta 3

Un startup dedicado al desarrollo de una aplicación de envíos tiene una política para incluir una nueva funcionalidad, que exige:

● Una serie de documentos establecidos.

● La funcionalidad debe encontrarse versionada.

● Se debe haber superado los casos de prueba.

Esta política es un ejemplo de:
◉ A: Gestión de la Configuración
◉ B. Control de Calidad
◉ C. Aseguramiento de la Calidad
◉ D. Desarrollo Iterativo

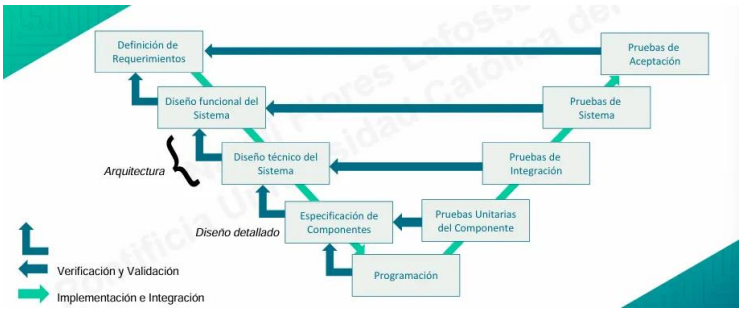
RPTA: Porque estamos hablando de políticas para poder realizar el pase.

Tipo de Pruebas

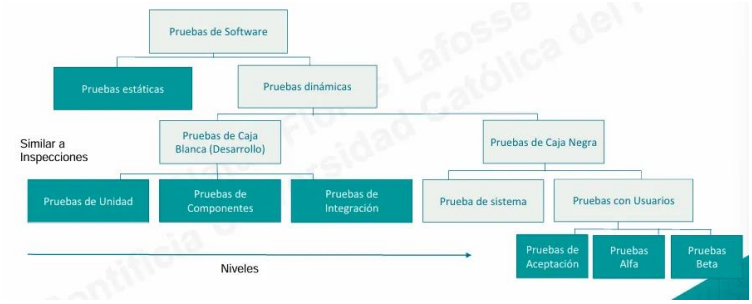
Clasificación ~ *generalidades*

Podemos clasificarlas por nivel, por el método o técnica de aplicación, por los objetivos que buscan, por el tipo de requerimientos que revisan, etc.

Modelo V



Árbol de tipo de prueba



Prueba estática ~ *Inspección de código*

Se refieren al análisis del software (o elementos relacionados) cuando no se encuentra ejecutando, Se busca encontrar errores antes de mayor desarrollo.
En lugar de ejecutar el programa, las pruebas estáticas implican revisar el código fuente, la especificación de requisitos, la documentación de casos de uso y otros artefactos del desarrollo. Se buscan errores como complejidad del código, incumplimiento de estándares de codificación, inconsistencias en la lógica o en los requisitos.

Prueba dinámica ~ *Ejecución de la aplicación*

Son pruebas en que se realizan con el código ya en ejecución, esta puede tipificarse por niveles.

Prueba de caja blanca ~ *Como está estructurado internamente el sistema*

Se analiza como el procedimiento va pasando parte por parte, se puede hacer a distintos niveles (granularidad), pruebas de cobertura.

Pruebas de Caja Negra ~ *Ingreso de datos en la interfaz*

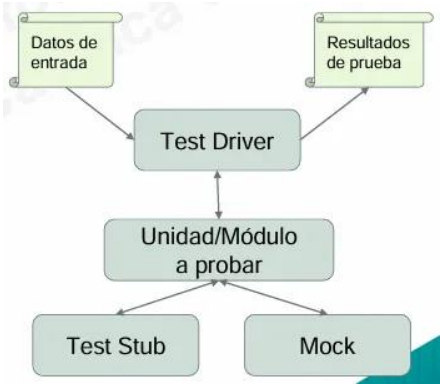
Sólo enfocarse en entradas VS salidas esperadas, se ve el componente o modulo sin el detalle interno.

Se diseñan casos de prueba que simulan el uso real del sistema por parte del usuario, comprobando si las salidas son las esperadas para cada entrada. Un ejemplo de pruebas de caja negra son las pruebas de aceptación, donde el cliente evalúa si el sistema cumple con sus requisitos

Pruebas de unidad

Se prueba el correcto funcionamiento de cada objeto (o módulo) por completo y de forma individual antes de integrarlo con otros componentes. Implica verificar:

◉Todos los métodos
◉Todas las salidas
◉Todas las entradas
◉Todos los estados
◉Todas las herencias



Automatización: Las pruebas de unidad se suelen automatizar. Esto significa que se utiliza un framework de pruebas que permite ejecutar las pruebas de forma repetitiva y automática, lo que facilita la detección temprana de errores y acelera el proceso de desarrollo

- Test Driver (Manejador de prueba): Objeto que llama al módulo a probar.

- Es una clase que instancia el objeto a probar, le pasa las entradas y valida las salidas obtenidas con las salidas esperadas. Actúa como un "controlador" que orquesta la ejecución de las pruebas.
- Se hace uno por cada clase que tengas

- Test Stub: Es un objeto que el módulo a probar llama y que siempre devuelve un resultado estático. Se utiliza para simular el comportamiento de un componente externo o dependiente, sin necesidad de implementarlo completamente

- Si tu método tiene n salidas tienes que hace n test Stub o Mock.
- Devuelve un resultado fijo. No se verifica la interacción.

- Mock Object: Similar al Test Stub, pero más flexible, ya que permite definir comportamientos específicos y verificar las interacciones con el objeto a probar. Se utiliza cuando se necesita simular un objeto complejo con diferentes respuestas en función de las entradas.

- Permite definir comportamientos específicos y verificar la interacción.

Pruebas de Componentes

Se enfocan en el uso de las interfaces entre los componentes

Características principales:

- **Enfoque en las interfaces:** El objetivo principal es asegurar que la comunicación entre los componentes se realiza correctamente, que los datos se transfieren adecuadamente y que se manejan los errores de forma eficaz.
- **Flexibilidad en el enfoque:** Las pruebas de componentes pueden adoptar un enfoque de caja blanca o caja negra, dependiendo del objetivo específico de la prueba.
  - *Caja blanca:* Se analiza el código fuente de los componentes y sus interacciones para identificar posibles problemas.
  - *Caja negra:* Se centra en las entradas y salidas de los componentes, sin considerar su estructura interna.
- **Puntos de atención:** Algunos aspectos que se suelen probar en las pruebas de componentes son:

- Validez de la arquitectura: Se verifica que la arquitectura del sistema se implementa correctamente y que los componentes interactúan de acuerdo a lo diseñado.

- Uso correcto de las interfaces: Se comprueba que los componentes utilizan las interfaces de forma correcta, pasando los parámetros adecuados y manejando las respuestas de manera esperada.

- Gestión de errores: Se evalúa cómo los componentes manejan las situaciones de error, como la falta de respuesta de un componente o la recepción de datos inválidos.

- Problemas de temporización: Se analizan los posibles problemas de sincronización entre componentes, como las condiciones de carrera, que pueden generar comportamientos inesperados.

Lineamientos sugeridos

◉Listar llamados de interfaces
◉Forzar llamados a nulos (si no está el componente)
◉Probar que pasa cuando falla el otro componente
◉Pruebas de esfuerzo al enviar mensajes
◉Pruebas de memoria compartida oner a prueba la arquitectura
◉Uso incorrecto de interfaz
◉Errores temporización (carrera)

Prueba de Integración

Las pruebas de integración son parte de las pruebas de sistema. Para integrar, se pueden usar diferentes estrategias:

Ejemplos en donde usar cada Estrategia

- Top Down

Caso: Se está desarrollando un sistema de gestión de inventario con una interfaz de usuario (modulo principal, es el punto de entrada del usuario), un módulo de control de stock y un módulo de conexión con la base de datos. La interfaz de usuario está lista para ser probada, pero los módulos de control de stock y de base de datos aún están en desarrollo.

Razonamiento: En este caso, se puede utilizar la estrategia Top Down para probar la interfaz de usuario y su interacción con los stubs que simulan los módulos de control de stock y de base de datos. Esto permite una validación temprana de la interfaz de usuario y la detección de errores de diseño en la arquitectura

- Bottom up

Caso: Se está desarrollando una biblioteca de funciones matemáticas que se utilizará en diferentes aplicaciones. Cada función matemática se implementa como un módulo independiente.

Razonamiento: La estrategia Bottom Up es adecuada en este caso, ya que permite probar individualmente cada función matemática (módulo atómico) y asegurarse de que funcionan correctamente antes de integrarlas en las aplicaciones.

- Sándwich

Caso: Se está desarrollando un sistema de comercio electrónico con una capa de presentación (interfaz de usuario), una capa de lógica de negocio (gestión de pedidos, carrito de compras) y una capa de acceso a datos (conexión a la base de datos). La capa de presentación y la capa de acceso a datos están listas para ser probadas, pero la capa de lógica de negocio aún está en desarrollo.

Razonamiento: La estrategia Sándwich es ideal en este caso, ya que permite probar la interacción entre la capa de presentación y la capa de acceso a datos utilizando stubs para simular la capa de lógica de negocio. Al mismo tiempo, se pueden probar los módulos de la capa de lógica de negocio individualmente desde abajo hacia arriba.

- Big Bang

Caso: Se está desarrollando una pequeña aplicación web con un único módulo que gestiona todas las funcionalidades.

Razonamiento: La estrategia Big Bang puede ser suficiente en este caso, ya que el sistema es pequeño y la integración de todos los módulos a la vez no supone un gran riesgo. Sin embargo, es importante tener en cuenta que esta estrategia no es recomendable para sistemas grandes o complejos, ya que la detección y corrección de errores puede ser muy difícil si se integra todo de una vez

◉Top Down

Se va probando la funcionalidad desde el módulo principal hasta los módulos inferiores.

Desventajas
◉Requiere muchos stubs de prueba.
◉Mayor abstracción

Ventajas
- Permite una visión temprana del funcionamiento general del sistema y facilita la detección de errores de diseño en la arquitectura

◉Bottom Up
Se empieza con los módulos más atómicos. Luego se van probando los de mayor jerarquía

Desventajas
- Errores pueden posponerse hasta el final (puede ser costoso)

Ventajas
- Permite detectar errores en los módulos básicos desde el principio y facilita la reutilización de código

◉Sandwich
Combinación de Top Down & Bottom Up.

◉El sistema se ve en 3 capas: Inferior, Media, Superior.
◉Las pruebas convergen en las capas medias, donde está la lógica del negocio.

Ventajas
- Permitiendo una visión temprana del sistema y la detección temprana de errores en los módulos básicos.

Deventajas
- Requiere un mayor esfuerzo de planificación y coordinación.

◉BigBang

◉Se prueban los módulos individuales y luego se junta todo.
◉Sólo para sistemas pequeños (económico)
◉Muy riesgoso en sistemas grandes.

Pruebas de Sistema
Se enfoca en evaluar el sistema completo e integrado, verificando si cumple con los requerimientos especificados y si funciona correctamente como una unidad.

Características principales:

Alcance amplio: Abarcan todo el sistema, incluyendo sus componentes, interacciones, interfaces, bases de datos y entornos.

Simulación del entorno real: Se realizan en un entorno lo más similar posible al entorno de producción, utilizando datos y configuraciones reales.

Enfoque en la funcionalidad y el rendimiento: Se evalúa tanto la funcionalidad general del sistema como su rendimiento, seguridad, usabilidad, escalabilidad, etc.

Pruebas de Sistema Especificas
Pruebas que se pueden realizar para tipos de sistemas especializados o en escenarios específicos. Dependen mucho del tipo de sistema y proyecto.

◉ *Pruebas basadas en requerimientos*
Se verifica que el sistema cumpla con cada uno de los requerimientos especificados, asegurándose de que sean medibles y específicos

◉ *Pruebas de escenario*
Se definen escenarios de uso del sistema según los roles de los usuarios, especificando las interacciones y validando el comportamiento del sistema

◉*Pruebas de aseguramiento de calidad de datos*
Se comprueba que los datos del sistema sean correctos, consistentes y correspondan a lo esperado. Estas pruebas son especialmente importantes en sistemas nuevos, migraciones o productos de datos

◉*Pruebas de rendimiento*
Se evalúa la capacidad del sistema para manejar diferentes cargas de trabajo, incluyendo

◎*Prueba de esfuerzo (estrés)*
Se somete al sistema a una carga superior a la máxima esperada para verificar su resistencia y estabilidad. Se suele automatizar (Jmeter, AB, Siege).

◎*Prueba de seguridad*
Se utilizan protocolos específicos para probar las capas del sistema (como la red) y detectar vulnerabilidades.

◉*Prueba de humo*
Se realiza una prueba rápida para confirmar que las funcionalidades principales del sistema funcionan correctamente. Se puede usar en diferentes niveles de desarrollo si resulta útil

◉*Pruebas de regresión*
Se ejecutan pruebas existentes después de cambios o actualizaciones en el sistema para asegurar que no se ha roto ninguna funcionalidad previa. Pueden incluir otros tipos de pruebas

◉*Prueba de cordura*
Similar a la prueba de humo, pero más enfocada en verificar que las nuevas funcionalidades o correcciones no hayan introducido nuevos errores

Sólo confirmar que lo principal no se ha roto por más desarrollos. (Regresionista)

Pruebas con usuario ~ *Pruebas de aceptación*

Se busca obtener la validación del sistema por parte de los usuarios finales. Estas pruebas se enfocan en evaluar si el sistema cumple con las expectativas, necesidades y requisitos del cliente, determinando si está listo para su lanzamiento o producción.

Nota: En XP (Extreme Programming) y otras metodologías ágiles, las pruebas de aceptación son parte del desarrollo del sistema, por lo que no existe como paso separado

- Pruebas alfa:

Se realizan en un entorno controlado, generalmente dentro de las instalaciones del equipo de desarrollo. Los usuarios trabajan con una versión preliminar del sistema, aún en desarrollo, y colaboran con el equipo para identificar errores y áreas de mejora.

- Pruebas beta:

Se libera una versión del sistema a un grupo más amplio de usuarios externos, quienes lo utilizan en su propio entorno y proporcionan retroalimentación. Esto permite obtener una perspectiva más diversa sobre el sistema y detectar problemas que podrían no haber surgido en las pruebas alfa.

◎ *"Always in Beta":* Algunos sistemas se mantienen en un estado de "beta perpetua", donde se lanzan actualizaciones y mejoras de forma continua, basándose en la retroalimentación de los usuarios

◎*Release Candidate:* Se denomina así a una versión del sistema que se considera potencialmente lista para su lanzamiento, pero que se somete a una última ronda de pruebas de usuario antes de su publicación final

Mini - Casos

Caso 1
El equipo de desarrollo del Banco ha terminado de desarrollar un nuevo módulo relacionado a Fondos Mutuos. El módulo utiliza una serie de clases ya existentes (cliente, cuenta, depósitos, etc.), que fueron implementados utilizando Stubs y Mocks. Además, el gestor del módulo (capa superior) utiliza una plantilla similar a los otros módulos ya existentes. ¿Cuál consideras que es la mejor estrategia de integración?

◉ A: Top Down
◉ B: Bottom Up
◉ C: Sandwich
◉ D: Big Bang

Razones:
Como la interfaz principal está definida, y los módulos están definidas.

Caso 2 y 3
Se va a liberar un nuevo juego en línea la próxima semana. Todo el juego ya ha sido probado funcionalmente (con 10 usuarios en simultáneo) y todos los módulos están integrados. ¿Qué otras pruebas sugerirían hacer antes del lanzamiento?
Se actualizó el módulo de compras del ERP de una institución y el módulo de contabilidad dejó de funcionar. ¿Qué tipo de prueba no se realizó, dado que no se esperaba dicha situación?

Razones:
- Pruebas de seguridad
- Pruebas de integración, Pruebas de Regresión

Caso 4
Francisco es uno de los representantes de la empresa GoGo, la cual ha solicitado a un equipo de desarrollo de Software la implementación de un aplicativo móvil para su tienda en línea. El día de hoy, ha recibido un correo electrónico del equipo, donde le hicieron llegar una lista de ítems (entradas y salidas esperadas) a probar, indicando

(Seleccionar todas las respuestas posibles) ? que debe seguirla para realizar las pruebas de caja blanca. Además, se adjunta un aplicativo. Francisco considera que el correo podría tratarse de un error.

¿Cuál podría haber sido la situación correcta? ¿Qué podría hacer Francisco

◉ A: El correo iba dirigido a miembros del equipo para que desarrollen las pruebas de caja blanca. Francisco puede contestarles indicando que se equivocaron de remitente.
◉ B: El correo no tiene errores, solo que Francisco no quiere hacerse responsable.
◉ C: El correo se refería a pruebas de Caja Negra, que sí podrían ser realizadas por Francisco. Francisco puede contestarles para confirmar si esas pruebas debe realizarlas él.
◉ D: La información enviada es insuficiente para que Francisco realice las pruebas de caja blanca. Francisco debe pedir el detalle del funcionamiento interno del aplicativo, a fin de poder realizar él mismo las pruebas de Caja Blanca.

Razones:
Puede ser la a de que como habla de caja blanca se abran equivocado de remitente, y la c si es asumir q se confundieron y quisieron decir caja negra y eso si lo pueden hacer

Diseño de Pruebas

Enfoques al planificar pruebas

Se puede elegir entre distintos enfoques para definir la estrategia a utilizar al planificar y/o estimar las pruebas

Analítico

◎ Riesgos, requerimientos u otros elementos para analizar

Basado en modelos

◎ Creación/selección de modelos (matemáticos) para comportamientos críticos.

Metodológico

◎ Procedimiento definido por experiencia, in house.

Por Cumplimiento de Estándares o Procesos

◎ Estándar IEE829, o XP (ágil) o cualquiera desarrollado por terceros.

Dinámico

◎ Adaptarse a errores encontrados en la ejecución (pruebas exploratorias o basadas en ataques)

Dirigido

◎ Foco en lo que quiere el usuario

Regresionista

◎ Mantener lo probado una vez, funcionando.
◎ La automatización es fundamental.

¿Cómo elegir cuál usar (o combinar)?

Se debe considerar:
Riegos, Regulaciones, Habilidades, Producto, Objetivos y Negocio

Caso de Prueba

Un caso de prueba es el elemento fundamental del proceso de pruebas. Se deben elegir casos efectivos, es decir:

Caso Ideal → Mostrar que el sistema funciona como se esperaba (cuando se usa como se indica).

Caso con problemas Comunes → Si hay "errores", que sean visibles

Partición de datos de entrada ~ para casos de prueba

◉ Datos de entrada agrupados por características similares.
◉ Estos grupos se llaman "clases de equivalencia".
◉ Definir (para cada una) un caso de prueba.
◉ Valores límites por separado para ver comportamientos no esperados
◉ Son pruebas de caja negra

Clases de Equivalencia

Rango de valores: una clase válida y dos clases no válidas.
Elementos de una clase que no se tratan igual: clases menores.

Conjunto de valores admitidos: una clase válida por cada valor.
Condición booleana (p.e. “el primer carácter debe ser una letra”): una clase válida (“es una letra”) y una no válida (“no es una letra”).

Ejemplo 1

Para calcular el Impuesto a la Renta se debe tomar en cuenta que las tasas varían según la Remuneración Neta Anual (RNA).
Armar las clases de equivalencia que se necesitaría para probar el ingreso como dato de entrada del RNA en un sistema.

RNA en S/.	Tasa
Desde 5 UIT	14%

Condición de Entrada	Clases Válidas	ClasesNo Válidas
Campo Remuneración Neta Anual	1. Valores mayores a 0 y menor o igual a 25 750	6. Vacío
	2. Valores mayores a 25 750 y menor o igual a 103 000	7. Valor menor o igual a 0
	3. Valores mayores a 103 000 y menor o igual a 180 250	8. Valor mayor al máximo admitido por el tipo de dato.
	4. Valores mayores a 180 250 y menor o igual a 231 750	9. Datos no numéricos
	5. Valores mayores a 231 750 y menores al máximo admitido por el tipo de dato	

Ejemplo 2

Para registrarse en un sistema, el usuario debe ingresar una contraseña que cumpla las siguientes condiciones:

- ❖ Debe incluir al menos una mayúscula
- ❖ Debe incluir al menos una minúscula
- ❖ Debe incluir al menos un número
- ❖ Debe tener una longitud mínima de 6 caracteres
- ❖ Debe tener una longitud máxima de 14 caracteres

Armar las clases de equivalencia que se necesitaría para probar que la contraseña es válida. Asumir sólo caracteres alfanuméricos.

Especificación de Caso de Prueba

Una vez diseñada la prueba, se debe crear su especificación. Debe contener por lo menos:

- ◉Identificador
- ◉Objetivo
- ◉Precondición
- ◉Descripción de la prueba
- ◉Resultados esperados

Ojo: Dependiendo del proyecto o institución, pueden requerirse otros elementos.

Prueba MU-25	
Objetivo	Dar de alta a un usuario dejando vacío el nombre de usuario.
Precondición	Se ingresó al sistema como “registrador”
Descripción de la prueba	En la interfaz de entrada introducir - Username: usuario1 - Primer Apellido: apellido - Segundo Apellido: apellido - Tipo de usuario: estándar Dejar vacío el campo “Nombre”
Resultados esperados	Se muestra el mensaje “El nombre de usuario es obligatorio”

Caso de prueba y Clases de Equivalencia

- ◉Cada combinatoria de clases válidas genera un caso de prueba.
- ◉Cada clase inválida genera un caso de prueba

Condición de Entrada	Clases Válidas	Clases No Válidas
Campo código	1. Cadena numérica de 5 caracteres	2. Vacío (0 caracteres)
		3. Entre 1 y 4 caracteres
		4. Más de 5 caracteres
		5. Cadena de caracteres no numéricos
Campo Tipo Usuario	6. Estándar	
	7. Administrativo	

- ◉Casos de Prueba Válidos: (1,6) (1,7)
- ◉Casos de Prueba Inválidos: (2,6) (3,6) (4,7) (5,7)

Otras Formas de Derivar Pruebas

Pruebas Basadas en Lineamientos

Se debe tomar en cuenta la experiencia del equipo y de la industria para probar errores recurrentes.
Algunos generales:

- ❖ Entradas que fuercen todos los mensajes de error
- ❖ Entradas que desborden el buffer
- ❖ Repetir las mismas acciones una y otra vez
- ❖ Forzar salidas inválidas
- ❖ Forzar cálculos muy grandes o muy pequeños

Otras Formas

- ◉Diseño de pruebas por ejecutables
- ◉Diseño de pruebas por ramas
- ◉Diseño de pruebas por cambio de estados
- ◉Diseño de pruebas por valores aleatorios

Mini-caso 01

La consultora de Software STEIN ha conseguido el contrato con una empresa vendedora de celulares, para desarrollar su tienda web y aplicativo móvil. En la primera reunión, los representantes de la empresa han dejado en claro que lo más importante para ellos es que los clientes puedan comparar los diferentes productos, así que no debe haber fallas en dicha funcionalidad. ¿Al seguir esta directiva, la consultora STEIN está siguiendo un enfoque de pruebas?

◉ A: Metodológico
◉ **B: Dirigido**
◉ C: Dinámico
◉ D: Regresionista

Este enfoque se alinea con la definición de pruebas dirigidas descrita en la fuente: "Foco en lo que quiere el usuario".

Las demás opciones no se ajustan al escenario:

A: Metodológico: Implica seguir un procedimiento predefinido por la experiencia de la empresa, no la prioridad del cliente.

C: Dinámico: Se basa en la adaptación a errores encontrados durante la ejecución de pruebas, no en un requerimiento preestablecido.

D: Regresionista: Se centra en garantizar que las funcionalidades previamente probadas sigan funcionando correctamente después de cambios en el código, no en una nueva funcionalidad específica.

Mini-caso 02

Liste las clases de equivalencia para diseñar los casos de prueba según las siguientes consideraciones:

❖ Todos los campos son obligatorios.
❖ El nombre del curso es de máximo 20 caracteres alfanuméricos.
❖ El Código curso es de 6 caracteres alfanuméricos. Los 3 últimos deben ser números.
❖ La especialidad puede ser “Ingeniería Informática”, “Ingeniería Industrial” o “Ingeniería Electrónica”.
❖ El año de Inicio de Matrícula tiene que ser mayor a 2000 y máximo 2024.
❖ Cuando el año es menor a 2019, el registro se guardará como “Histórico”

Hacer la tabla de equivalencia

Api

Grupo de protocolos que permiten comunicar diferentes componentes de software para transferir datos.

REST (Representational State Transfer):
Los recursos se acceden por un endpoint (url) y se usan los métodos de HTTP como GET, POST, PUT y DELETE. Comúnmente usa JSON.

SOAP (Simple Object Access Protocol)
XML para transferir mensajes altamente estructurados entre cliente y servidor. Se usa mucho en sistemas legados, suele ser algo más lento.

Diferencia de enfoque al hacer pruebas de APIs

◉ No se cuenta con interfaz gráfica.
◉ El usuario no suele tener acceso directo a ellas.
◉ Son de caja negra (hay documentación).
◉ Es conveniente usar software específico/automatizado

Tres consideraciones en Pruebas de API

◉ Especificación de interfaz y documentación
◉ Seguridad
◉ Desempeño

Especificación del API

Antes de poder generar nuestros casos de prueba, debemos conocer su especificación y comportamiento.

¿Cómo se comporta la interfaz cuando...
◉ ...no hay parámetros?
◉ ...hay un parámetro correcto con valor correcto?
◉ ...hay un nombre de parámetro incorrecto?
◉ ...hay parámetros sin valor?
◉ ...hay un valor de parámetro incorrecto?
◉ ...hay una combinación correcta de parámetros?

Sobre su resultado:
◉ ¿Cuál es el formato de respuesta si no especifica un formato?
◉ ¿Cuál es el formato de respuesta tanto para condiciones de éxito o error?
◉ ¿Cuál es el código de respuesta tanto para condiciones de éxito o error?
◉ ¿Cuál es la respuesta del API para métodos, cabeceras, URLs inesperados?

Documentación del API

◉ Es la descripción del contrato del API.
◉ Debería responder (si no todas) la mayoría de las preguntas; pero es necesario asegurarse de ello.
◉ Verificar la correspondencia entre la versión de la documentación y la del API.

Pruebas de contrato

◉ Son pruebas para confirmar que la especificación del API corresponde a lo esperado por nuestro sistema.
◉ Debe Incluir:

◎ **Validación del código y formato de respuesta:**200 en REST y un JSON válido, por ejemplo
◎ **Verificación de estructura:** Contiene objetos, cabeceras y propiedades URL requeridas
◎ **Validación de datos enviados:** Validar que cumplen lo esperado
◎ **Consistencia de datos:** Verifica que la data enviada corresponde a la respuesta

Beneficios

◉ **Colaboración:** El proveedor y consumidor entiende la especificación de forma temprana.
◉ **Detección de incompatibilidad temprana:** Se identifica los problemas y desviaciones antes de que la integración se haga compleja.
◉ **Fiabilidad aumentada:** Permite que proveedor y consumidor cumplan los términos del contrato y así asegurar un comportamiento prescindible de la interfaz.

Desventajas

◉ Las Pruebas de contrato no deberían usarse para probar escenarios completos.
- Por ello mismo, un comportamiento complejo tal vez no sería probado desde el inicio y su complejidad relegada a otro momento.
◉ Las pruebas de contrato no deben usarse para probar ni la seguridad ni el desempeño del API

```
Contract.make {
        request {
                description('Get a list of all the attendees at a conference')
                method GET()
                url '/conference/1234/attendees'
                headers {
                        contentType('application/json')
                        }
                }
        response {
                status OK()
                headers {
                        contentType('application/json')
                        }
                body(
                value: [
                        $(
                                id: 123456,
                                givenName: 'James',
                                familyName: 'Gough'
                        ),
                        $(
                                id: 123457,
                                givenName: 'Matthew',
                                familyName: 'Auburn'
                        )
                        ]
                )
                }
        }
```

Pruebas unitarias en APIs

◉ Sirven para realizar las pruebas de que el endpoint de la API responde a nuestros casos de prueba específicos.
◉ Ayuda a localizar los casos de error.
◉ Podemos usar clases de equivalencia, considerando los parámetros del API.

Pruebas de Autenticación y Autorización

◉ Necesarias si el recurso es privado o limitado.
◉ Las APIs pueden usar un juego de llaves mutuas (intercambio de llaves) o tokens usando OAuth/OpenID.
◉ Las pruebas deben enfocarse en el acceso correcto a sólo la información que nos corresponde.

Otras pruebas (si el API es nuestra)

Si nosotros somos los proveedores del API deberíamos considerar:

◉ Pruebas para evitar API Fuzzing (Envío de parámetros aleatorios para conseguir información de la estructura de base de datos o sistema de archivos)

◉ Pruebas para protegerse de inyección maliciosa o malformada:

◎ **Malformada:** Estructura dentro de estructura para saturar el sistema al des-anidar.

◎ **Contenido:** Códigos ingresados como texto plano. Revisar que no se recibe: SQL, javascript, shell, XQuery, python, etc

Pruebas de Desempeño

◉ Realizar pruebas en condiciones normales, para descubrir cuellos de botella.

◉ Realizar pruebas con carga artificial, para obtener métricas de comportamiento: tiempo total, latencia, errores obtenidos, uso de CPU.

◉ Realizar pruebas de estrés, para determinar los puntos de quiebre.

◉ Realizar pruebas de remojo (Soak) manteniendo el sistema activo por largos periodos, para analizar si se generan errores de memoria o inestabilidad.

Pruebas punto a punto

◉ Son pruebas de escenarios completos, equivalente a las pruebas de sistema.

◉ Es importante ver como los diferentes servicios funcionan en combinación.

◉ Es importante analizar si alguna de las pruebas anteriores no consideró parte de alguno de los escenarios

Mini-caso 01

Maricarmen es parte del equipo que ha desarrollado un sistema que utiliza el API de Niubiz y además se conecta a la SUNAT para realizar la facturación electrónica. Antes de ponerlo en producción, el equipo decidió realizar una prueba automatizada de carga, usando su tarjeta de crédito. Sin embargo, la tarjeta se bloqueó por comportamiento sospechoso.

¿Qué sugerencias podrían darle al equipo de Maricarmen? ¿Cómo habrían planteado las pruebas?

Planteamiento de Pruebas:

1.Pruebas Unitarias:

- Verificar individualmente cada endpoint de la API de Niubiz y SUNAT.
- Utilizar clases de equivalencia para probar diferentes tipos de datos de entrada, incluyendo valores válidos, inválidos y límites.
- Asegurar que los códigos de respuesta y el formato de las respuestas sean correctos, tanto en casos de éxito como de error.

2.Pruebas de Contrato:

- Validar que la documentación de las APIs coincide con su comportamiento real.
- Confirmar que la estructura de las respuestas, incluyendo objetos, cabeceras y propiedades URL, es la esperada.
- Verificar la consistencia de los datos enviados y recibidos.

3.Pruebas de Autenticación y Autorización:

- Asegurar que el sistema solo puede acceder a la información que le corresponde.
- Probar diferentes escenarios de autenticación, incluyendo tokens OAuth/OpenID y otros métodos de seguridad.

4.Pruebas de Desempeño:

- **Pruebas de Carga:** Simular una cantidad realista de usuarios concurrentes para identificar cuellos de botella.
- **Pruebas de Estrés:** Incrementar la carga del sistema hasta encontrar su punto de quiebre y determinar su capacidad máxima.
- **Pruebas de Remojo:** Ejecutar el sistema bajo una carga constante durante un periodo prolongado para detectar fugas de memoria o problemas de estabilidad.

5.Pruebas Punto a Punto:

- Probar el sistema completo, incluyendo la interacción entre el API de Niubiz, SUNAT y la aplicación desarrollada.
- Simular escenarios reales de facturación electrónica para asegurar que el sistema funciona correctamente de principio a fin

Mini-caso 02

Carlos Alberto ha desarrollado una API para la generación de mapas en un videojuego y está documentando su contrato/especificación.
Un mapa se envía en la siguiente forma JSON En texto plano:

{"Mapa":"\"Nuevo Oriente\"","version":3.1,"tierra1":[{"arbol1":{"hoja1": 123,"hoja2": 124},"colina":{"flor":{"petalo1": 187,"petalo2": 125,"petalo3": 167,"petalo4": 120}}}]}

Ha pedido a sus compañeros que lancen diferentes llamadas a su endpoint para comprobar su comportamiento. Una de las llamadas recibidas tiene la siguiente forma (para guardar un nuevo mapa):

Analizar las diferencias con la especificación.

{'Mapa':'NuevoOriente',version:'3.1',tierra1:[{arbol1:{hoja1:123,hoja2:124},colina:{flor{petalo:187,petalo:125,167}}}]}

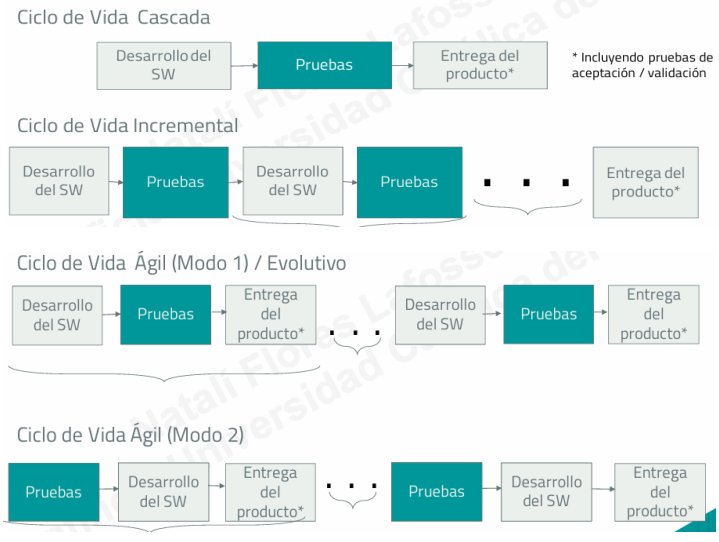
◉ ¿Tiene la forma adecuada?
No. La forma no es adecuada porque utiliza comillas simples en lugar de dobles para delimitar las cadenas. En JSON, las cadenas deben estar delimitadas por comillas dobles.
◉ ¿Los parámetros en cadena están encomillados?
No todos. El valor de "version" está entre comillas simples en lugar de dobles. Además, los nombres de los elementos dentro del arreglo "tierra1" (como "arbol1", "hoja1", "colina", "flor", "petalo") deberían estar entre comillas dobles.
◉ ¿Los caracteres de escape necesarios están incluidos?
No. En el nombre del mapa "Nuevo Oriente" en la especificación JSON, se utilizan caracteres de escape para las comillas dobles dentro de la cadena. La llamada recibida omite estos caracteres de escape.

Otra llamada recibida tiene la siguiente forma (para guardar un nuevo mapa):
{"Mapa":"'; DROP TABLE maps; DROPTABLEmapas;","version":3.1,"tierra1":[{"arbol1":{"hoja1":123,"hoja2":124},"colina":{"flor":{"petalo1": 187,"petalo2": 125,"petalo3": 167,"petalo4": 120}}}]}

◉ ¿Qué está queriendo hacer esta llamada?
Esta llamada está intentando realizar un ataque de inyección SQL. El valor del parámetro "Mapa" contiene código SQL malicioso (DROP TABLE maps; DROPTABLEmapas;) que, si se ejecuta en una base de datos, eliminaría las tablas llamadas "maps" y "mapas".
◉ ¿Cómo evitarlo?
Validación de entradas: Verificar que los datos recibidos en el endpoint cumplen con el formato y los tipos de datos esperados. En este caso, se debería validar que el valor de "Mapa" sea una cadena de texto que no contenga código SQL.
Sanitización de entradas: Escapar o eliminar caracteres especiales que puedan ser interpretados como código SQL.
Utilizar consultas parametrizadas: En lugar de concatenar directamente los datos recibidos en la consulta SQL, utilizar consultas parametrizadas donde los valores de los parámetros se pasan por separado. Esto evita que el código malicioso inyectado sea interpretado como parte de la consulta.
Implementar un firewall de aplicaciones web (WAF): Un WAF puede ayudar a detectar y bloquear ataques comunes como la inyección SQL.

TDD y Desarrollo Ágil

Concepto de Ciclo de Vida del SW



Algunos Conceptos Ágiles

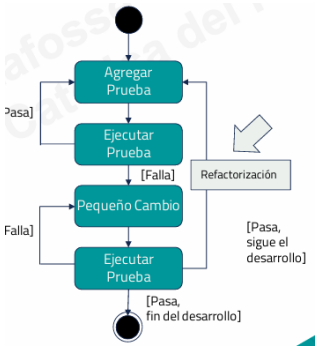
◉ **Pequeños Releases:** Se entrega un SW funcional después de cada Sprint.
◉ **Pruebas de Aceptación:** Cada historia de usuario (requisito) debe tener una (o más) pruebas de aceptación.
◉ **Propiedad Colectiva:** No existe dueño del código. Y Todos pueden modificarlo.
◉ **Espacio informativo:** Todos tienen acceso a la información del producto y proyecto
◉ **Integración Continua**: Se compila/integra varias veces al día.
◉ **Desarrollo (Diseño) dirigido por Pruebas:**
o Técnica de diseño, funcionalidades requeridas.
o Minimizar el número de errores.
o Desarrollo modular y reutilizable.
o TDD: TFD + Refactorización
o TFD: Test-First Development

◉ **Diseño Emergente:** El diseño surge de la revisión y refactorización del código. Y No hay un gran diseño anticipado (aparte de la Arquitectura)
◉ **Refactorización:**
○ Desarrollo iterativo -> diseño simple
○ Mejora Continua del Diseño
○ Procesos se enfocan en: ▪ Remover duplicaciones ▪ Incrementar Cohesión (modularidad) ▪ Disminuir acoplamiento

Desarrollo dirigido por pruebas

Para aplicarlo, se sigue una serie de pasos:

1. Escribir test (según requisitos)
2. Escribir test (según requisitos)
3. Escribir sólo lo necesario
4. Verificar que pase el test
5. Refactorizar código
6. Repetir



Beneficios del T.D.D

❑ Mayor calidad del SW
❑ Mayor cobertura de código
❑ Código reutilizable
❑ Comunicación entre el equipo
❑ Tests como documentación
❑ Integración continua asegura regresión.
❑ Se evita trabajo innecesario.

Limitaciones y Problemática

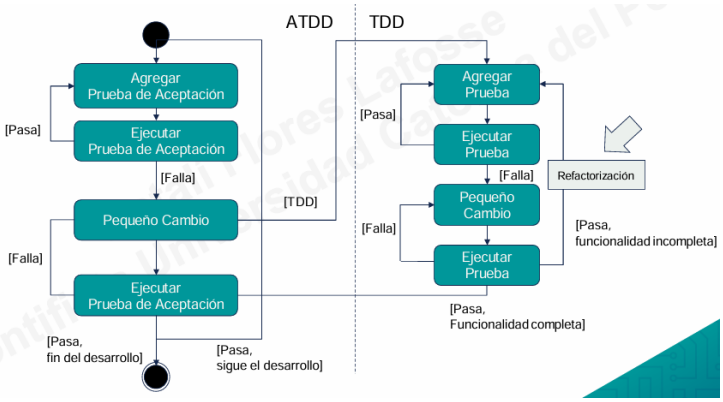
❑ Alta dependencia entre tests y estructuras internas
❑ Nuevos requerimientos →tests inmantenibles.
❑ Abandono de los tests.
❑ El ahorro de tiempo → del diseño de tests
❑ Equipo con habilidades para crear pruebas
❑ La prueba es inservible si se interpreta mal el requerimiento
❑ Las pruebas se hacen largas y complejas. (Escalabilidad)

Desarrollo dirigido por comportamientos (BDD)

❑ B.D.D: Behavior driven development
❑ A.T.D.D: Acceptance T.D.D.
❑ Ambas son consideradas sinónimos.

El B.D.D. se centra en el comportamiento del sistema (más alto nivel)
Se basa en escenarios completos, especificados por el patrón:
❖ Given (Dado): Estado inicial
❖ When (Cuándo): Acción/Evento
❖ Then (Entonces): Cambios.

Se relaciona con el patrón “Role-Feature-Reason” (Historias de Usuario)
❖ As a (Como): Rol del usuario
❖ I want (Quiero): Necesidad
❖ So that (Entonces): Razones de la necesidad



T.D.D - Unit Testing – B.D.D.

Los tres términos se encuentran interrelacionados:

- ❑ Prueba Unitaria: Única funcionalidad o módulo
- ❑ TDD: Desarrollo a partir de las pruebas
- ❑ BDD: Desarrollo a partir de comportamiento

Las pruebas unitarias te dicen qué probar, TDD cuándo probarlo y BDD cómo probarlo.

Preguntas

¿Cómo se diferencia un ciclo incremental de uno ágil?

A. Son iguales

B. Solo en el orden de las pruebas

C. Al terminar cada Sprint (en ágil), ya se tiene un elemento funcional, lo cual no se cumple en el incremental.

Si bien ambos ciclos dividen el desarrollo en etapas, la principal diferencia radica en que Agile se enfoca en la entrega de valor y funcionalidad al cliente al final de cada Sprint, mientras que Incremental se centra en construir el producto en incrementos, sin la obligación de que cada incremento sea funcional por sí solo.

¿Se puede aplicar T.D.D en un sistema legacy?

Aplicar TDD (Test-Driven Development) en sistemas legacy puede ser beneficioso para mejorar la calidad del código, reducir riesgos y facilitar la refactorización. Escribir pruebas antes del código ayuda a identificar errores temprano y refactorizar con mayor confianza. Sin embargo, los sistemas legacy suelen carecer de pruebas, tener código complejo y encontrar resistencia al cambio. Para aplicar TDD en estos sistemas se recomienda comenzar gradualmente, priorizar áreas críticas, refactorizar el código legacy y utilizar herramientas de prueba.

¿Por qué se habla de un diseño emergente?

A. Porque el diseño detallado va surgiendo de las iteraciones de TDD.

B. Porque no se define una arquitectura.

C. Porque el diseño no es importante en TDD.

El diseño emergente es un concepto clave en TDD (Test-Driven Development). No se trata de no definir una arquitectura o de que el diseño no sea importante, sino de un enfoque iterativo donde el diseño se refina y mejora a medida que se avanza en el desarrollo guiado por las pruebas.

¿Cómo relacionamos las historias de usuario con el TDD?

A. No están relacionadas.

B. Cada historia debe tener criterios de aceptación, cada uno de los cuales genera una o más pruebas para TDD.

C. Las historias de usuario se usan para generarla arquitectura previa al TDD.

1.Definición de pruebas: Los criterios de aceptación de una historia de usuario se traducen en pruebas unitarias en TDD. Cada criterio debe generar al menos una prueba que verifique su cumplimiento.
2.Diseño guiado por pruebas: Al escribir las pruebas primero, se piensa en la funcionalidad desde la perspectiva del usuario y se define el comportamiento esperado. Esto ayuda a guiar el diseño del código y a asegurar que se cumplan los requisitos de la historia de usuario.
3.Validación de la funcionalidad: Las pruebas creadas a partir de los criterios de aceptación sirven para validar que la funcionalidad implementada cumple con lo que el usuario espera.

Las opciones A y C son incorrectas:
A. No están relacionadas: Las historias de usuario y TDD están estrechamente relacionadas en el desarrollo ágil.
C. Las historias de usuario se usan para generarla arquitectura previa al TDD: Las historias de usuario no se usan para generar la arquitectura, aunque pueden influir en ella. La arquitectura se define en etapas previas del desarrollo.