Testing

Proceso destructivo, obj: encontrar defectos para posteriormente resolverlos y contribuir a la calidad del sw

Asumimos que los defectos existen.

Cuanto testing es suficiente?

- Umbral
- Criterio económico

Error vs Defecto (Pregunta de final)

Son fallas en un producto de sw. Depende en que momento se encuentra o se produce

Cuando encontramos la falla en la misma etapa que la origino, es un error

Cuando trasciende la etapa, es un defecto

El testing busca encontrar DEFECTOS

El obj del testing no es encontrar errores, el testing se ejectua posteriormente a la implementacion y brinda una retroalimentacion.

La idea es encontrar las fallas cuando son errores que son mas baratos.

Proceso de Testing

Planificacion y control -> Analisis y diseño -> Ejecucion -> Evaluacion y Reportes

Niveles de testing

1- Testing unitario: busca proveer un mecanismo de verificacion de comprension de la funcionalidad. Garantizo que la funcionalidad cumple con el requerimiento dado. Lo lleva adelante el DESARROLLADOR

2- Testing de integración: buscamos encontrar defectos. prueba las interfaces entre los componentes del sw. bottom up, top down, sandwich. Lo lleva adelante el TESTER

3- Testing de Sistema: Probamos una funcionalidad en su totalidad. Ejecutamos un escenario con ciertas caracteristicas para encontrar si el resultado coincide con el esperado. Lo lleva adelante el TESTER

4- Testing de Aceptación: No deberiamos encontrar defectos. Es validar que lo que entregamos corresponde a el requerimiento planteado. Es de alguna forma capacitar al cliente en la funcionalidad. Es lo mas similar posible al productivo. Lo lleva adelante el PRODUCT OWNER (en agil) o el USER/CLIENTE

Estrategias

Criterio económico: Maximizar la cantidad de defectos minimizando el esfuerzo requerido

Casos de prueba: Artefacto del testing. Contiene condiciones y pasos que se deben ejecutar para garantizar que el resultado esperado sea o no igual al resultado obtenido y de esa forma encontrar defectos.

Diseñamos casos de prueba economica e inteligentemente (Minimizar casos de prueba y maximizar defectos encontrados)

2 ESTRATEGIAS

Caja blanca: podemos ver el detalle de la implementacion de la funcionalidad (codigo). Diseñar caso de prueba para garantizar la cobertura.

Caja negra: analizo en terminos de entradas y salidas, no conozco la fucionalidad interna.

Elijo los valores con los que ejecuto la funcionalidad y analizo las salidas (resultado obtenido)

Métodos (dentro de las estrategias)

Caja negra

-Basado en especificaciones:

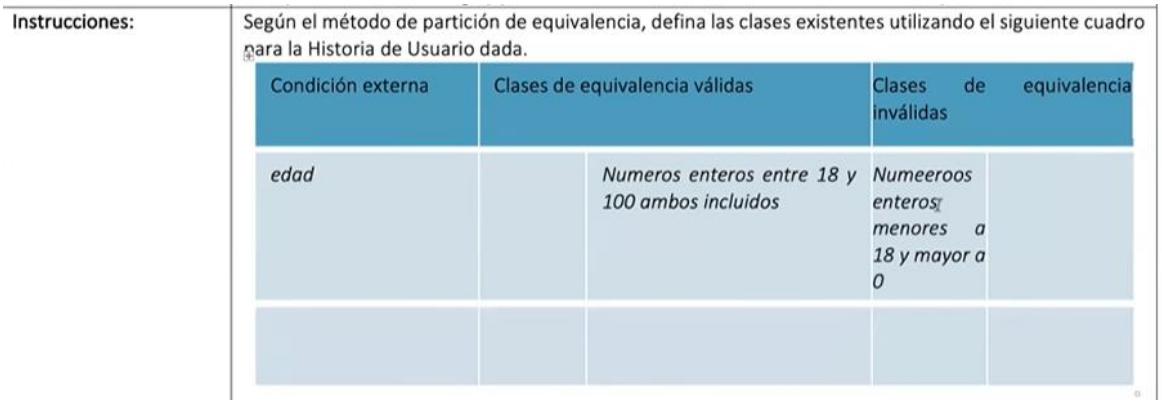
.Partición de equivalencias: analizar las condiciones externas involucradas en la funcionalidad (entradas y salidas).

1) Identificar las clases de equivalencia: Para c/u analizo los subconjuntos de valores posibles que pueden tomar que producen un resultado equivalente. Casos: Rango de valores continuos, valores discretos, selección simple, selección múltiple.

2) Identificar los casos de prueba: tomo de cada condición externa una clase de equivalencia y elijo un valor representativo para formar el caso de prueba. Cualquier valor que tome por clase de equivalencia me va a dar un resultado equivalente por eso se toma un solo valor.

Ejemplo: web de MAE bebidas. te pregunta cuál es tu edad. Condiciones externas: Edad de la persona (entrada), Ingreso/No ingreso (salida). A la edad la podemos dividir en subconjuntos de valores posibles donde cada uno va a producir un resultado equivalente. Por lo menos 2 clases de equivalencia validas -> mayor igual 18 y menor 18. Hay clases de equivalencia no validas, en este caso, texto, vacío, números no enteros, símbolos, etc. La división se basa en los resultados posibles/mensajes de error asociados. Un caso de prueba seria: Ingresar a la web con una edad que supere los 18. Resultado esperado: Ingreso valido. El único valor que elegí fue 18 que está dentro de la clase de equivalencia "mayor igual a 18", no necesito usar más valores dentro de esa clase porque se supone que el resultado es equivalente -> me deja ingresar a la web

Ejemplo del práctico



Instrucciones:	Según el método de partición de equivalencia, defina las clases existentes utilizando el siguiente cuadro para la Historia de Usuario dada.				
	Condición externa	Clases de equivalencia válidas		Clases de equivalencia inválidas	
	edad		*Numeros enteros entre 18 y 100 ambos incluidos*	*Numeeroos enteros menores a 18 y mayor a 0*	

Id del Caso de Prueba	Prioridad (Alta, Media, Baja)	Nombre del Caso de Prueba	Precondiciones	Pasos	Resultado esperado
			El usuario "juan" esta logueado con permisos de admninistrador. La fecha actual es 15/5/2020	1. El cliente ingresa a la opción "Ingresar" 2. El cliente ingresa "18" en el campo de "edad" 3. El cliente selecciona l aopción ingresar	1. Se muestra el mensaje "Bienvenido al sitio web"

Prioridad (orden que le voy a dar a la ejecución de los casos de prueba para encontrar defectos)
Alta: Camino feliz.
Media: Todo lo que está al medio (lo dijo textual asi xd). Combinaciones de valores que pueden producir fallas/caminos infelices.
Baja: Validaciones, valores no ingresados, no corresponden con el formato.

Nombre: Describe el escenario que estamos probando

Precondiciones: conjunto de valores/características que tiene que tener el contexto para llevar adelante el caso de prueba. Ejemplo: el usuario "juan" esta logueado con permisos de admin. TIENEN QUE SER SIGNIFICANTES PARA EL RESULTADO DEL CASO DE PRUEBA. Tienen que tener VALORES CONCRETOS.

Pasos: conjunto de operaciones numeradas cuyo formato es: 1- El ROL ingresa la opción "x" 2- El ROL ingresa "x" en el campo de "y"… etc

Resultado esperado: salida esperada.

Que es una clase/partición de equivalencia? (Pregunta de final)

Subconjunto de valores que puede tomar una condición externa para el cual, si yo tomo cualquier miembro de ese subconjunto el resultado de la ejecución de la funcionalidad es EQUIVALENTE.

.Análisis de valores límites: Es una variante de la partición de equivalencias. La mayor cantidad de defectos se encuentran en los extremos de los intervalos. A la hora de escribir el caso de prueba, en vez de tomar CUALQUIER valor de la clase de equivalencia, tomamos uno que se encuentra en los límites.

-Basados en la experiencia

.Adivinanza de defectos

.Testing exploratorio

Nexus

Es un marco que amplía Scrum para coordinar de 3 a 9 equipos Scrum que trabajan sobre un mismo producto. Todos comparten un solo Product Owner y un único Product Backlog. Define roles, eventos y artefactos adicionales que integran y sincronizan el trabajo de los equipos, con el fin de entregar un Incremento Integrado del producto. Se basa en Scrum y lo extiende solo lo necesario para facilitar la colaboración entre múltiples equipos

Teoria

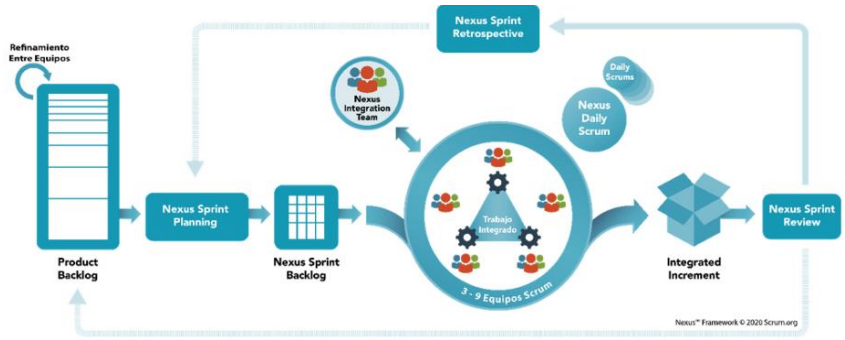
Busca mantener los principios de Scrum (empirismo, autogestión e inteligencia colectiva) mientras permite que varios equipos trabajen juntos en un único producto para generar más valor. Su objetivo es reducir la complejidad y las dependencias entre equipos, facilitando la entrega de un Incremento Integrado de producto útil en cada Sprint. Nexus hace visibles las dependencias que surgen por la **estructura del producto** (grado al cual diferentes asuntos están separados independientemente en el producto) y la **estructura de la comunicación** (forma de comunicación entre las personas dentro y entre los equipos), y ofrece mecanismos para ajustarlas y mejorar la colaboración. Además, destaca que escalar no siempre implica sumar personas, ya que esto puede aumentar la complejidad; a veces, reducir el tamaño del grupo permite entregar más valor.

Nexus introduce mejoras en tres áreas:

Responsabilidades: incorpora el Nexus Integration Team, formado por el Product Owner, un Scrum Master y miembros del Nexus, responsable de asegurar un Incremento Integrado útil en cada Sprint.

Eventos: adapta o amplía los eventos de Scrum para coordinar el trabajo conjunto de todos los equipos, estableciendo un Objetivo de Sprint común (Nexus Sprint Goal).

Artefactos: todos los equipos comparten un único Product Backlog. Se utiliza un Nexus Sprint Backlog para mantener la transparencia del trabajo y un Incremento Integrado que representa el resultado combinado de todos los equipos del Nexus.



Responsabilidades

Un Nexus está compuesto por varios Scrum Teams que trabajan juntos hacia un mismo Objetivo de Producto. Además de los roles definidos en Scrum (Developers, Product Owner y Scrum Master), Nexus introduce una nueva responsabilidad: el Nexus Integration Team (NIT).

El Nexus Integration Team se encarga de asegurar que en cada Sprint se entregue un Incremento Integrado valioso y funcional. Actúa como punto focal para resolver problemas de integración entre equipos, abordando tanto restricciones técnicas como de comunicación.

Está formado por:

Product Owner: único para todo el Nexus, responsable del valor del producto y de la gestión del Product Backlog.

Scrum Master: garantiza que el marco Nexus se entienda y aplique correctamente; puede desempeñar el rol también en equipos Scrum individuales.

Miembros del Nexus Integration Team: suelen ser integrantes de los Scrum Teams, con habilidades para acompañar, guiar y ayudar a mejorar prácticas de integración.

La membresía en el NIT tiene prioridad sobre la pertenencia a un Scrum Team individual, garantizando que los problemas que afectan a varios equipos se resuelvan primero.

Eventos

En Nexus, los eventos de Scrum se amplían o adaptan para coordinar el trabajo entre varios equipos. Su duración se basa en los eventos originales de Scrum, añadiendo tiempo adicional cuando es necesario, y participan solo las personas requeridas para lograr el objetivo de cada evento.

Los principales eventos Nexus son:

Sprint: igual que en Scrum, pero todos los equipos del Nexus trabajan juntos para entregar un único Incremento Integrado.

Refinamiento Entre Equipos: actividad continua destinada a reducir o eliminar dependencias entre los equipos. Consiste en descomponer el Product Backlog para identificar qué equipo se encargará de cada ítem y hacer visibles las dependencias. Cada equipo puede luego refinar individualmente sus ítems para tenerlos listos en la Nexus Sprint Planning.

Nexus Sprint Planning: coordina el trabajo de todos los Scrum Teams del Nexus para un Sprint. Se definen:

- Un Objetivo de Sprint del Nexus, alineado con el Objetivo del Producto.
- Un Objetivo de Sprint por equipo, alineado con el del Nexus.
- Un Nexus Sprint Backlog, que muestra las dependencias entre equipos.
- Un Sprint Backlog individual por equipo.

Nexus Daily Scrum: reunión diaria entre representantes de los equipos para detectar problemas de integración, revisar el progreso y coordinar acciones. Las Daily Scrum de cada equipo complementan esta reunión, enfocándose en resolver los problemas detectados.

Nexus Sprint Review: se realiza al final del Sprint para presentar el Incremento Integrado a los interesados, obtener retroalimentación y decidir adaptaciones futuras. Sustituye las revisiones individuales de cada equipo.

Nexus Sprint Retrospective: busca mejorar la calidad y eficacia del Nexus. Se inspeccionan interacciones, procesos, herramientas y la Definición de Terminado. Las retrospectivas de los equipos complementan esta revisión general aportando observaciones desde la práctica diaria.

Este evento cierra el Sprint y establece acciones de mejora para el siguiente ciclo.

Artefactos y compromisos

En Nexus, los artefactos amplían los de Scrum para mantener la transparencia y asegurar que todo el Nexus trabaje hacia un mismo objetivo. Cada artefacto tiene un compromiso asociado que refuerza el empirismo y la entrega de valor.

Product Backlog: único para todo el Nexus; contiene el trabajo necesario para mejorar el producto y debe gestionarse considerando las dependencias entre equipos.

Compromiso: Objetivo del Producto, que define la meta a largo plazo y la dirección general del Nexus.

Nexus Sprint Backlog: conjunto del Objetivo de Sprint del Nexus y los ítems de los Sprint Backlogs de cada equipo. Permite visualizar dependencias y el flujo de trabajo durante el Sprint.

Compromiso: Objetivo de Sprint del Nexus, que unifica los esfuerzos de todos los equipos en un propósito común, evitando que trabajen de forma aislada.

Integrated Increment: suma de todo el trabajo integrado y completado por el Nexus. Debe cumplir con los estándares de calidad y puede presentarse en la Nexus Sprint Review o antes.

Compromiso: Definición de Terminado (DoD), que establece los criterios de calidad y completitud que todos los equipos deben cumplir. Los equipos pueden aplicar una versión más estricta, pero nunca menos rigurosa que la del Nexus.

La transparencia de los artefactos es esencial: información incompleta puede llevar a decisiones erróneas, y sus consecuencias se amplifican cuando se trabaja a gran escala.

TESTING AGIL

Prácticas de prueba colaborativas que ocurren desde el inicio hasta la entrega, apoyando la entrega frecuente de productos de calidad que aportan valor de negocio a nuestros clientes. Las actividades de prueba se enfocan en la prevención de defectos en lugar de la detección de defectos, y trabajan para fortalecer y respaldar la idea de la responsabilidad del equipo completo por la calidad.

Las pruebas ágiles incluyen (pero no se limitan a) las siguientes actividades: guiar el desarrollo con ejemplos concretos, hacer preguntas para probar ideas y suposiciones, automatizar pruebas, realizar pruebas exploratorias, probar atributos de calidad como el rendimiento, la confiabilidad y la seguridad.

Equipos ágiles

En los equipos ágiles, se distinguen dos grupos principales: el equipo del cliente y el equipo de desarrollo, diferenciados por sus habilidades para entregar un producto.

Equipo del Cliente:

Incluye expertos en negocio, dueños de producto, analistas, gerentes y expertos en dominio. Su función es redactar historias o características que el equipo de desarrollo implementa, proporcionando ejemplos y pruebas orientadas al negocio. También colaboran estrechamente con los desarrolladores durante las iteraciones, respondiendo preguntas y revisando historias completadas. Los testers en este equipo ayudan a expresar requisitos en forma de pruebas.

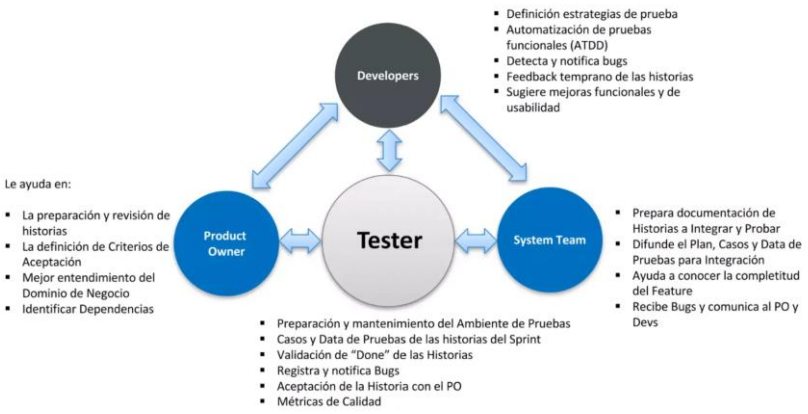
Equipo de Desarrollo:

Comprende a todos los involucrados en la entrega de código: programadores, administradores de sistemas, arquitectos, escritores técnicos, entre otros. Aunque se fomenta la polivalencia, cada equipo decide las especializaciones necesarias. Los testers también son parte de este equipo, asegurando la calidad en representación del cliente y ayudando a maximizar el valor del negocio.

Interacción entre equipos:

Ambos equipos trabajan estrechamente con el objetivo común de entregar valor a la organización. Las iteraciones, que duran de 1 a 4 semanas, involucran la priorización de historias por el equipo del cliente y la implementación por el equipo de desarrollo, definiendo requisitos con ejemplos y pruebas. Los testers, que pueden estar en ambos equipos, son cruciales para escribir pruebas, automatizar regresiones y asegurar el éxito del proyecto.

En equipos ágiles, aunque no haya roles específicos de testers, todos son responsables de las tareas de pruebas, ya que estas habilidades son esenciales para el éxito del proyecto.



Manifiesto del testing

Testing throughout over at the end (a lo largo y no al final del proceso)

Las pruebas no son una fase final, sino una actividad continua que avanza junto con otras tareas. Es útil añadir una columna de verificación para revisiones por compañeros.

Preventing bugs over finding bugs

Aclarar todas las suposiciones y hacer preguntas antes de escribir cualquier línea de código, asegurando que todos tengan el mismo entendimiento del requerimiento.

Testing understanding over checking functionality

Automatizar tareas simples y enfocarse en defender y representar al cliente, asegurando que el producto cumpla con sus necesidades reales más allá de las especificaciones.

Building the best system over breaking the system

Deja de ser un proceso destructivo, ayuda a construir el mejor sistema

Team responsibility for quality over tester responsibility

La calidad del software depende de todo el equipo, no solo del tester.

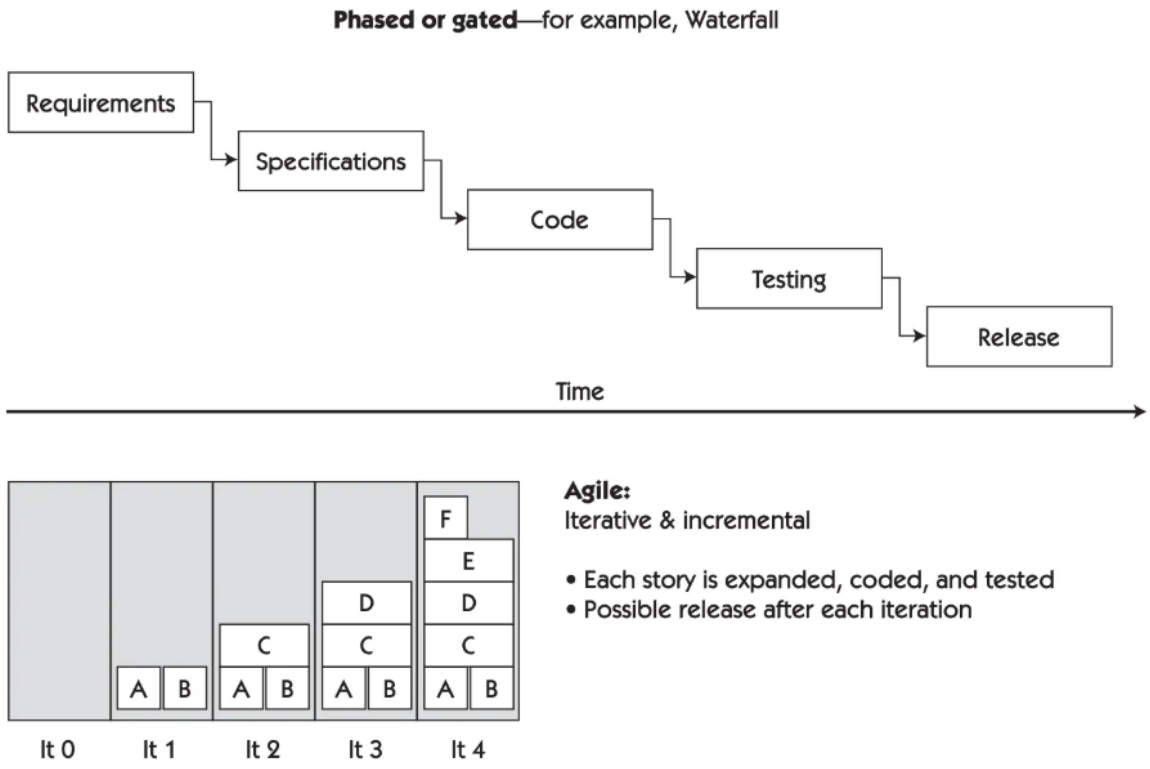
Testing Tradicional vs Agile

En el desarrollo tradicional, las pruebas suelen realizarse al final del proceso, justo antes del lanzamiento, lo que frecuentemente se traduce en tiempo limitado para probar debido a retrasos en el desarrollo. En cambio, en Agile, las pruebas son iterativas e incrementales, realizándose en cada ciclo de desarrollo. Los testers verifican el código tan pronto como está terminado, asegurándose de que las historias no se consideren "completadas" hasta que pasen las pruebas.

En Agile, las pruebas no se basan únicamente en documentos de requisitos predefinidos; en su lugar, se crean de manera colaborativa con expertos de negocio y otros miembros del equipo, a menudo justo antes de codificar. Además de pruebas funcionales detalladas, los testers realizan pruebas exploratorias manuales para detectar errores importantes y colaboran en la automatización y ejecución de casos de prueba.

A diferencia del desarrollo tradicional, las pruebas de rendimiento, seguridad y otros aspectos se integran tempranamente en el proceso para influir en el diseño y la codificación. En Agile, no se deja todo para el final: actividades como pruebas de aceptación, corrección de errores y despliegues se realizan de forma continua o al final de cada iteración.

El feedback rápido de las pruebas impulsa el proyecto y evita bloqueos por hitos no alcanzados. Aunque algunos testers pueden resistirse a Agile, temiendo falta de disciplina o caos, los equipos ágiles buscan calidad repetible y eficiencia, haciendo de este enfoque un entorno positivo para los testers.



Principios para Testing en Proyectos Ágiles

1. Testing se mueve hacia adelante en el proyecto
2. Testing no es una fase
3. Todos hacen testing
4. Reducir la latencia del feedback
5. Las pruebas representan expectativas
6. Mantener el código limpio, corregir los defectos rápido
7. Reducir la sobrecarga de documentación de las pruebas
8. Las pruebas son parte del “done”
9. De probar al final a Conducido por Pruebas

Prácticas Concretas

1. Pruebas unitarias e integración automatizadas
2. Pruebas de regresión a nivel de sistema automatizadas
3. Pruebas exploratorias
4. TDD
5. ATDD (Acceptance Test-driven Development)
6. Control de versión de las pruebas con el código

Prácticas Continuas

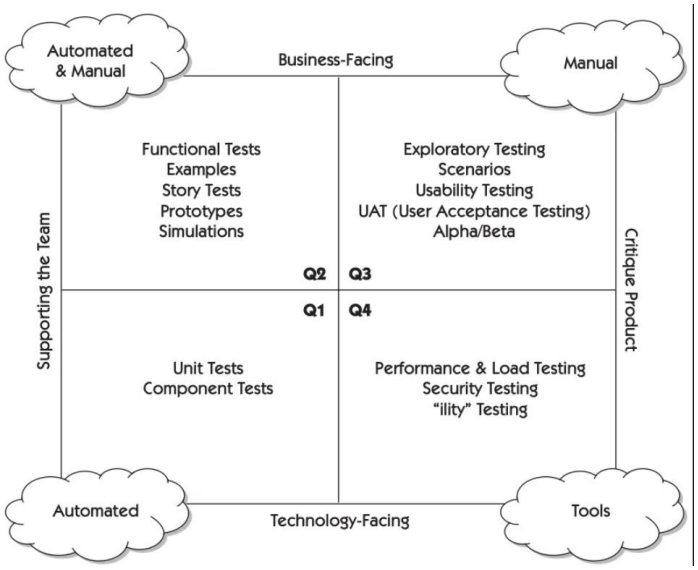
1. CI: Integración continua. Automatizamos nivel de integración
2. CD: Delivery continuo. Automatizamos nivel de sistema
3. CD: Deployment continuo. Automatizamos nivel de aceptación

Desafíos Culturales

Resumen del libro:

- Considera la cultura organizacional antes de realizar cualquier cambio.
- Los testers tienen un tiempo más fácil para integrarse en equipos ágiles cuando toda la organización valora la calidad, pero los testers con una mentalidad de "policía de calidad" tendrán dificultades.
- Algunos testers podrían tener problemas para adaptarse a la propiedad de la calidad del "equipo completo", pero un enfoque de equipo ayuda a superar las diferencias culturales.
- Los equipos de atención al cliente y los equipos de desarrollo deben trabajar estrechamente, y mostramos cómo los testers pueden ser clave para facilitar esta relación.
- Las grandes organizaciones que tienden a tener equipos de especialistas más aislados enfrentan desafíos culturales particulares en áreas como la comunicación y la colaboración.
- Las principales barreras para el éxito de los testers en la adopción ágil incluyen el miedo, la pérdida de identidad, la falta de capacitación, experiencias negativas previas con nuevos procesos de desarrollo y diferencias culturales entre roles.
- Para ayudar a introducir cambios y promover la comunicación, sugerimos animar a los miembros del equipo a discutir sus miedos y celebrar cada éxito, por pequeño que sea.
- Directrices como una "Carta de Derechos del Tester" brindan a los testers confianza para plantear problemas y ayudarles a sentirse seguros mientras aprenden y prueban nuevas ideas.
- Los gerentes enfrentan sus propios desafíos culturales, y necesitan proporcionar apoyo y capacitación para ayudar a los testers a tener éxito en los equipos ágiles.
- Los testers pueden ayudar a los equipos a satisfacer las expectativas de los gerentes al proporcionar la información que necesitan para rastrear el progreso y determinar el ROI.

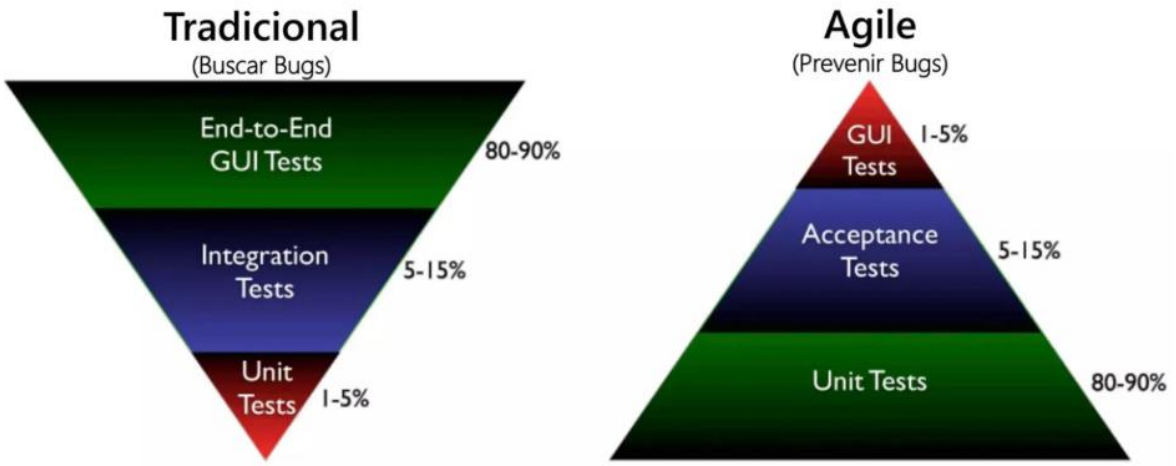
Cuadrantes del Testing Ágil



En un eje dividimos la matriz en tests que soportan al equipo y tests que critican el producto. El otro eje los divide en pruebas de negocio y de tecnología.

1. **Cuadrantes 1 y 2: Pruebas que apoyan al equipo**
   - **Cuadrante 1**: Incluye pruebas técnicas como TDD, pruebas unitarias y de componentes. Estas pruebas son automatizadas, están orientadas al código interno, y ayudan a diseñar y mantener la calidad del sistema desde el inicio.
   - **Cuadrante 2**: Pruebas orientadas al negocio, como pruebas funcionales y de aceptación derivadas de ejemplos proporcionados por los clientes. Estas definen la calidad externa y aseguran que el producto cumple con los requisitos del cliente. También suelen ser automatizadas y proporcionan retroalimentación rápida al equipo.
2. **Cuadrantes 3 y 4: Pruebas que critican el producto**
   - **Cuadrante 3**: Pruebas manuales orientadas al negocio, como pruebas de aceptación de usuarios (UAT) y pruebas exploratorias. Evalúan si el producto cumple con las expectativas del cliente y ofrecen oportunidades para identificar errores críticos.
   - **Cuadrante 4**: Pruebas técnicas que evalúan atributos como rendimiento, seguridad y robustez. Estas pruebas suelen requerir herramientas especializadas y deben ser planificadas desde el inicio del desarrollo para evitar problemas al final.

Pirámide del Testing Ágil



Antes de la popularidad de metodologías ágiles como Scrum, ya se sabía que las pruebas debían automatizarse, pero a menudo no se hacía debido a los altos costos y a que se escribían mucho tiempo después de desarrollar las funcionalidades. Un error común era automatizar pruebas en niveles equivocados.

La **pirámide de pruebas automatizadas** propone tres niveles que permiten una estrategia eficaz y rentable:

1. **Base: Pruebas unitarias**

Constituyen la mayor parte de la pirámide y son la base sólida de una estrategia de automatización. Estas pruebas, escritas generalmente en el mismo lenguaje del sistema, permiten detectar errores específicos en el código, brindando retroalimentación rápida y detallada. Son económicas y fáciles de mantener, especialmente cuando se usa TDD (Test-Driven Development). Esta capa tiene el mayor ROI (retorno de inversión) y garantiza la solidez del sistema, ya que automatiza los componentes tecnológicos básicos.

2. **Capa intermedia: Pruebas de servicios**

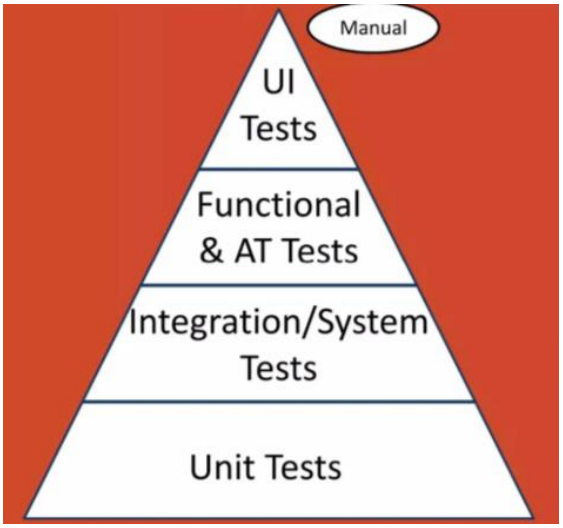
Aquí se prueban las funcionalidades del sistema de forma independiente de la interfaz gráfica, como las APIs o servicios específicos (por ejemplo, “multiplicar” y “dividir” en una calculadora). Estas pruebas son más económicas y menos frágiles que las de la interfaz, ya que no dependen de cambios en el diseño gráfico o elementos HTML. Se diseñan para garantizar que se construyan las funcionalidades correctas, verificando escenarios de negocio y operaciones más amplias que las pruebas unitarias.

3. **Cima: Pruebas de interfaz de usuario (UI)**

Estas pruebas, ubicadas en la parte superior de la pirámide, deben minimizarse porque son costosas, lentas y frágiles. Involucran la interacción directa con la interfaz gráfica para probar flujos completos, pero cualquier cambio en la UI (como renombrar un elemento) puede romperlas. Aunque son útiles para validar la experiencia del usuario, ofrecen el menor ROI y suelen llevar mucho más tiempo ejecutarlas.

Además de estas capas, es esencial incluir **pruebas manuales** como las exploratorias y de aceptación para validar casos que la automatización no cubre bien. Sin embargo, la mayor parte de las pruebas de regresión debe automatizarse para maximizar el retorno de inversión.

El enfoque tradicional, con pirámides invertidas (priorizando pruebas de interfaz), lleva a sistemas con pruebas costosas y frágiles. En cambio, los equipos ágiles trabajan en fortalecer la base con TDD y pruebas unitarias, integrando pruebas funcionales en la capa de servicios y reduciendo las pruebas de UI. Esto fomenta una estrategia más robusta y colaborativa, donde programadores y testers trabajan juntos para lograr automatización efectiva en todos los niveles.



Niveles de la pirámide de pruebas automatizadas:

1. **Unit Tests (Pruebas Unitarias):**

Constituyen la base de la pirámide. Verifican el funcionamiento de pequeñas unidades de código (como funciones o métodos) de manera aislada. Son rápidas, económicas, y fáciles de mantener. Proveen retroalimentación precisa y son esenciales para detectar errores en etapas tempranas del desarrollo.

2. **Integration/System Tests (Pruebas de Integración/Sistema):**

Se enfocan en probar la interacción entre componentes o módulos del sistema. Aseguran que las partes trabajen juntas correctamente. Aunque más lentas que las pruebas unitarias, son importantes para validar la cohesión del sistema.

3. **Functional & Acceptance Tests (Pruebas Funcionales y de Aceptación):**

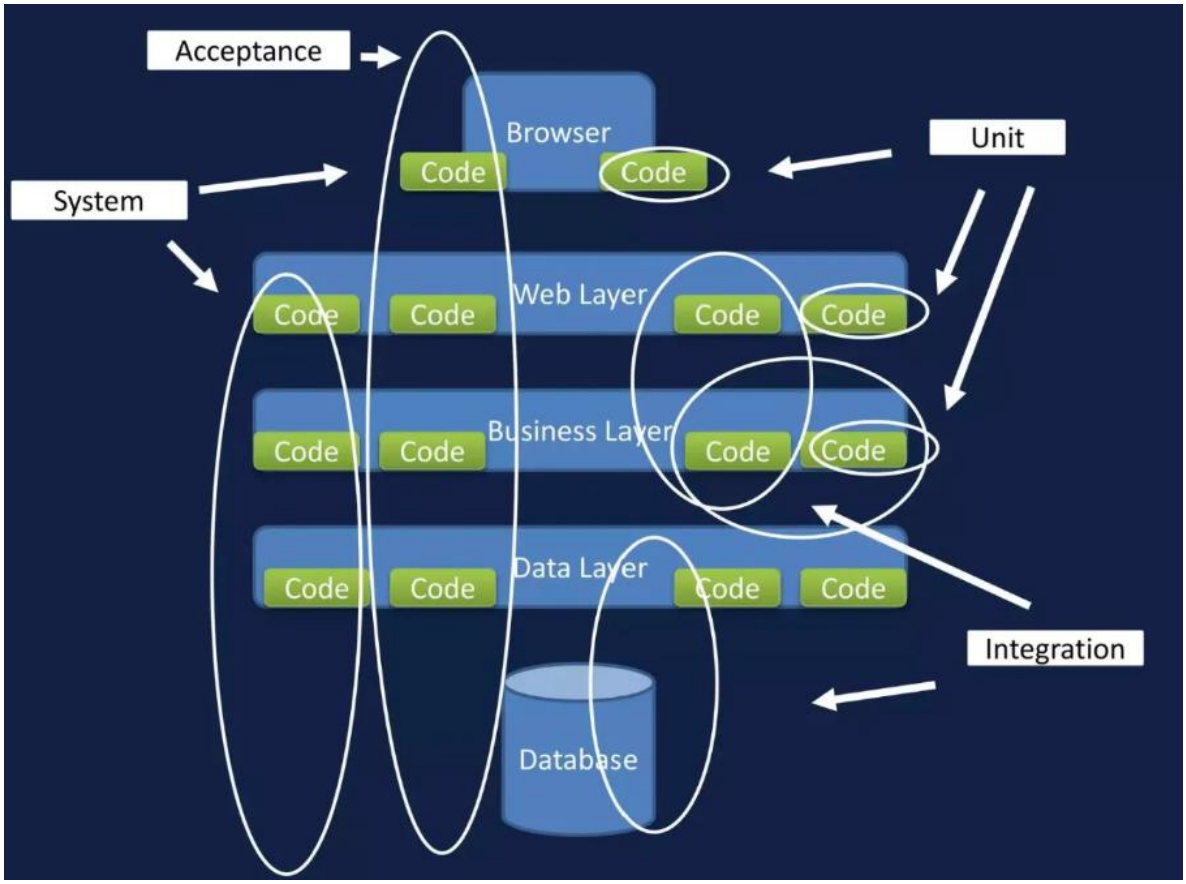
Verifican que el sistema cumple con los requisitos de negocio y las expectativas del cliente. Operan a nivel de API o detrás de la interfaz gráfica, siendo menos costosas y frágiles que las pruebas de UI. Son ideales para escenarios de usuario específicos.

4. **UI Tests (Pruebas de Interfaz de Usuario):**

Están en la cima de la pirámide y deben ser limitadas. Simulan interacciones de usuario con la interfaz gráfica. Son las más lentas, frágiles y costosas, pero necesarias para validar la experiencia del usuario y flujos completos.

5. **Manual (Pruebas Manuales):**

Representan actividades no automatizadas, como pruebas exploratorias o de aceptación final. Complementan la automatización para garantizar la calidad en aspectos no fácilmente automatizables.



1. **Unit Tests (Pruebas Unitarias):**

   Se realizan a nivel de código individual dentro de cada capa (Web Layer, Business Layer, Data Layer). Validan funcionalidades específicas y aisladas, como funciones o métodos.

2. **Integration Tests (Pruebas de Integración):**

   Verifican la interacción entre capas adyacentes, como la integración entre la Business Layer y la Data Layer, o entre la Web Layer y la Business Layer. Aseguran que los módulos trabajen en conjunto correctamente.

3. **System Tests (Pruebas de Sistema):**

   Evalúan el sistema completo, incluyendo todas las capas, desde la base de datos hasta el navegador. Se enfocan en el comportamiento del sistema como un todo.

4. **Acceptance Tests (Pruebas de Aceptación):**

   Validan el sistema desde la perspectiva del usuario, generalmente interactuando con la capa de presentación (navegador). Aseguran que los requisitos de negocio y las expectativas del cliente se cumplan.

Cada nivel se orienta a diferentes aspectos de la funcionalidad, desde la validación interna (unitaria) hasta la validación externa (aceptación)

Deberíamos Automatizar

1. Las pruebas manuales, como las unitarias o de regresión, no son prácticas ni eficientes para tareas repetitivas.

2. La automatización permite obtener retroalimentación rápida, prevenir errores y liberar tiempo para actividades de mayor valor, como pruebas exploratorias.

Elementos clave para la automatización:

1. **Integración continua, builds y despliegues:**
   - Automatizar tareas repetitivas (como builds y despliegues) es esencial.
   - Un proceso de integración continua rápido (<10 minutos) maximiza el ROI y evita bloqueos en el desarrollo.
   - Estrategias para acelerar builds incluyen eliminar dependencias costosas (por ejemplo, bases de datos reales), usar bases en memoria, o distribuir pruebas en varias máquinas.
2. **Pruebas unitarias y de componentes:**
   - Son esenciales para prevenir regresiones y diseñar código robusto, especialmente al usarse junto con TDD.
   - Deben ser la prioridad al comenzar a automatizar.
3. **Pruebas de API y servicios web:**
   - Fáciles de automatizar mediante herramientas como Ruby y pruebas basadas en datos.
   - Permiten validar entradas, salidas y comportamientos sin depender de la interfaz gráfica.
4. **Pruebas detrás de la GUI:**
   - Más estables y fáciles de mantener que las pruebas de interfaz gráfica.
   - Usan herramientas declarativas (como tablas o hojas de cálculo) para escribir pruebas comprensibles para desarrolladores y clientes.
5. **Pruebas de interfaz gráfica (GUI):**
   - Aunque necesarias, deben mantenerse simples para validar elementos como botones o enlaces.
   - Su mantenimiento puede ser costoso si los elementos de la interfaz cambian con frecuencia.
   - Usar buenas prácticas, como asignar identificadores únicos a objetos de la GUI, facilita su automatización.

Otras áreas de automatización:

1. **Pruebas de carga y rendimiento:**
   - Requieren herramientas específicas para simular grandes volúmenes de usuarios o ataques.
2. **Comparaciones de archivos o bases de datos:**
   - Herramientas como diff o macros pueden automatizar la validación de cambios en archivos o datos.
3. **Tareas repetitivas o de configuración de datos:**
   - Automatizar procesos repetitivos (como creación de datos para pruebas o generación de formularios) ahorra tiempo y garantiza consistencia.
4. **Tareas administrativas:**
   - Automatizar cálculos complejos o flujos administrativos también puede aportar valor más allá de las pruebas.

No deberíamos automatizar

1. **Pruebas de usabilidad:**
   - Requieren la participación humana para evaluar la experiencia del usuario.
   - La automatización puede ayudar en la preparación de escenarios, pero no puede captar problemas visuales o de "look and feel" como un humano.
   - El registro de acciones de los usuarios puede complementar las pruebas manuales.
2. **Pruebas exploratorias:**
   - Aunque la automatización puede facilitar la configuración de datos y escenarios, la ejecución y diseño requieren un tester capacitado.

    - Estas pruebas buscan aprender sobre el producto y mejorar su desarrollo futuro, algo que los scripts automatizados no logran.
3. **Pruebas que nunca fallarán:**
    - Si un requisito es tan obvio que el riesgo de defectos es insignificante (como un campo no obligatorio que no cambia), podría no justificar la automatización.
    - Sin embargo, errores como copiar/pegar ocurren, y automatizar estas pruebas simples puede prevenir problemas menores. La decisión debe basarse en el análisis de riesgos.
4. **Pruebas únicas (one-off):**
    - Si una prueba se ejecutará una sola vez y no es tediosa, generalmente es mejor mantenerla manual.
    - Sin embargo, si la tarea es repetitiva o consume mucho tiempo manualmente, podría valer la pena automatizarla, incluso si no se realiza frecuentemente.

Aplicación de principios ágiles a la automatización de pruebas

1. Keep it Simple

- Usa la máxima ágil: “haz lo más simple que pueda funcionar”.
- Diseña pruebas simples, con el alcance mínimo y herramientas sencillas.
- Evita complicarte innecesariamente, priorizando tareas con mayor ROI. Toma pasos pequeños para facilitar ajustes rápidos si algo falla.

2. Retroalimentación iterativa

- Experimenta con enfoques de automatización en iteraciones cortas.
- Evalúa resultados después de cada iteración y ajusta estrategias según sea necesario.
- Evita quedarte atado a herramientas o enfoques inadecuados; la iteración te permite adaptarte.

3. Enfoque de equipo

- La automatización debe ser un esfuerzo colaborativo entre programadores, testers y otros miembros.
- Diseñar código para ser testeable facilita la automatización.
- Especialistas externos pueden ser útiles para pruebas específicas (seguridad, rendimiento), pero el equipo debe tomar la responsabilidad global.

4. Dedica tiempo a hacerlo bien

- Explica a la gerencia que un enfoque apresurado genera deuda técnica, afectando la velocidad y calidad a largo plazo.
- Invierte tiempo en soluciones robustas y sostenibles desde el principio.
- Reserva bloques de tiempo para la automatización y evita la multitarea.

5. Aprende haciendo

- La práctica es clave para dominar herramientas y técnicas de automatización.
- Trabaja en pareja o usa estrategias como explicar problemas en voz alta para encontrar soluciones.
- Acepta los errores como oportunidades de aprendizaje.

6. Aplica prácticas ágiles de codificación a las pruebas

- Trata las pruebas como código: refactoriza, diseña modularmente, usa buenas prácticas y mantén las pruebas simples e independientes.
- Guarda las pruebas en el mismo sistema de control de versiones que el código de producción.
- Escribe pruebas automatizadas siguiendo enfoques como TDD (desarrollo guiado por pruebas).

7. Personaliza según tu equipo y contexto

- Usa la creatividad para encontrar soluciones que se adapten a los desafíos únicos de tu equipo y proyecto.
- Adopta un enfoque disciplinado y colaborativo para maximizar el impacto de la automatización.

Resumen del Libro

- Utilice los cuadrantes de pruebas ágiles para ayudar a identificar dónde necesita la automatización de pruebas y cuándo la necesitará.
- La pirámide de automatización de pruebas puede ayudar a su equipo a realizar las inversiones adecuadas en automatización de pruebas que darán los mayores beneficios.
- Aplique valores, principios y prácticas ágiles para ayudar a su equipo a avanzar en la automatización de pruebas.
- Las tareas repetitivas, los procesos de integración y compilación continuos, las pruebas unitarias, las pruebas funcionales, las pruebas de carga y la creación de datos son buenas candidatas para la automatización.
- Las pruebas del cuadrante 3, como las pruebas de usabilidad y las pruebas exploratorias, pueden beneficiarse de cierta automatización para configurar escenarios de prueba y analizar resultados, pero los instintos humanos, el pensamiento crítico y la observación no se pueden automatizar.
- Un enfoque simple de todo el equipo, utilizando comentarios iterativos y tomando el tiempo suficiente puede ayudarlo a comenzar a trabajar en una buena solución.
- Al desarrollar una estrategia de automatización, comience con el área más problemática, considere un enfoque de múltiples capas y esfuércese por revisar y mejorar continuamente su estrategia en lugar de lograr la perfección desde el principio.
- Considere el riesgo y el retorno de la inversión al decidir qué automatizar.
- Tómese el tiempo para aprender haciendo; aplique prácticas de codificación ágil a las pruebas.
- Decida si puede simplemente generar entradas en memoria o si necesita datos de estilo de producción en una base de datos.
- Suministre datos de prueba que permitan que las pruebas sean independientes, reejecutables y lo más rápidas posible.
- Aborde una necesidad de herramienta a la vez, identifique sus requisitos y decida qué tipo de herramienta elegir o crear que se ajuste a sus necesidades.
- Utilice buenas prácticas de desarrollo para la automatización de pruebas y tómese el tiempo para un buen diseño de pruebas.
- Las herramientas automatizadas deben adaptarse a la infraestructura de desarrollo del equipo.
- Pruebas automatizadas de control de versiones junto con el código de producción que verifican.
- Una buena gestión de pruebas garantiza que las pruebas puedan proporcionar una documentación eficaz del sistema y del progreso del desarrollo.

TDD

“El acto de diseñar tests es uno de los mecanismos conocidos más efectivos para prevenir errores…El proceso mental que debe desarrollarse para crear tests útiles puede descubrir y eliminar problemas en todas las etapas del desarrollo”

Diferencia entre Test-First Development y Test-Driven Development:

1. **Test-First Development (TFD):**
    - **Definición:** Los tests se escriben antes del código de producción, pero no necesariamente se desarrolla el código siguiendo un enfoque iterativo "test por test".
    - **Características:**
        - No implica un diseño emergente; el diseño puede estar predefinido (por ejemplo, en una pizarra).
        - Se aplica tanto a nivel de pruebas unitarias como de pruebas de cliente.

        - Útil en proyectos grandes donde el diseño suele planificarse previamente mediante discusiones arquitectónicas.
2. **Test-Driven Development (TDD):**
    - **Definición:** Enfatiza un desarrollo iterativo y incremental donde el código de producción se crea "test por test", siguiendo un diseño emergente.
    - **Características:**
        - El diseño surge a medida que se escriben y pasan las pruebas.
        - Promueve un enfoque más adaptativo y exploratorio.
        - Más común en proyectos pequeños o ágiles, donde las decisiones de diseño evolucionan con el desarrollo.

Diferencia clave:

En TDD, el diseño se desarrolla mientras se escribe el código a partir de las pruebas, mientras que en TFD, las pruebas pueden guiar el desarrollo, pero el diseño puede estar predefinido antes de escribir las pruebas o el código.

Es una técnica avanzada que combina dos prácticas clave: **Escribir las pruebas primero (Test First Development)** y la **Refactorización (Refactoring)**. Esta metodología se basa en un ciclo iterativo y continuo para garantizar que el código sea funcional, limpio y mantenible.

Para escribir las pruebas, generalmente se utilizan **pruebas unitarias**, que validan el funcionamiento de pequeñas unidades de código en aislamiento. TDD fomenta un enfoque basado en diseño emergente, donde el desarrollo del código sigue el flujo definido por las pruebas.

3 reglas de TDD

1. **No escribir código de producción sin antes tener un test unitario fallando.**
2. **Escribir solo lo suficiente de un test para que falle (incluso errores de compilación cuentan).**
3. **Escribir solo el código de producción necesario para pasar el test fallido.**

El proceso consiste en escribir un test mínimo, hacerlo fallar, y luego escribir el código necesario para que pase, manteniendo el sistema siempre ejecutable. Esto minimiza errores, reduce el tiempo de depuración y fomenta diseños desacoplados y testeables.

Los beneficios incluyen:

- Mayor confianza para hacer cambios gracias a los tests.
- Código más limpio y mantenible.
- Tests como documentación práctica y actualizada.
- Diseño mejorado y más modular.

Aunque inicialmente puede parecer lento o restrictivo, TDD transforma el flujo de trabajo, haciendo que el desarrollo sea más eficiente y menos propenso a errores.

Ciclo del TDD

1. **Nano-ciclo (segundos): Las tres leyes de TDD**
    - Escribir un test fallido antes de código de producción.
    - No escribir más test del necesario para fallar.
    - Escribir solo el código de producción necesario para pasar el test. Este ciclo hipergranular asegura que cada paso avance hacia un sistema funcional constantemente.
2. **Micro-ciclo (minutos): Red-Green-Refactor**
    - **Red**: Crear un test que falle.
    - **Green**: Escribir código para pasarlo.

    - **Refactor**: Limpiar y mejorar el código. La idea es dividir objetivos en dos fases: funcionalidad correcta y estructura correcta.
3. **Milli-ciclo (decenas de minutos): Specific/Generic**
    - A medida que los tests se vuelven más específicos, el código debe hacerse más general.
    - Evitar optimizaciones locales que lleven a un "atasco", retrocediendo y añadiendo generalidad al código.
4. **Ciclo primario (horas): Límites arquitectónicos**
    - Cada hora, revisar la arquitectura general para garantizar que las decisiones de diseño mantengan una estructura limpia.
    - Ajustar actividades según los límites definidos para no perder el enfoque arquitectónico.

Este enfoque secuencial y jerárquico combina la granularidad de los detalles con una visión global, promoviendo diseños robustos, flexibles y mantenibles.

Beneficios del TDD:

- Reduce defectos en el código desde las primeras etapas del desarrollo.
- Fomenta un diseño más limpio, claro y modular.
- Facilita el mantenimiento y refactorización del código sin temor a romper funcionalidades existentes.
- Proporciona una suite de pruebas automatizadas que sirven como documentación viva del sistema.

TDD no solo es una técnica de desarrollo, sino también una práctica que ayuda a equipos ágiles a mantener alta calidad en sus entregables.