Resumen Testing de Software (Prueba de software)

Testing: Proceso destructivo de tratar de encontrar defectos en el código, se debe tener una actitud negativa. Se trata de verificar que el software se ajusta a los requerimientos y además validar que las funciones se implementan correctamente. Es exitoso aquel que encuentra muchos defectos, por lo que un desarrollo exitoso lleva a un test fallido (sin defectos). Expectativa al hacer testing: validación (si es el producto correcto o no) y verificación (si el producto funciona correctamente).

→ Un programador no debería testear su propio código, ni una unidad de programación de sus propios proyectos. Esto ocurre porque uno está muy embebido en una determinada situación por lo que le cuesta hacer una separación y lograr objetividad necesaria para poder encontrar defectos.

Testing no es depurar código, no es verificar que se implementen las funciones, no es demostrar que hay errores y corregirlos.

Testing no es control de calidad, porque el aseguramiento de la calidad va a todo el proceso de software.
Hay otras actividades además del testing que ayudan con la calidad buscando encontrar defectos:

- *Revisiones técnicas:* es una forma de asegurar calidad en el producto, la cual si se puede hacer a lo largo de todo el ciclo de vida del producto. A diferencia del testing que solo se realiza sobre el código, estas se pueden realizar sobre requerimientos, diseño, arquitectura, etc.

Estas actividades tienen la intención de prevenir defectos, detección temprana. Busca encontrar los errores antes de que estos se conviertan en defectos.
Si el aseguramiento de calidad (QA) se realiza correctamente tenemos un producto que llega en mejores condiciones al testing, entonces es muy probable que se encuentren menos defectos, inclusive la calidad del producto será mejor si se realizaron estas actividades.

Error: Algo que está mal hecho pero lo encuentro y corrijo en la etapa en que se produce.

Defecto: Es encontrar un error en una etapa posterior a la cual se introdujo el mismo . Es más grave que un error y más costoso de corregir. Los errores en la etapa de requerimientos son los peores, más graves, caros y difíciles de resolver. Entonces, mientras más tarde sean identificados los defectos más costoso es ya que son arrastrados. Se asume que el código siempre tiene defectos. Es un defecto si lo puedo reproducir, siguiendo una serie de pasos llego nuevamente. Si solo apareció una vez, no lo busco más. Se clasifica en Severidad y Prioridad.

- **Severidad:** qué tan grave es un defecto, qué tanto esfuerzo voy a necesitar hacer para corregirlo, cuántos problemas me traerá si no resuelvo el defecto hoy. Evalúa cuánto daño le hace al software ese defecto. La define el tester. Clasificación:
  1. Bloqueante: No me permite usar el software.
  2. Crítico: Defecto importante en una funcionalidad importante. Ej. Generación de reporte importante.
  3. Mayor:
  4. Menor:
  5. Cosmético:

- **Prioridad:** qué tan rápido lo vamos a solucionar. No siempre tiene relación 1 a 1 con la severidad. Evalúa qué tan importante es para el negocio la resolución del defecto, por lo cual la define el cliente. Clasificación:
  1. Urgencia:
  2. Alta:
  3. Media:
  4. Baja:

Niveles de prueba: cada uno tiene asociado una prueba.

• ***Pruebas Unitarias:*** Destinadas a encontrar errores en el código, sólo prueban una funcionalidad claramente definida. Funcionan a nivel de código. Las realiza en la etapa de Desarrollo el Desarrollador y el mismo los corrige. Hoy en día generalmente están automatizadas. Son códigos que prueban código.
Es la primera etapa de la prueba, está enfocada a los componentes más pequeños del Software que se puedan probar (programas y módulos), se verifican tipos de datos inconsistentes, precisión en las funciones de cálculos, comparación entre tipos de datos, terminación de loops, correspondencias entre parámetros y argumentos, entre otros.
Relacionando con el PUD, se realizan en el WF de Implementación.

• ***Pruebas de Integración:*** Se juntan funcionalidades y se ve si funcionan bien en conjunto. Es ir integrando de a poco para saber dónde se produce el error. Realizadas entre desarrollador y tester. Se busca encontrar defectos en la unión entre diferentes componentes. Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto. También pueden ser automatizadas (con la práctica de Integración Continua), de esta forma obtenemos permanentemente un buil que está en condiciones de ser la entrada al nivel de prueba del sistema. Relacionando con el PUD, se realizan en un determinado WF según como cada equipo de cada proyecto funcione.

• ***Pruebas de Sistema:*** No probamos a nivel de código sino a nivel de sistema. Se prueba una funcionalidad en su totalidad la cual ya está implementada. Es como un test de unidad, de una unidad formada por otras unidades ya integradas. Realizadas entre desarrollador y tester. Especialmente en este nivel se debe respetar que un desarrollador no pueda probar su propio proyecto. Relacionando con el PUD, se realizan en el WF de Prueba.

o *Prueba de verificación de versión:* al incorporar ciclo de vida iterativo e incremental, a las pruebas de sistemas se las suele llamar pruebas de versión, porque si estamos trabajando con este ciclo de vida cada vez que generamos una versión no es el sistema completo pero es a esa versión a la cual le vamos a hacer el nivel de prueba de sistema y luego la aceptaremos. Su objetivo es tener una rápida visión de la estabilidad de la aplicación, antes de realizar una prueba en profundidad.

• ***Pruebas de Aceptación de Usuario:*** Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a las necesidades, es decir, si el software hace lo que tenía que hacer lo cual es lo que el usuario quería. En estas pruebas no se deberían encontrar defectos. Tienen que ser siempre lo más similar al entorno productivo. Realizadas por el usuario, el cliente (la persona que quiere cumplir sus requerimientos) y en Metodología Ágil las realiza el Product Owner.
Incluye tanto pruebas alfa (el usuario en el laboratorio) como beta (en el ambiente de trabajo, para que las utilicen los usuarios finales). Relacionando con el PUD, se realizan en el WF de Despliegue.

Ambientes para construcción del Software: como mínimo se deben tener Desarrollo, Prueba y Producción.

• ***Desarrollo:*** Computadora / Servidor en nube (en caso de remoto).
• ***Prueba:*** Acá se realizan las pruebas de sistema, entonces son preparados con datos de prueba y todas las herramientas necesarias para realizar las pruebas. Servidor para que los testers puedan hacer las pruebas.
• ***Pre-Producción:*** Entorno lo más parecido posible a las condiciones en que se usará el software. Donde se deberían hacer las pruebas de aceptación. Deberían porque a veces no son posibles ya que son muy costosas.
• ***Producción:*** Ámbito en el que el software se ejecuta. (Personas / clientes utilizándolo).

Debemos tener un ambiente de Prueba separado del de Desarrollo porque se debe tener un ambiente limpio para hacer el testing en el cual los desarrolladores no puedan intervenir ya que según una estadística, por cada error que se corrige se introducen tres. Mientras más nos acercamos con errores/defectos al ambiente de producción, más caro es (costos tangibles como dinero y costos intangibles como enojo de los clientes).

En las empresas que solo producen software (Globant, Dycsis, etc.) tienen solo ambiente de Desarrollo y de Prueba, luego lo instalarán y probarán en el cliente.

Casos de Prueba: Es el artefacto por excelencia del testing que va a contener un conjunto de condiciones/pasos que se deben ejecutar para poder garantizar que el resultado esperado sea o no igual al resultado obtenido y de esta forma encontrar defectos. Se busca diseñar la menor cantidad de casos de prueba posibles y encontrar la mayor cantidad de defectos posibles.

Es un escenario específico con datos específicos que intenta probar un conjunto de situaciones o criterios de aceptación o condiciones de prueba. Entonces para poder ejecutarlo debemos tener bien especificado que queremos probar, la condición de seteo del sistema (en que estado debe estar el sistema en ese momento), el set de datos que se utilizará y el resultado esperado.

Los lugares ideales para extraer información para diseñar un caso de prueba para poder hacer validación y verificación son: Requerimientos (casos de uso - user stories). Si extraigo la información del código solo estaré realizando verificación (ver si el producto hace lo que tiene que hacer). Derivación de casos de prueba: Documentos del cliente, Información de relevamiento, Requerimientos, Especificaciones de programación, Código.

Estrategia de Caja Blanca: Podemos ver el detalle de la implementación de la funcionalidad, vamos a disponer del código y vamos a poder diseñar nuestro caso de prueba para garantizar cobertura (si quiero cubrir todos los if, los else o todas las ramas de ejecución de cierta funcionalidad). Como yo puedo ver ese detalle de implementación puedo guiar la ejecución de los casos de prueba. Estático, no ejecuta el código.

Estrategia de Caja Negra: No conozco la estructura interna de la implementación y solamente lo voy a analizar en términos de entradas y salidas, es decir, voy a identificar cuales son las diferentes entradas que una funcionalidad puede tener, coy a elegir los valores con los cuales voy a ejecutar esa funcionalidad, y finalmente voy a comparar esa salida obtenida contra las que esperaba obtener. Dinámico, para hacer las pruebas debo ejecutar el software.

Cobertura: Porcentaje de pruebas que se hacen sobre el total de pruebas que deberían hacerse para cubrir la totalidad de las condiciones de prueba.

Métodos: Es como la implementación de una estrategia en particular. Solo sirven para diseñar casos de prueba, con la premisa de que queremos la menor cantidad de casos de prueba que sean lo más eficientes y efectivos posible.
No se puede probar todo ya que existen infinitas combinaciones de valor para probar funcionalidades. Por esto surge la necesidad de buscar algún criterio económico para el testing: Se busca maximizar la cantidad de defectos encontrados minimizando el esfuerzo requerido para hacerlo. Métodos dentro de las estrategias:

Caja Negra:

• **Partición de equivalencias:** 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. No igual, equivalente, por ejemplo si la funcionalidad es calcular un porcentaje será equivalente porque calculó el porcentaje pero no igual porque el número será diferente.
Lo que se hace es analizar primero cuáles son las condiciones externas que van a estar involucradas en el desarrollo de una funcionalidad.
Las condiciones externas van a ser las entradas y las salidas. Entradas: cualquier tipo de variable que pueda estar en juego (campo de texto, combo de selección, coordenadas de posición, etc.), es decir, cualquier condición que va a guiar la ejecución de la funcionalidad. Salidas: Mostrar mensaje en pantalla, teniendo en cuenta cual es la interfaz. Por ejemplo, el mensaje puede ser una luz en un control remoto.
Una vez identificadas las condiciones externas para cada una de ellas voy a analizar cuales son los subconjuntos de valores posibles que pueden tomar cada una de ellas que producen un resultado equivalente.

Las clases de equivalencia me sirven para que cuando diseñe los casos de prueba y elija valores para ingresar a las diferentes condiciones externas de entrada, voy a tomar solo una por clase de equivalencia, porque cualquiera que yo tome va a producir un resultado equivalente.

• **Análisis de valores límites:** Variante del método anterior, pero en vez de seleccionar cualquier elemento como representativo de una clase de equivalencia, se seleccionan los bordes de una clase. El anterior método y éste se basan en las especificaciones.

• **Adivinanza de defectos:** Técnica basada en la experiencia para identificar pruebas que probablemente expondrán defectos. Se lista defectos posibles o situaciones propensas a error, desarrollo de pruebas basadas en la lista.

• **Testing Exploratorio:** Técnica basada en la experiencia que generalmente se usa para iniciar un proceso de testing cuando uno no conoce el producto, entonces hace una pasada para ver de qué se trata el producto, más o menos con que me puedo encontrar, en qué situación está, para tener un poco de información para hacer después una derivación de cobertura un poco más sistemática.

Caja Blanca: Utiliza la estructura de control del diseño procedural para derivar casos de prueba que:

• Garanticen que todos los caminos independientes dentro de un módulo han sido ejercitados por lo menos una vez.
• Ejerciten todas las decisiones lógicas en sus lados verdaderos y falsos.
• Ejerciten sus estructuras de datos internas para asegurar su validez.

El testing de caja blanca se utiliza porque hay errores que pueden no ser detectados por testing de caja negra, como errores tipográficos, caminos lógicos que se cree que no se ejecutarán, etc.

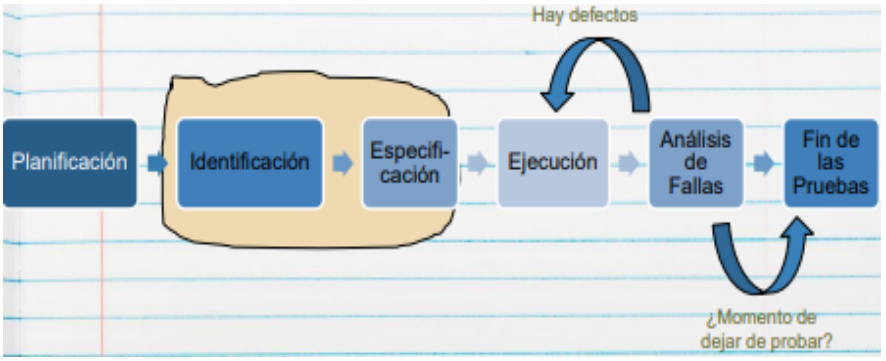
• Cobertura de secuencias: Recorrer cada uno de los posibles caminos lógicos
• Cobertura de decisión: Ejecutar cada decisión por lo menos una vez obteniendo un resultado verdadero y uno falso.
• Cobertura de condición: Cada condición en una decisión tenga todos los resultados posibles al menos una vez, ejercita las condiciones lógicas contenidas en un módulo de programa.
• Cobertura de loop: Se focaliza en la validez de las instrucciones de loops, se prueba saltarse completamente el bucle, sólo una iteración, dos, m iteraciones donde m < n, n – 1, n, n + 1 iteraciones.
• Cobertura de caminos básicos: ejecutar por lo menos una vez cada instrucción del programa y cada decisión se habrá ejecutado en su lado verdadero y falso.

Ciclos de Prueba/Test: Abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una misma versión del sistema a probar. Termina cuando se ejecutaron todos los casos de prueba o cuando existe un defecto invalidante que impide que ese ciclo de prueba continúe. Defecto invalidante/bloqueante: no funciona el logueo. Lo ideal sería tener 2 ciclos de prueba, se ejecuta el primer (ciclo cero), se corrigen errores, se ejecuta el segundo (ciclo uno) y se esperaría tener la salud del producto para ponerlo en producción.

Pruebas de Regresión: Al concluir un ciclo de pruebas (la ejecución de la totalidad de los casos de prueba), y reemplazarse la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión, a fin de prevenir la introducción de nuevos defectos al intentar solucionar los detectados.
“La única manera de estar seguro de que no existe un error es resetear todo”

Técnicas de Prueba
• *Sin testeo de regresión:* Esta se utiliza ya que es menos costosa pero se corre el riesgo de errores ocultos, porque no sabemos si el programador ha introducido errores nuevos al tratar de corregir lo que les reportamos.
• *Con testeo de regresión:* Se corren todos los test cases en cada ciclo de prueba, lo que permite detectar los nuevos problemas. Todos los ciclos de prueba se van a ejecutar como si fueran un ciclo cero, es decir que se deben probar todos los casos de prueba como si fuera la primera vez que estoy probando. Es mejor con regresión.

Proceso de Pruebas: Se planifican las pruebas, la parte pintada de amarillo es el Diseño de Pruebas (se diseñan los casos de prueba). Luego se toma la versión del producto que me dan, se corren los casos de prueba y se obtiene como resultado lo que suele llamarse Reporte de Defectos. Este reporte se analiza y se vuelve a desarrollo para que corrijan.
Se repite hasta llegar al criterio de corte.



• **Planificación:** Determina cómo se incluirá el Testing en el plan del proyecto, y cómo será el Test Plan (que recursos se usará, que riesgos se tendrá, cuál será el criterio de aceptación, qué entornos se emularon, quién realizará cada Testing, cuando, etc.). El resultado de la planificación es el Plan de Pruebas, que debe contener:
  o Riesgos y objetivos del Testing.
  o Estrategia de Testing.
  o Recursos.
  o Criterio de Aceptación.

• **Diseño (Identificación y especificación de casos de prueba):** Revisando las bases de la planificación del Testing, se identifican los datos necesarios, diseñan y priorizan los CP que se llevarán a cabo (se define qué entorno se usará, cómo se llevarán a cabo, por quién, cuándo, etc.). Analiza si los requerimientos son testeables o no. Se define si se usa regresión o no.

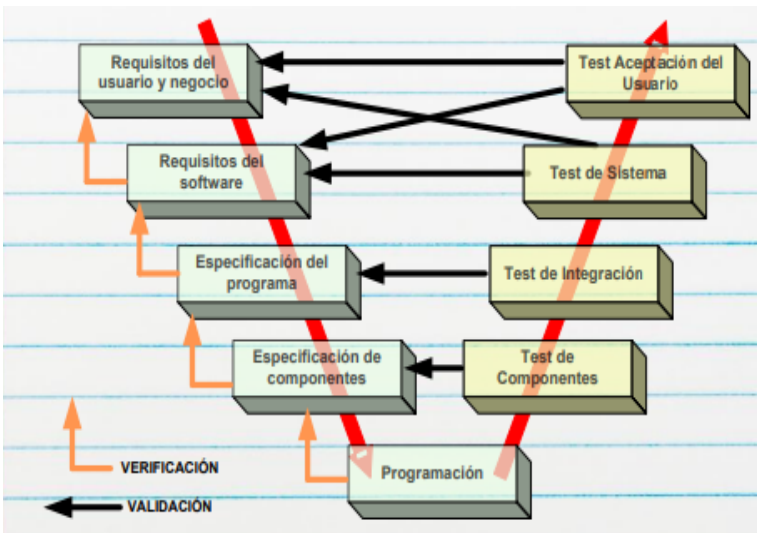
• **Ejecución:** Cuando se ejecutan los CP, se registran y se comparan los resultados generando un reporte de defectos (con defectos encontrados, condiciones, entornos). Se trata de automatizar lo más que se pueda respecto de estas ejecuciones (ahorrando tiempo/costo). Incluye: Creación de los datos necesarios para la prueba, automatización de todo lo que sea necesario, implementar y verificar el ambiente, ejecutar los casos de prueba, registrar los resultados de la ejecución y comparar los resultados reales con los esperados.

• **Análisis de fallas (Evaluación y Reporte):** Se hace un seguimiento de la corrección de los defectos encontrados hasta que se cierren todos los CP, es decir que se hayan solucionado todos. Se evalúa el criterio de aceptación, se reporta el resultado de las pruebas a los interesados, se verifica los entregables y que los defectos se hayan corregido y se evalúa cómo resultaron las actividades de testing y se analizan las lecciones aprendidas. Acá se confecciona el informe de reportes.

• **Fin de las pruebas:** En las empresas más maduras se deja de probar recién cuando no hay defectos bloqueantes, críticos, mayores y los defectos menores y cosméticos son muy pocos, los que se ponen en la nota de Release. Se confecciona el informe final (opcional).

Modelo en V: Este modelo se utiliza para poder determinar cuándo se está en condiciones de realizar determinadas actividades, en función de la etapa de desarrollo en la que se encuentre el software. Acá se puede notar cómo las pruebas se realizan en orden de granularidad inverso al proceso de desarrollo del software. Es decir, se desarrolla desde componentes de granularidad gruesa, como son los requerimientos abstraídos de detalles de implementación, hacia componentes de menor granularidad, como son clases o módulos, dependiendo del paradigma. En cambio, para las pruebas, la granularidad se da en sentido inverso.

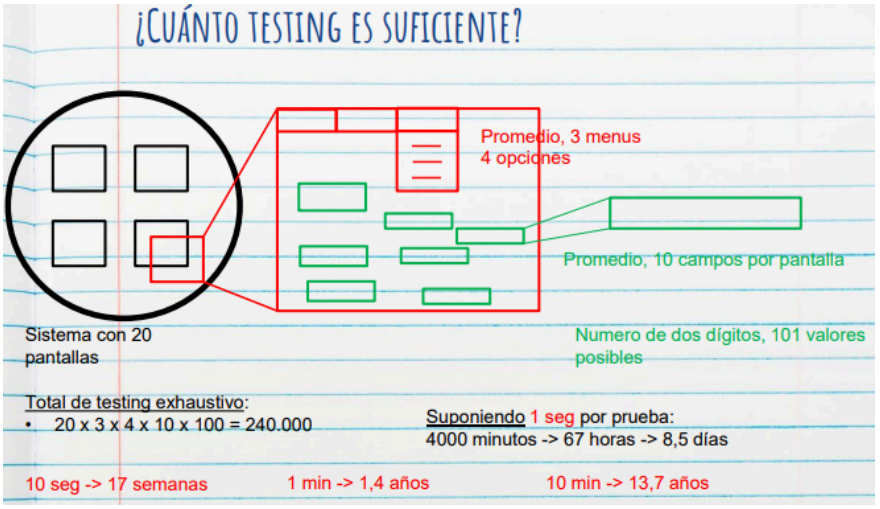
● Testing de Componentes → Testing unitario.
● Especificación del programa → diseño/arquitectura.



¿Cuánto Testing es suficiente? El testing exhaustivo es imposible por la cantidad de tiempo que requiere. El momento en que se deja de hacer testing depende del nivel de riesgo o costo asociado al proyecto. Los riesgos permiten definir prioridades de qué se debe testear primero y con qué esfuerzo. El criterio de aceptación se utiliza normalmente para decidir si una determinada fase de testing ha sido completada. Este puede ser definido en términos de:

- Costos.
- % de tests corridos sin fallas.
- Inexistencia de defectos de una determinada severidad.
- Pasa exitosamente el conjunto de pruebas diseñado y la cobertura estructural.
- Good Enough: Cierta cantidad de fallas no críticas es aceptable.
- Defectos detectados es similar a la cantidad de defectos estimados.

El criterio de aceptación sirve para definir y negociar en ágil con el Product Owner y en tradicional con el Líder del Proyecto a cuántos defectos son aceptables para terminar.



Principios del Testing

1. Una parte necesaria de un caso de prueba es definir el resultado esperado. Si el resultado esperado de un caso de prueba no ha sido predefinido, lo más probable es que un resultado erróneo se interpretará como un resultado correcto, debido a el fenómeno de "el ojo viendo lo que quiere ver", a pesar de la definición destructiva adecuada de prueba, hay todavía un deseo subconsciente de ver el resultado correcto.

2. Un programador debe evitar testear su propio programa. Los propios desarrolladores “no quieren” encontrar sus propios defectos, por lo que el carácter de pruebas destructivas se deja de lado. Además, puede que el programa contenga errores por malentendidos del programador sobre el dominio, y si él mismo testea, no se van a detectar estos defectos.

***3.** Una empresa de desarrollo no debe testear sus propios programas.* Misma explicación que el principio 2 orientado a empresas, agregando que si las mismas empresas testean sus programas, es posible que destinen menos recursos y eviten encontrar defectos para cumplir con el calendario y los costos establecidos.

***4.** Cualquier proceso de prueba debe incluir una inspección minuciosa de los resultados de cada prueba.* Se deben realizar inspecciones minuciosas para detectar la totalidad de los defectos encontrados tras la ejecución de una prueba, ya que pueden surgir más de un defecto (y esto es lo que se busca) por cada caso de prueba.

***5.** Los casos de prueba deben escribirse para condiciones de entrada que no son válidas e inesperadas, así como para las que son válidas y esperadas.* Muchos errores que se descubren repentinamente en el desarrollo de software aparecen cuando se usa de alguna manera nueva o inesperada, y no responde cómo debería (catcheando el error)

***6.** Examinar un programa para ver si no hace lo que se supone que debe hacer es sólo la mitad de la batalla; la otra mitad es ver si el programa hace lo que no se supone que debe hacer.* Los programas deben ser examinados para detectar defectos secundarios no deseados.

***7.** Evite los casos de prueba desechables, a menos que el programa sea realmente un programa de descarte.* Los casos de pruebas utilizados en el testing deben ser reproducibles, es decir, no se deben realizar casos de prueba sobre la marcha (ad-hoc) ya que es imposible reportar un defecto sin tener las condiciones y los pasos en los cuáles el defecto surgió.

Además, realizar testing es muy costoso, por lo cuales los casos de prueba deben ser reutilizados para volver a testear escenarios luego de la corrección de errores.

8. *No planee un esfuerzo de prueba bajo la suposición tácita de que no se encontrarán errores.* Es un error pensar de esta manera al momento de realizar software. Se debe tener en claro la definición de testing, en la cual se define que el objetivo de esta actividad es encontrar errores, y se presume de antemano su existencia.

9. *La probabilidad de que existan más errores en una parte de un programa es proporcional al número de errores ya encontrados en esa parte.* El concepto es útil porque nos da una idea de en qué sección del programa hacer foco o asignar más recursos, si una sección particular de un programa parece ser mucho más propenso a errores que otras secciones, es recomendable realizar pruebas adicionales y es probable que encontremos más errores.

10. *Las pruebas son extremadamente creativas e intelectualmente desafiantes.* Aunque existen métodos y estrategias para abarcar un mejor nivel de cobertura testing en los casos de pruebas, siempre es necesario un poco de creatividad del diseñador de estos.

Mitos del Testing

- "El testing es el proceso para demostrar que los errores no están presentes".
- "El propósito del testing es demostrar que un programa realiza sus funciones previstas de forma correcta".
- "El testing es el proceso que demuestra que un programa hace lo que se supone que debe hacer''.

Estas definiciones o afirmaciones sobre el testing son incorrectas, ya que el objetivo de testear un programa es agregar valor al producto revelando su calidad y brindando confianza en el software, de forma más concreta, encontrar y remover defectos en el código de este. Entonces, no se prueba un sistema para mostrar que funciona, sino que se comienza con la suposición de que el software contiene defectos, y se realiza el testing para encontrar la mayor cantidad de ellos posible.

Por otro lado, un programa puede hacer lo que se supone que debe hacer, y aun así, contener defectos. Es decir, un error está claramente presente si un programa no hace lo que se supone que debe hacer, pero los errores también están presentes si un programa hace lo que no se supone que debe hacer.

Tipos de prueba

• ***Smoke Test:*** Es un tipo de prueba que se hace para validar que no haya fallas groseras, de gran magnitud, en el producto de software. Se realiza antes de comenzar el ciclo 0. Nos ahorra empezar a testear formalmente si encuentra un error, porque todavía hay algo que corregir. Consiste en una corrida rápida para ver si el producto está en condiciones de pasar al ciclo de prueba.

• ***Testing Funcional:*** Controla que el software se comporte de la misma manera que lo especificado en la documentación, cumpliendo con las funcionalidades y características definidas. Se basa en los requerimientos funcionales y el proceso de negocio. Hay testing funcional basado en dos aspectos:

  o *Basado en requerimientos:* cuando se prueban requerimientos específicos, apunta a probar una funcionalidad sola (utilizan a los requisitos definidos en una ERS o los acuerdos que contienen las pruebas de usuario y los criterios de aceptación de una US para realizar las pruebas.)
  o *Basado en proceso de negocio:* cuando se prueba un proceso de negocio completo, es decir, se prueba todo el proceso. Por ejemplo, en una venta se prueba la búsqueda del artículo, la selección y facturación del mismo.

• ***Testing No Funcional:*** Se basa en cómo trabaja el sistema haciendo foco en los requerimientos no funcionales (foco en el "cómo", no en el "que"). Son las pruebas más complejas, por su gran dependencia al entorno del sistema, por lo que debe ser lo más parecido posible al del cliente. Sin embargo, se tienen características que no pueden probarse, como el

ancho de banda del internet, la seguridad o performance en una determinada situación. Y se pueden solucionar con las pruebas de aceptación. Incluye varios tipos de prueba como:

- *Performance:* Se ve el tiempo de respuesta (escenario esperado respecto a los tiempos de respuesta) y la concurrencia. Deben pasar esta prueba sí o sí.
- *Carga:* no solo mira performance, mira el comportamiento de los dispositivos de hardware (procesadores, discos, etc.) y de las comunicaciones.
- *Stress:* queremos forzar al sistema para que falle, se lo somete a condiciones más allá de las normales. Se ve el tiempo que hay que esperar para probar nuevamente el sistema y la robustez del sistema (tiempo de recuperación).
- *Mantenimiento:* para ver si el producto está en condiciones de evolucionar, se controla que haya documentación, manual de configuración, etc. Se observa la facilidad que existe para corregir un defecto.
- *Usabilidad:* que sea cómodo para el usuario.
- *Portabilidad:* se prueba en los distintos entornos acordados con el cliente.
- *Fiabilidad:* probamos que podemos depender del sistema. Resultados que se obtienen, seguridad física del software.
- *De interfaz de usuario:* suelen ser más complejas las GUI's que las interfaces de comandos.
- *De configuración.*

Test Driven Development – TDD: Este es un tipo de proceso en el que nos enfocamos en construir primero la prueba unitaria cuando tengo los requerimientos y luego codear el componente. Si pienso en las pruebas después tengo menos errores, el fundamento filosófico de esto es que si no tengo claro que tengo que hacer no puedo crear pruebas para eso. Es una técnica de desarrollo del software que involucra dos prácticas:

1. Test first development: en esta técnica primero se escriben las pruebas unitarias referentes a la característica de producto a implementar. Definidas las pruebas unitarias, se piensa en la codificación necesaria para que las pruebas se ejecuten con éxito. Dado que las pruebas unitarias prueban unidades concretas de código, esta técnica obliga al desarrollador a modularizar su codificación haciendo que los métodos o clases a probar tengan una única responsabilidad. Además de pruebas unitarias, pueden incluirse en primera instancia pruebas de integración.

2. Refactoring: esta técnica consiste en reestructurar el código existente sin modificar el comportamiento que el mismo provee. Implica la mejora de aspectos no funcionales del software, cuyo objetivo es mejorar la claridad del código y reducir su complejidad, con el fin de hacerlo más mantenible.