

Fundamentos de las Pruebas

¿Por qué son necesarias las pruebas?

El software está en todo lugar. Está presente en todos los aspectos y momentos de la vida de las personas. Algunas veces el software es obvio, otras veces no.

¿Qué son las pruebas?

Es una manera de explorar, de revisar el producto que tú quieras desarrollar, experimentarlo, verlo, entenderlo. Entre menos entiendas el producto que estas desarrollando más errores va a cometer.

Razones para hacer pruebas

- Estamos viendo el problema y estamos en mejores continuas
- Los costos se están volviendo muy altos
- Estándares o implicaciones legales

Testing, Objetivos, Importancia

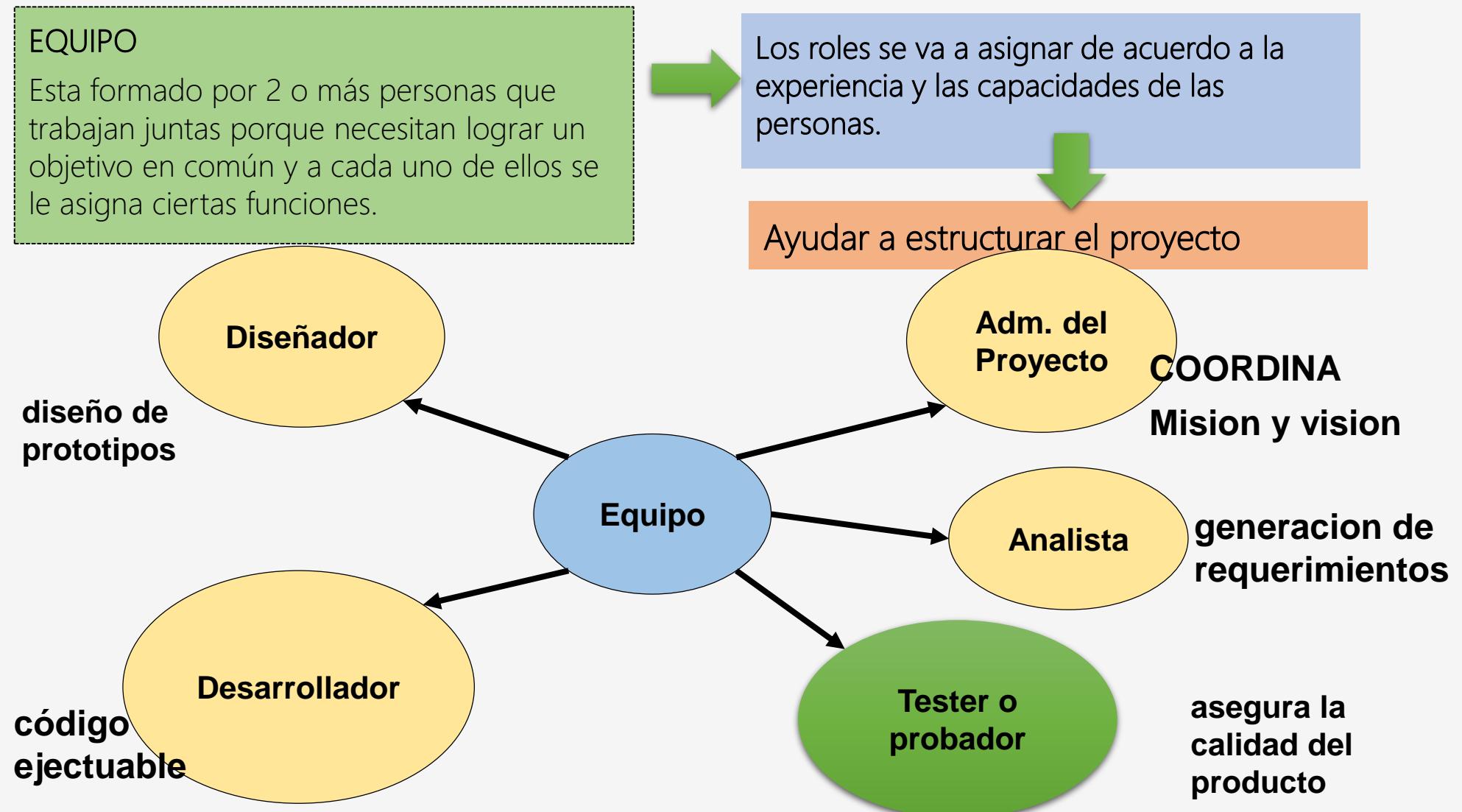
Error, Defecto, Fallo

Proceso Fundamental

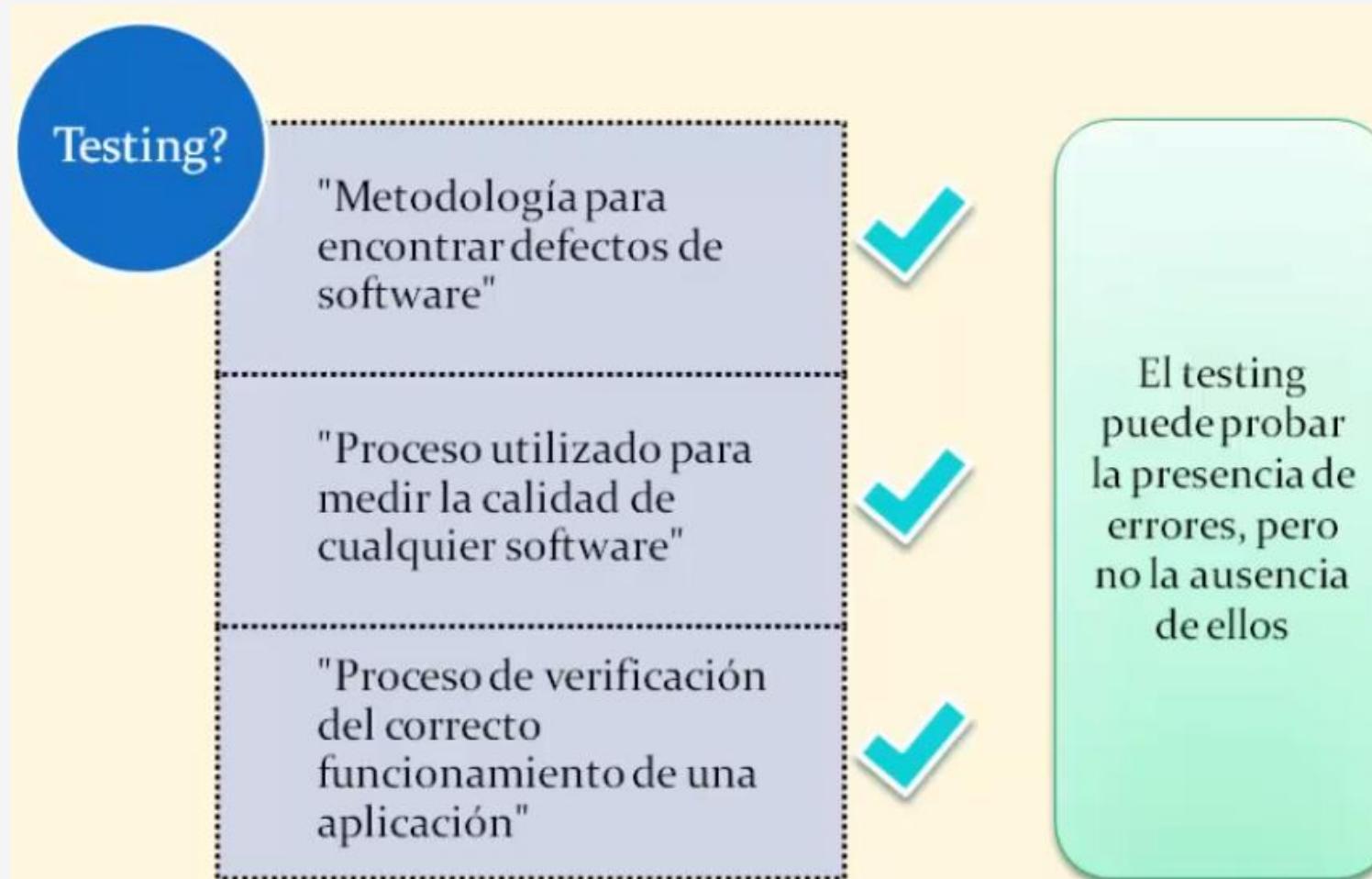
Cuándo finalizamos las pruebas?

Cómo realizar un caso de prueba?

Conceptos básicos – Desarrollo de Software



Conceptos de Testing (pruebas de software)



Las pruebas de software o testing es el proceso que permite verificar la calidad de un producto de software. Permite identificar posibles fallos en la implementación, calidad o usabilidad de un programa.

Conceptos de Testing (pruebas de software)

ERROR: Acción humana que produce un resultado incorrecto

DEFECTO: Desperfecto en un componente que puede causar que el mismo falle en su funcionamiento

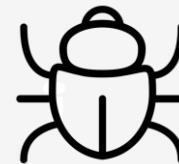
FALLO: Manifestación física de un defecto



un error
humano



puede generar



defectos



va a causar

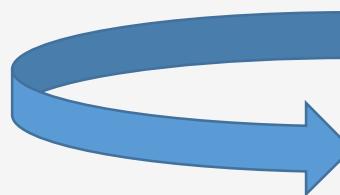


? !

Un
fallo

Causas de Fallos

Error humano, porque el defecto se introdujo en el código de software, en los datos o en los parámetros de configuración.



- plazos cortos para entregar las cosas
- tenemos mucha complejidad
- desarrollador o analista se distrajeron.

Condiciones ambientales, los cambios en las condiciones ambientales como ser: radiación, magnetismo, fallo en el disco duro, fluctuación en el suministro eléctrico, etc.

Conceptos de requisitos y calidad

Requisito

- Atributo funcional deseado o considerado obligatorio

Calidad

- Grado en el cual un componente, sistema o proceso satisface requisitos especificados y/o necesidades y expectativas del cliente

Calidad del Software: la suma de todos los atributos que se refieren a la capacidad del software de satisfacer los requerimientos dados.

Atributos de calidad

Funcionales

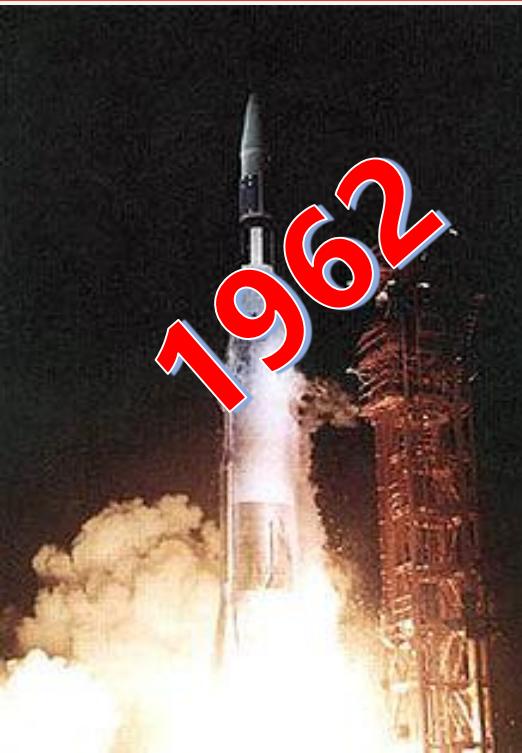
- Correctitud
- Completitud

No Funcionales

- Fiabilidad
- Usabilidad
- Portabilidad
- Eficiencia
- Mantenibilidad

No significa que mi sistema deba tener todos los atributos funcionales y no funcionales (mencionados), dependerá del sistema para priorizar los atributos que necesitamos.

Porque son necesarias las pruebas

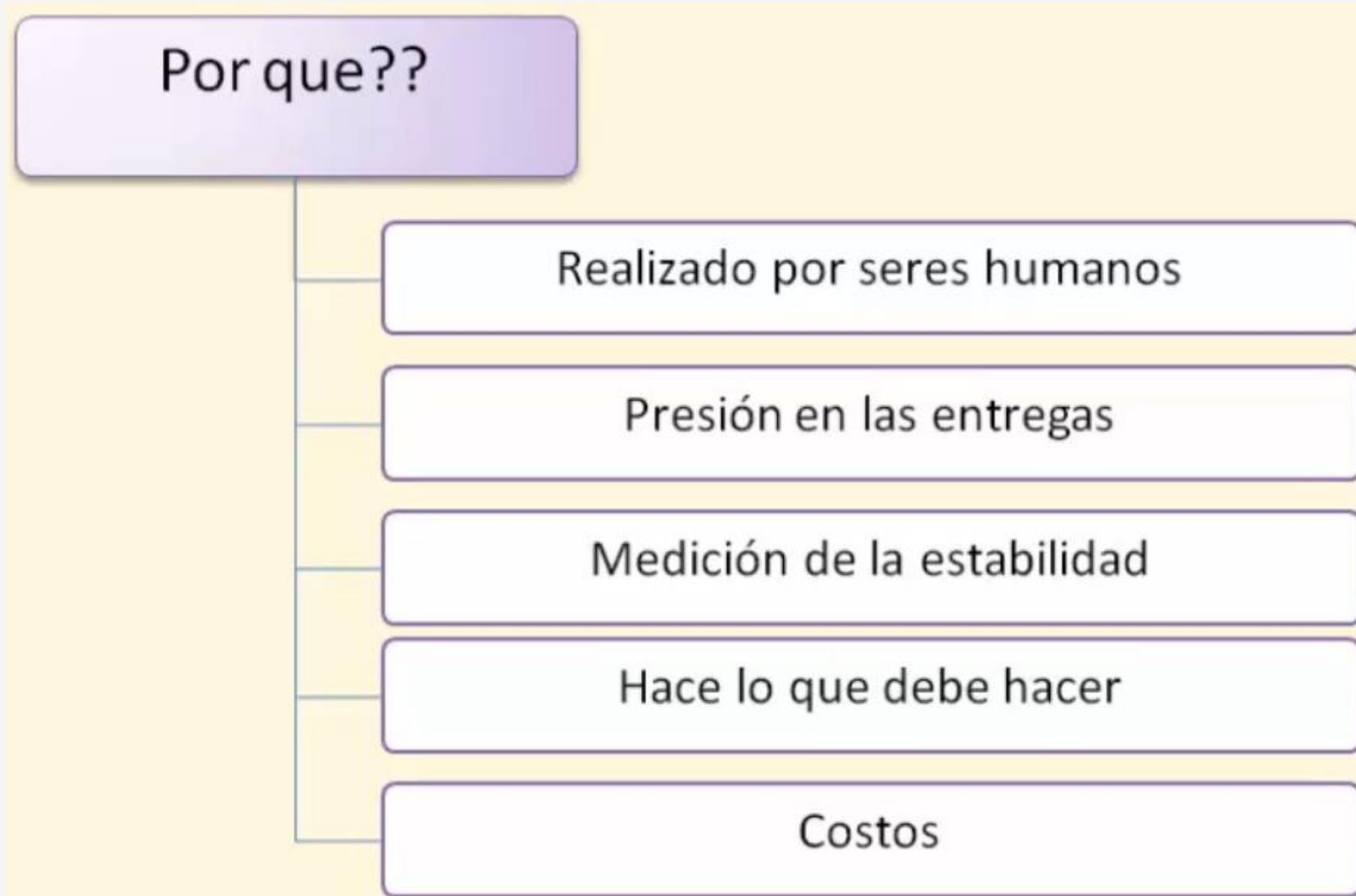


Ej.

- La Mariner 1
- Therac-25
- MIM 104
- AT&T, etc



Porque son necesarias las pruebas



Porque son necesarias las pruebas

Objetivos:

1

- Por lo tanto hay que diseñar pruebas que saque a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

2

- Para realizar pruebas efectivas un equipo de software debe efectuar revisiones técnicas formales y efectivas..

3

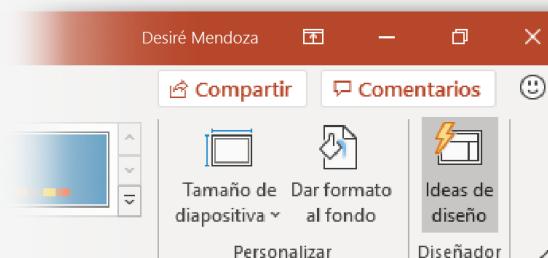
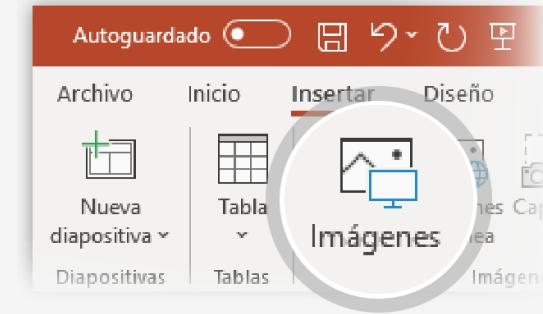
- La prueba comienza al nivel de componentes y trabaja "hacia fuera", hacia la integración de todo el sistema

4

- Las pruebas deberían empezar por lo "pequeño" y progresar hacia "lo grande" (módulos).

5

- Diferentes técnicas de prueba son apropiadas en diferentes momentos.



Que obtenemos a partir de las pruebas?

Adquirir conocimiento

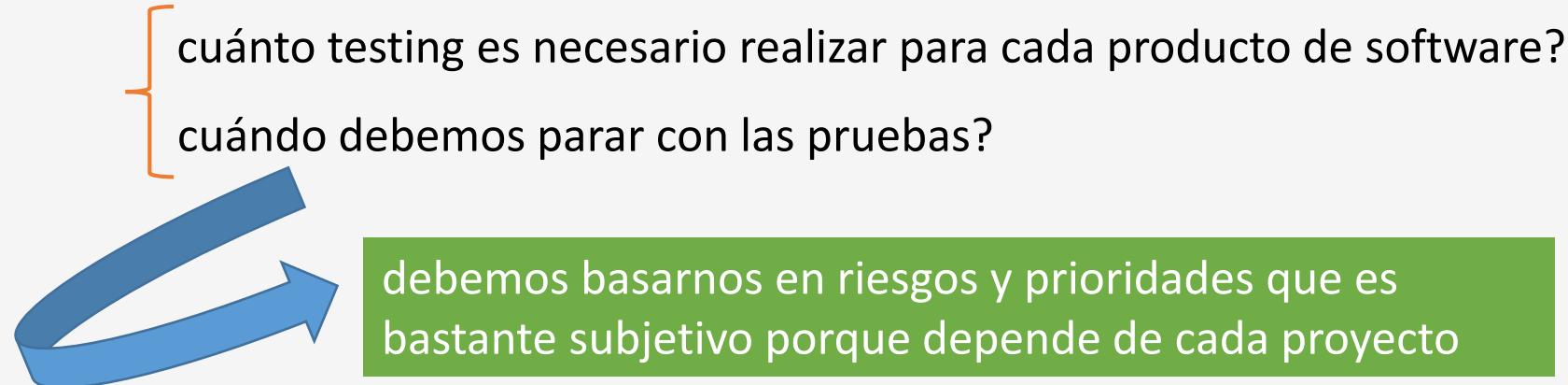
Confirmación de la funcionalidad

Generación de Información

Confianza

Cuantos testing es necesario?

El testing exhaustivo es imposible.



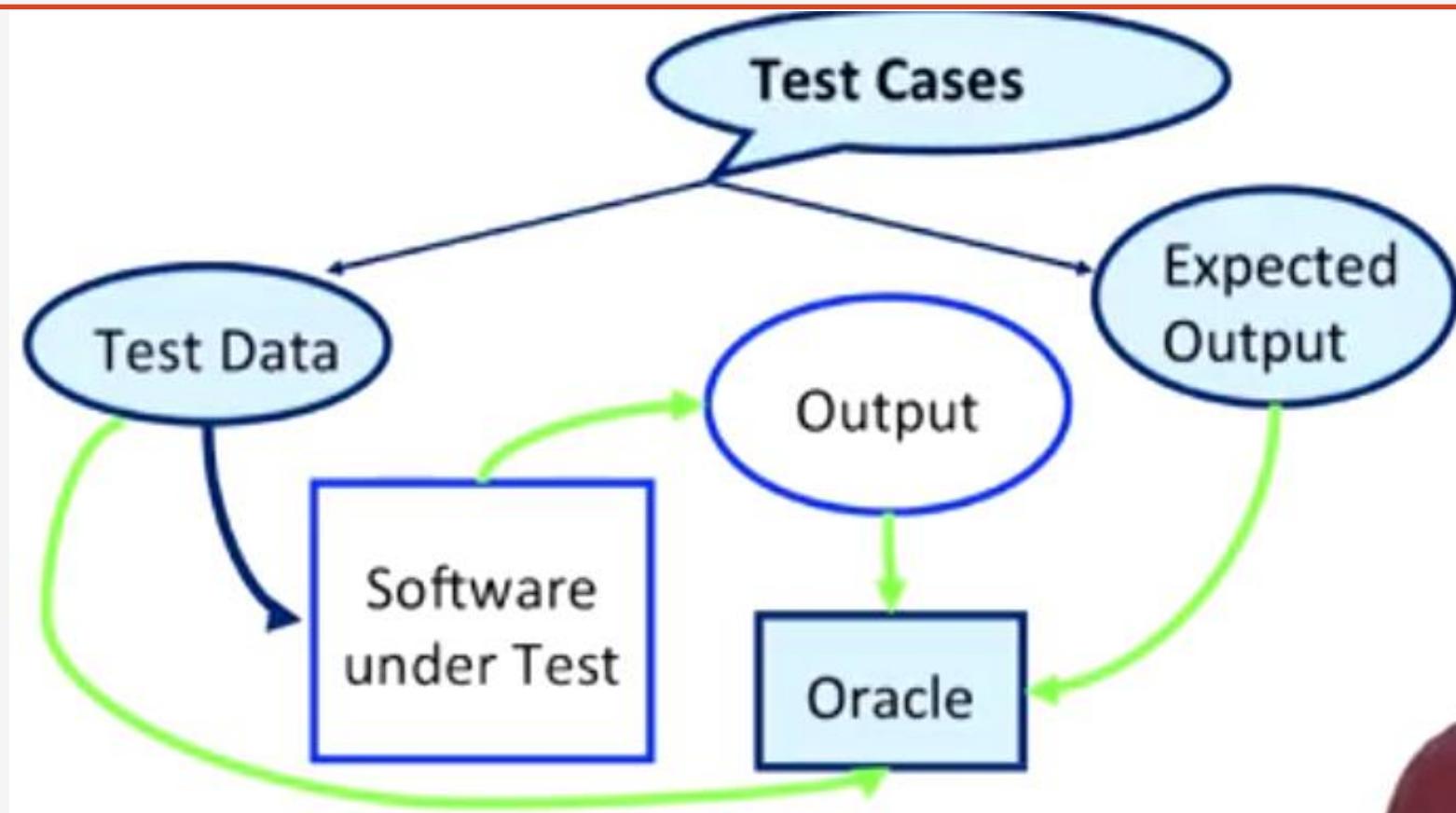
debemos basarnos en riesgos y prioridades que es bastante subjetivo porque depende de cada proyecto



- ✓ ***Criterios de Salida.***
- ✓ ***Basado en Riesgos.***
- ✓ ***Basado en Plazo y Presupuesto.***

no encontrar mas defectos puede ser un criterio para terminar las pruebas).-->
Conjunto de condiciones que se debe acordar entre todos para que el proceso concluya. Ej. Se va a terminar con el testing cuando se haya probado un 80% de la aplicación.

Proceso de ejecución la prueba?



- ✓ Proporcionar información
- ✓ Observamos la salida ?¡Que hacemos con esto.... Esta bien?
- ✓ ¿Quién debe hacer esta comparación?
- ✓ El comportamiento coincidió con lo esperado

UNIDAD 1

CONCEPTOS BASICOS DE TESTING

1.1 INTRODUCCIÓN

El Testing o pruebas de software va relacionado con lo que es la calidad de software, dentro de este contexto vamos a repasar algunos conceptos para comprender porque es necesario realizar el Testing.

"No me preocupa si algo es barato o caro. Solo me preocupa si es bueno. Si es lo suficientemente bueno, el público te lo pagará".

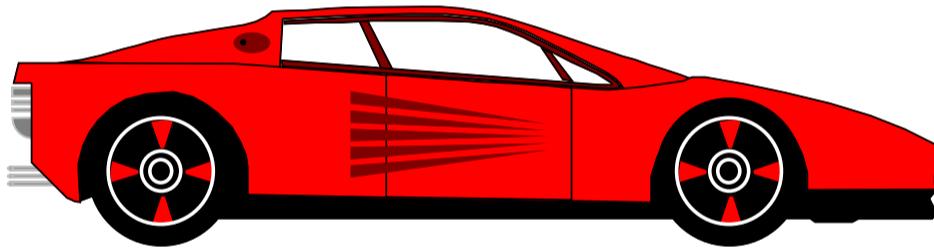
Que tiene más calidad

Los dos tienen la misma calidad siempre y cuando cumplan con sus requerimientos. Para ello debemos probar sus especificaciones



La calidad es algo subjetivo.

Si nos ponemos a definir sobre lo que es calidad, puede que sea algo subjetivo porque para una persona puede ser que un producto sea de más o menos calidad que para otra, sin embargo, para poder tener un concepto global de lo que es calidad es necesario apoyarse en estandares.



La calidad es relativa a las personas, a su edad, a las circunstancias de trabajo, el tiempo...

- Un caramelo para un niño.
- Un mapa gastronómico mundial.
- El tiempo varia las percepciones.

1.2 CONCEPTO DE CALIDAD - DEFINICIONES

- ✓ Propiedad o conjunto de propiedades inherentes a una cosa, que permiten apreciarla como igual, mejor o peor que las restantes de su especie (DRAE).
- ✓ Totalidad de las características de un producto o servicio que le confieren su aptitud para satisfacer unas necesidades expresadas o implícitas (Norma UNE 66-001-92 traducción de ISO 8402).

1.3 CALIDAD DE SOFTWARE

La **calidad** del software es el conjunto de cualidades que lo caracterizan y que determinan su utilidad y existencia. La **calidad** es sinónimo de eficiencia, flexibilidad, corrección, confiabilidad, mantenibilidad, portabilidad, usabilidad, seguridad e integridad.

Calidad del Software: la suma de todos los atributos que se refieren a la capacidad del software de satisfacer los requerimientos dados.

Cuando hablamos de calidad de software tenemos que diferenciar 2 puntos:

Calidad del producto

El punto principal es ver cómo está construido el software por dentro, analizar su código, sus componentes y como interactúan con otros. Para que el software sea de calidad si bien es cierto se debe capturar los requerimientos funcionales y no funcionales es importante determinar cuáles son los atributos de calidad, que son aquellos puntos que para el usuario o para el cliente que va a usar la aplicación son importantes ya que de nada sirve que una aplicación cumpla con todos los requerimientos funcionales y no funcionales si al final el cliente para lo que él percibe o la necesidad que él está esperando no la cumple.

Se debe aplicar diferentes estándares para ver si cumple con las características y subcaracterísticas. Por ej. La ISO 25000 que viene de la ISO 9126, pero más específicamente para lo que es calidad de software la ISO 25010.

ISO 25010, existe un conjunto de características y subcaracterísticas el cual esta norma aconseja que un software debe tener. No es necesario que el software tenga todos, sino definir un criterio de cuál es más importante de acuerdo al contexto de la aplicación.

- El modelo de calidad representa la piedra angular en torno a la cual se establece el sistema para la evaluación de la calidad del producto. En este modelo se determinan las características de calidad que se van a tener en cuenta a la hora de evaluar las propiedades de un producto software determinado

Con esta guía nos podemos apoyar para ver cuáles son estas características y subcaracterísticas que para el usuario o interesado está esperando, por tanto, hay una alta probabilidad que nuestro software sea aceptado, hay muchas técnicas para evaluar el código (técnicas de caja blanca, caja negra, etc.)



Calidad del Proceso (de software)

Se conoce de términos como CMMI/MOPROSOFT/ ISO 15504 ISO 12207. Estamos haciendo énfasis en los procesos que intervienen para la creación de este producto, no vamos a analizar ni el código ni el producto, se va a analizar los procesos que una empresa utiliza para llegar al producto final.

- **MOPROSOFT**, origen mexicano, establece los principios a implementar en una organización a través del modelo del proceso de software.
- **CMMI**, modelo de madurez integrado, también define un conjunto de procesos que una organización debe presentar para poder crear un software de calidad. CMMI1 – CMMI5.



1.4 ASEGURAMIENTO DE LA CALIDAD vs CONTROL DE CALIDAD

¿Qué es aseguramiento de la calidad?

El aseguramiento de la calidad de software es el conjunto de actividades planificadas y sistemáticas necesarias para aportar la confianza en que el

Conjunto de actividades sistemáticas que proveen capacidad al proceso de software para producir un producto adecuado para el uso.

¿Qué es el control de calidad de software?

Es la estructura que organiza evaluaciones, inspecciones, auditorias y revisiones que aseguren que se cumplan las responsabilidades asignadas, se utilicen eficientemente los recursos y se logre el cumplimiento de los objetivos del producto

Las definiciones de Control de Calidad y Aseguramiento de Calidad se parecen mucho, pues ambas hablan de técnicas y actividades para garantizar la calidad del producto.



En resumen: El Control de Calidad de Software verifica el cumplimiento de estándares definidos para el producto, y el Aseguramiento de la Calidad de Software asegura que el producto haya sido construido siguiendo los procesos establecidos para la organización.

Generalmente cuando le preguntamos a un ingeniero de sistemas sobre lo que Software Quality assurance inmediatamente piensa en testing (validación, verificación, revisiones) lo cual son solo extensiones del testing. Sin embargo en la práctica vienen a ser casi lo mismo quizás con la diferencia que el SQA tiene un contexto más global y el testing es algo más específico.

1.5 EL EQUIPO DE DESARROLLO

El desarrollo de software no es una actividad simple y por eso se hacen equipos, por ello lo primero que tenemos que empezar a definir es ¿Qué es un equipo?

Un equipo está formado por dos o más personas que trabajan juntas porque necesitan lograr un objetivo que tienen en común, y cada una de ellas se le asignan ciertas funciones que son específicas para lograr ese objetivo.

Tenemos que si nuestro equipo es pequeño puede que una sola persona cubra múltiples roles, pero en equipos más grandes es muy común tener funciones muy diferenciadas y dedicadas.

Los roles se van a asignar de acuerdo a la experiencia y las capacidades de las personas y esta identificación de los roles nos va a ayudar a estructurar el proyecto, tenemos varios roles dentro de un equipo y se los puede llamar de igual o diferente manera de los que veremos a continuación, pero normalmente estos son los básicos que encontramos en todo equipo:

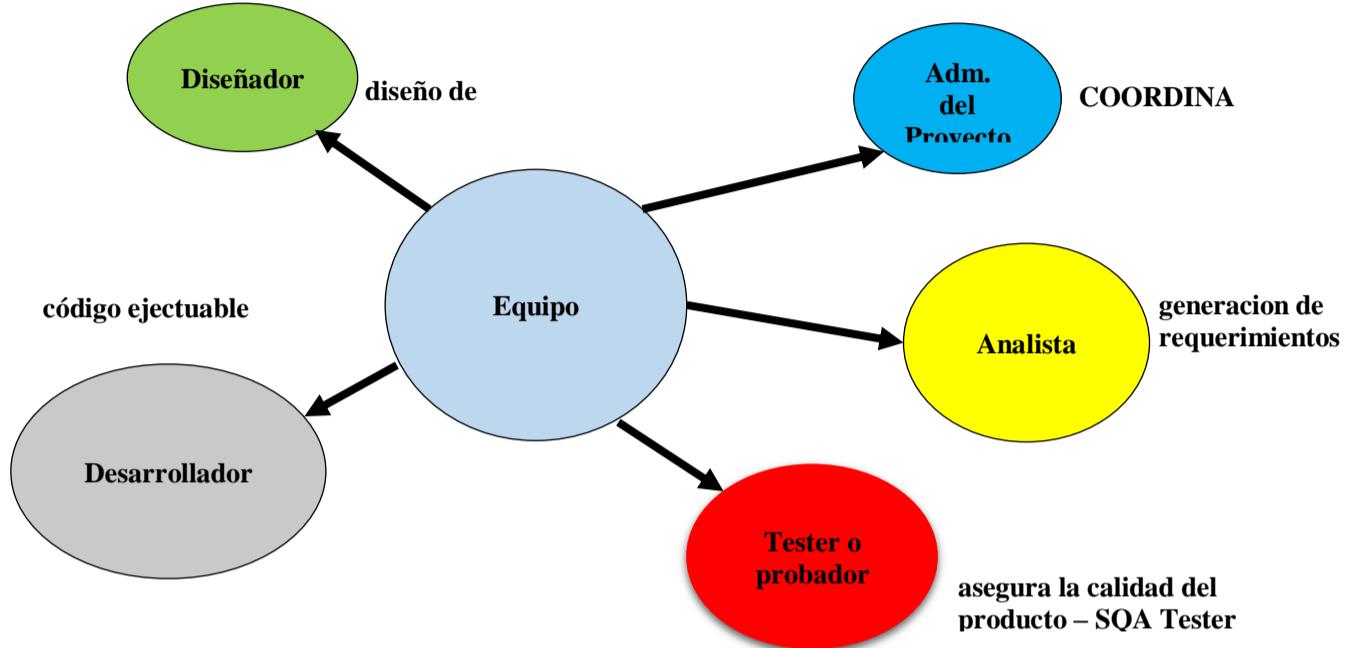
-**Project Manager** (líder de proyecto o administrador de proyecto), es quien coordina el equipo y asegura que todos los demás roles cumplan con su trabajo. Una de las preocupaciones principales que tienen estos administradores debe ser que tienen que tener una visión y misión bien clara del proyecto para que ambas se cumplan.

-**Analista Funcional**, luego de entrevistar al cliente es capaz de generar una serie de requerimientos de sistema para poder definir la estructura básica de este sistema.

-**Diseñador**, es el que toma los requerimientos que obtuvo el analista funcional y en base a ellos genera el diseño de sistemas y el prototipo.

-**Desarrollador**, toma estos requerimientos del analista y los prototipos del diseñador y los traduce en un código ejecutable utilizando algún tipo de lenguaje de desarrollo por ej. Java, C#, etc.

-**Tester o probador**, es el rol que asegura la calidad del producto, es el que va a asegurar que se concuerden con los requerimientos del cliente y los prototipos del diseñador.



1.6 PORQUE ES NECESARIO TENER SQA TESTER

Como todo proyecto importante, el éxito de un software depende del trabajo en equipo. Los equipos de TI están formados por Software Quality Assurance (SQA) testers, desarrolladores, y líderes de proyecto, quienes deben trabajar juntos como reloj suizo para crear proyectos satisfactorios.

Los miembros del equipo de desarrollo de software se pueden comparar con las herramientas necesarias para completar trabajos de la máxima calidad. Por ejemplo, para construir una silla, necesitaremos madera, tornillos, un martillo, un manual de instrucciones, una sierra eléctrica, pintura, una cinta métrica y lija. Por supuesto, usted seguramente pensará que puede construir una silla con menos herramientas, y estaría en lo correcto; pero no será la mejor silla, no tendrá buena estabilidad, ni soporte, tampoco será estética y seguramente tampoco cómoda. Muchas veces para ahorrarnos materiales o pasos nuestro producto no acaba siendo el más óptimo ni adecuado.

Lo mismo pasa cuando queremos desarrollar un software, una aplicación o un sitio web: no basta con tener solo un desarrollador es necesario tener un equipo de profesionales con funciones específicas. En ese sentido, los software QA testers son esenciales para cualquier equipo de desarrollo de software, pues generan un valor agregado al producto que se va a entregar.

1.6.1 Importancia del SQA

El software testing es fundamental, ya sea implementando como un rol de soporte para desarrolladores o como una entidad independiente. Hay que hacer varias pruebas de software para poder detectar los errores, corregirlos y entregar un producto de buena calidad.

Es necesario que los software SQA testers trabajen de forma paralela con los desarrolladores, pues una comunicación más efectiva permite al equipo encontrar errores y hacer mejoras durante todo el proceso.

Un software quality assurance tester le ahorra dinero a la empresa, identificando y corrigiendo errores durante la fase de desarrollo. En comparación, si no se realizará ningún control de calidad hasta después de terminado el producto, los errores encontrados costarían más tiempo y por lo tanto, dinero para corregir.

De los múltiples beneficios que el SQA brinda al proyecto, el más importante es que **asegura la satisfacción del cliente**, lo cual a su vez mantiene la reputación de su empresa. Es por esta razón que las mejores compañías no escatiman en el salario de los software quality assurance testers.

1.6.2 Quality Control vs. Quality Assurance

Quality Control (QC) y Quality Assurance (QA) son dos términos que ocasionalmente se usan indistintamente, sin embargo, se refieren a distintos aspectos de la administración de calidad.

A través del Quality Control, el equipo verifica que el producto cumple con los requerimientos funcionales. El Consejo Internacional de Calificaciones de Pruebas de Software (ISTQB por sus siglas en inglés) lo define como la serie de actividades diseñadas para evaluar la calidad.

Por otro lado, Quality Assurance son los procesos que auditán los resultados de las mediciones de Quality Control, para garantizar que los estándares de calidad se están cumpliendo. Estos procesos son los que previenen errores y defectos en el producto.

1.6.3 ¿Cómo se lleva a cabo el QA y por quién?

El control de calidad es ejecutado durante todas las fases del ciclo de vida del desarrollo de software (SDLC por sus siglas en inglés). Realizar un control continuo durante cada etapa o iteración evita que se cometan errores significativos que causen un retraso en el proyecto y un mayor gasto de presupuesto.

Los software quality assurance testers son los responsables de garantizar este control de calidad y no necesariamente tienen que formar parte de la empresa, existe la opción de completar un equipo de desarrollo con un SQA tester mediante la subcontratación o mediante Aumento de Personal.

El Staff Augmentation es una alternativa flexible para completar temporalmente los equipos de desarrollo con talento profesional. De esta manera las organizaciones se ahorran el costo de infraestructura y entrenamiento, a comparación de contratar un QA tester a tiempo completo.

1.6.4 Actividades de un Software QA Tester

Las actividades de un software QA tester durante la ejecución se pueden resumir en los siguientes pasos:

- **Análisis de Documentación**

El primer paso que hacen los SQA testers es el análisis de documentación del proyecto, porque tienen que entender en su totalidad los requerimientos del cliente. De este modo se aseguran que los aspectos funcionales y no funcionales del proyecto cumplan con las especificaciones.

Posteriormente, planifican la estrategia de la prueba de software, el alcance la prueba, y establecen los plazos. Adicionalmente, definen todas las herramientas que se van a utilizar para buscar los errores, los métodos, recursos y responsabilidades a los demás SQA testers.

- **Diseño de planes**

El siguiente paso es diseñar los planes, aquí se elaboran casos de prueba y listas de verificación donde abarcan todos los requisitos del Software. Cada caso de prueba tiene que tener condiciones, datos y los pasos necesarios para validar que funcione, además de un resultado esperado bien definido para poder comparar contra los resultados reales.

- **Ejecución de pruebas y reporte de defectos**

Este es el paso donde se ejecutan las pruebas y se reportan los defectos, se empieza con los desarrolladores realizando unit tests y los SQA testers realizan pruebas en los niveles de API y UI. Los errores que se detectan se envían a un sistema de seguimiento de defectos. Una vez que se encontraron los errores y se corrigieron, los SQA testers vuelven a probar las funciones para asegurarse que no les haya pasado desapercibido algún otro error. También hacen pruebas de regresión para verificar que las correcciones no hayan afectado las demás funciones.

Por último, ya que los desarrolladores hicieron un reporte con las funciones implementadas y errores corregidos, el SQA elabora un informe final en el cual se certifica que el producto cumple con todos los

requerimientos de desarrollo, cumple con los más altos estándares de calidad y está listo para ser lanzado al público.

Sin embargo, el trabajo de un Software QA tester no termina allí, puesto que aún después del despliegue a producción, se deben ejecutar pruebas post-producción para evaluar aspectos excluidos en el ambiente de testeo. Un ejemplo de esto es la diferencia de datos, como el número de usuarios que el producto puede soportar en un determinado momento.

Si bien los desarrolladores son los miembros del equipo de TI de quienes más se habla (y sin duda son vitales), los software quality assurance testers no deben de ser subestimados. Ellos trabajan mano a mano con programadores para prevenir, solucionar, y documentar errores en el producto final. Sin su buen ojo, un software plagado de errores podría llegar a desplegarse, causando grandes problemas en el futuro.

Por lo tanto, si está pensando en armar un equipo de desarrollo de software, asegúrese de que esté completo. Servicios de Aumento de Personal o Equipos Dedicados pueden ayudar a llenar cualquier vacío en sus recursos de talento actuales, asegurando que su proceso de desarrollo sea eficiente, efectivo, y de calidad superior.

1.7 QUE ES TESTING O PRUEBAS DE SOFTWARE

- Metodología para encontrar defectos de software.
- Proceso utilizado para medir la calidad de cualquier software.
- Proceso de verificación del correcto funcionamiento de una aplicación.

Existen muchas definiciones y todas son válidas, pero básicamente *las pruebas de software o testing es el proceso que permite verificar la calidad de un producto de software. Permite identificar posibles fallos en la implementación, calidad o usabilidad de un programa*. No solo es que vamos a ejecutar la prueba 1, prueba 2, etc. *incluye muchas otras actividades desde la planificación de las pruebas, la preparación de las pruebas, la evaluación de los productos de software para determinar si cumple o no con los requerimientos especificados*.

EL TESTING PUEDE PROBAR LA PRESENCIA DE ERRORES, PERO NO LA AUSENCIA DE ELLOS.

Durante la planificación determinamos que pruebas vamos a correr, en la ejecución vamos a correr esas pruebas, pero que pasa si existe alguna funcionalidad que no estamos tomando en cuenta o al algún “test case” no cumple una determinada funcionalidad por completo.

Para realizar un buen testing de software debemos tener en cuenta varios conceptos involucrados y la diferencia entre ellos.

- **Error**, es una acción humana que produce un resultado incorrecto. Es una idea equivocada de algo, como las ideas las tienen las personas el error es una equivocación del desarrollador o del analista. Un error puede llevarnos a generar uno o más defectos. Ej. error en la programación, error en el typeo, un requerimiento que este mal especificado.

Defecto, es un desperfecto que se encuentra ya sea en un componente o en un sistema, y que puede causar que este componente o este sistema falle en su funcionamiento. Ej. Una sentencia o una definición de datos que este incorrecta.

La relación entre defecto y fallo es que un defecto causará un fallo. Es decir, si hemos localizado un defecto durante una ejecución entonces podremos causar un fallo en el componente o en el sistema. Ej. Que el desarrollador hubiese utilizado el operador menor en lugar de menor o igual.



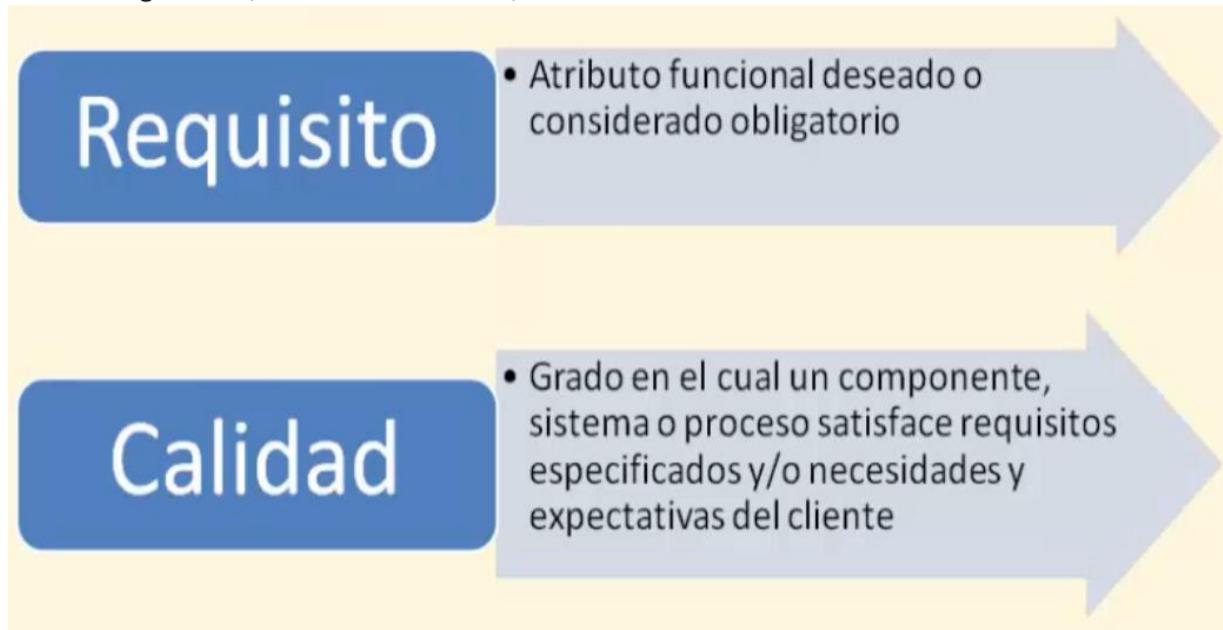
Fallo, es la manifestación física o visible de un defecto. Si un defecto es encontrado durante la ejecución de una aplicación, entonces va a producir un fallo. Ej. Un crash del sistema.

¿Cómo se relacionan estos tres conceptos?

Un error puede generar uno o más defectos, y un defecto va a causar un fallo.

Las causas de los fallos de pueden deber a dos grandes grupos:

- **Error humano**, porque el defecto se introdujo en el código de software, en los datos o en los parámetros de configuración, y ¿porque hubo un error?, porque tenemos plazos cortos para entregar las cosas o porque tenemos mucha complejidad o porque el desarrollador o analista se distrajeron.
- **Condiciones ambientales**, los cambios en las condiciones ambientales como ser: radiación, magnetismo, fallo en el disco duro, etc.



-**Requisito** (o requerimiento), describe un atributo funcional deseado que es considerado obligatorio. Significa que un sistema no va a ser aprobado por el cliente sino cumple con esos atributos que el mismo ha definido. Ej. El sistema permita editar o eliminar un determinado contacto, no se permita continuar hasta que no se haya registrado un campo obligatorio, que sea fácil de instalar (requisito no tan tangible).

-**Calidad** (calidad de software), es el grado en el cual un componente, un sistema o un proceso satisface los requerimientos o requisitos especificados y las expectativas del cliente. La calidad del software es *la suma de todos los atributos que se refieren a la capacidad del software de satisfacer los requerimientos dados*.

Los atributos pueden estar categorizados en dos grupos:

Atributos de Calidad

Funcionales	No Funcionales
<ul style="list-style-type: none">• Correctitud• Completitud	<ul style="list-style-type: none">• Fiabilidad• Usabilidad• Portabilidad• Eficiencia• Mantenibilidad

- a) **Atributos funcionales:** (tendremos 2 características básicas y sumamente importantes)
- correctitud**, la funcionalidad satisface correctamente los atributos requeridos.
 - completitud**, la funcionalidad de software satisface todos los requisitos que el cliente ha pedido, incluirá: adecuación, exactitud, interoperabilidad, seguridad y el cumplimiento de la funcionalidad.

- b) **Atributos no funcionales:** (son más difíciles de lograr porque no están exactamente definidos o bien establecidos)
- fiabilidad**, el sistema va a mantener su capacidad y funcionalidad a lo largo de un periodo determinado de tiempo. Ej. Si tengo un bug que se presenta cada dos días mi sistema no es confiable.
 - usabilidad**, el sistema es fácil de usar, es fácil de aprender, tiene un uso intuitivo y se ha desarrollado conforme a las normas iso 9000, etc. de acuerdo al sistema.
 - portabilidad**, que sea fácil de instalar o desinstalar, que sea fácil de configurar los parámetros necesarios, que sea fácil de trasferir a otro entorno.
 - eficiencia**, que el sistema requiera de un mínimo de recursos para ejecutar una tarea determinada. Ej. Si el logeo tarda 30 seg. mi eficiencia no es buena.
 - mantenibilidad**, es una medida del esfuerzo que se requiere para realizar cambios en los componentes de un sistema.

Esto no significa que mi sistema deba tener todos estos atributos funcionales o no funcionales, el cual será dependiendo del sistema en el que estamos trabajando vamos a tener que priorizar los atributos que necesitamos, y en base a eso determinaremos que tipo de testing aplicaremos a nuestro sistema. Ej. Aplicación de escritorio no hace falta priorizar la portabilidad. Aplicación web si necesitamos la eficiencia.

1.8 PORQUE SON NECESARIAS LAS PRUEBAS

El software está en todo lugar. Se podría decir que está presente en todos los aspectos y momentos de la vida de las personas o mejor dicho en casi todos. Algunas veces el software es obvio, otras veces no.

¿QUE SON LAS PRUEBAS?

Es una manera de explorar, de revisar el producto que tú quieras desarrollar, experimentarlo, verlo, entenderlo. Entre menos entiendas el producto que estas desarrollando más errores va a cometer.

¿Pasando todas las pruebas tendré un software sin errores?

¿Al hacer miles de pruebas se tiene un software sin errores? → NO

La seguridad de que el 100% de que un software no tenga errores no va a suceder, podrá llegar a un buen porcentaje de un 99%.

Se puede tener un software muy estable pero cuando los usuarios empiezan a utilizar nuevos dispositivos, salen nuevas versiones de software, etc. Entonces se debe actualizar el software para que funcione correctamente con los nuevos dispositivos.

El tiempo y los clientes nos han enseñado que pese a tener un buen plan de pruebas siempre se nos puede ir algunos detalles.



Para aclarar el motivo de esta necesidad veremos algunos ejemplos:

- La Mariner 1**, en el año 1962, fue un cohete la 1ra. Misión de la NASA para sobrevolar Venus, el cohete no duro más de 5 min. en vuelo cuando desvió su trayectoria teniendo que ser autodestruido por la NASA. El motivo la omisión de un guion en el programa que controlaba el cohete. ¿Qué hubiera pasado si se le hubiera realizado el testing necesario?
- **Therac-25**, máquina de radioterapia usada para entornos médicos, entre 1985 y 1987 hubo seis accidentes en la que los pacientes recibieron una sobredosis de radiación donde varios murieron. Se determinó que el software tenía en su diseño un código indocumentado. Otro ejemplo de porque es necesario hacer muchas pruebas y distintos tipos de pruebas.
- MIM 104**, misil antiaéreo que se usaba para interceptar otro tipo de misiles a modo de defensa, durante la guerra del golfo en 1991 un misil mato 28 soldados debido a una falla debido a un error en el software del reloj del sistema porque se había retrasado en un tercio de segundo al haber estado activado 100 horas seguidas.

- Apple enseña la nueva característica de reconocimiento facial, la cual falló
- Windows, la pantalla azul WIN98 Bill Gate fue descubierto en una presentación.

No solamente hay situaciones que pueden poner en duda si lo que estamos haciendo está bien o no, sino también situaciones legales de manera implícita → caso UBER, en su cobertura de pruebas no estimaron la parte de seguridad, causando el despido del responsable de seguridad.

A pesar de toda esta situación mucha gente que desarrolla software, pequeñas o medianas empresas, free lancers, no consideran las pruebas como parte del desarrollo ya sea porque contractualmente no lo incluyeron, o les faltó tiempo para terminar y la clásica respuesta de que no se pudo y será en la siguiente iteración, o la cobertura que tienen es muy mala y nada más están pensando en algunos procesos y no en todo el flujo de datos, ni en la parte de BackEnd, ni en la Base de Datos, o en la seguridad, etc.

Así como esto existe millones de ejemplos, el software es parte de nuestra vida, constantemente estamos interactuando con él en todos los ámbitos: Ej. cuando prendemos el microondas, cuando encendemos el televisor, por eso es necesario que el software sea de calidad.

Por lo tanto, podemos resumir que hay **muchas razones por lo que un software debe ser probado** pero las más importantes son:

- **Es realizado por seres humanos**, los seres humanos cometemos errores, podemos saber mucho o poco pero no vamos a saber todo. Podemos tener muchas habilidades, pero no vamos a ser perfectos.
- **Presión en las entregas**, no hay tiempo para chequear las cosas que hacemos y asumimos que está todo bien y eso nos lleva a cometer errores.
- **Medir la estabilidad del software**, es otra de las razones por las que hacemos pruebas.
- **Hacer lo que debe hacer**, para demostrar que el software no tiene fallas y que hace lo que tiene que hacer.
- **Costos**, encontrar las fallas durante el proceso de desarrollo es mucho más barato que cuando ya está en producción y lo está usando el cliente.

1.1 Objetivos de las pruebas

- **Adquirir conocimiento**, para conocer los defectos que tiene un objeto de prueba y así describirlos de tal forma que facilite su corrección.
- **Confirmación de la funcionalidad**, si se ha implementado tal cual como se había especificado.
- **Generación de información**, proporcionamos información sobre posibles riesgos relacionados al sistema antes que este sea entregado al usuario.
- **Confianza**, que el sistema cumple con la funcionalidad esperada.

El testing exhaustivo es imposible, así como probar todas las combinaciones de entrada y condiciones, y no terminaríamos de probar nunca porque hay infinitas posibilidades, entonces como sabemos cuánto testing es necesario realizar para cada producto de software, cuando debemos parar con las pruebas, para esto debemos basarnos en riesgos y prioridades que es bastante subjetivo porque depende de cada proyecto pero básicamente por un lado tenemos "**los criterios de salida**" (no encontrar más defectos puede ser un criterio para terminar las pruebas).--> Conjunto de condiciones que se debe acordar entre todos para que el proceso concluya. Ej. Se va a terminar con el testing cuando se haya probado un 80% de la aplicación.

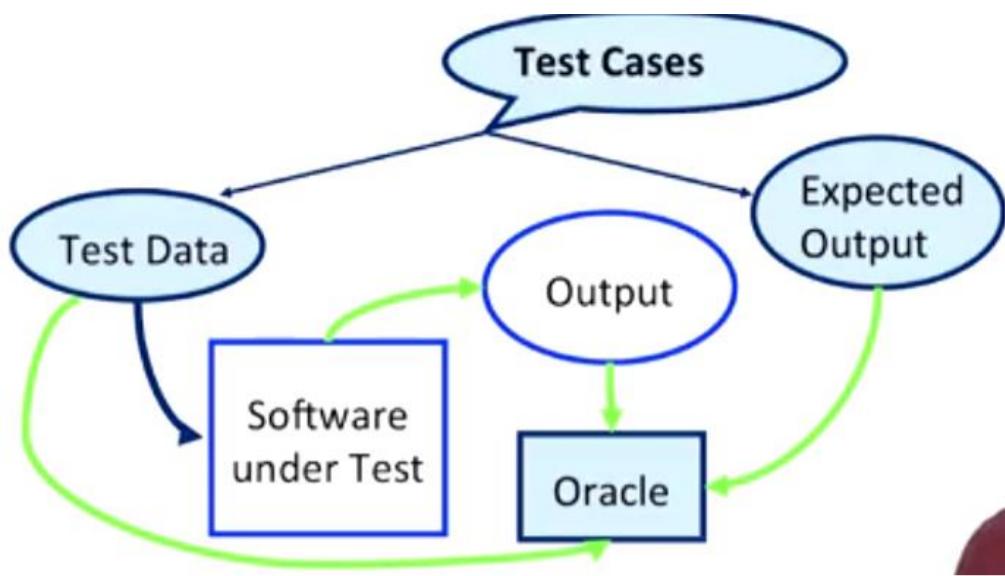
1.9 RAZONES PARA HACER PRUEBAS

Debemos considerar los siguientes aspectos:

- Si decidimos querer tener un software bien hecho, estamos viendo el problema y estamos en mejoras continuas, que nos lleve a ver qué pruebas realizar para que en la siguiente liberación de software trabaje de la forma deseada.
- Los costos se están volviendo muy altos, hay equipos que están resolviendo problemas en lugar de tener buenas prácticas y monitorear todo el proceso, no conocer lo que se está queriendo hacer puede ser que genere más defectos dentro del mismo software porque al tratar de arreglar algo vamos a descomponer otras partes del software o todo lo demás.
- Estándares o implicaciones legales, qué condiciones debe tener el usuario para vender el software cumplir los estándares tanto de desarrollo, entrega, etc.

1.10 PROCESO DE EJECUCIÓN DE PRUEBAS

- ✓ Proporcionar información
- ✓ ¿Observamos la salida? Que hacemos con esto...! ¿Está bien?
- ✓ ¿Quién debe hacer esta comparación?
- ✓ El comportamiento coincidió con lo esperado



Que es una prueba y como se puede ejecutar algunas pruebas simples en su propio código.

Comenzamos con lo que se denomina “software bajo prueba”, no se puede probar algo que no está creado (exceptuando TDD) ya que estamos hablando de como ejecutar una prueba. Algo que solo se puede realizar una vez que se tenga el software a ejecutar.

Cuando decimos software nos referimos no necesariamente al producto de software terminado, sino todo lo contrario, aunque finalmente llegaremos a ese punto.

Software bajo prueba significa cualquier parte o subconjunto del programa que hemos completado donde podemos ejercer un comportamiento, este es algún modulo o unidad de código (pruebas unitarias).

Las unidades en este contexto generalmente significan algo así como un método, una función, subrutina o procedimiento, algún pequeño conjunto definido de pasos a tareas que tenemos una expectativa de cómo debería comportarse cuando ejecutamos este código.

Para ejecutar una prueba contra nuestro software o unidad, debemos proporcionar la información sobre la cual actúa la entrada, es decir los datos de prueba.

Hay muchas formas de seleccionar o generar estos datos de prueba.

- Basados en un perfil de cómo creemos que actuara el usuario.
- Basados en probabilidad, o estudios de usuarios
- O podemos atacar el código con algunos datos que a menudo causan errores como ser: “cero” 0, valores mayores, menores, palabras, números, null, etc.

Para cada una de estas entradas, ejecutaremos el software bajo prueba, dados los datos de prueba potencialmente después de configurar el software para que este en el estado correcto para que los datos de prueba de software tengan sentido.

Una vez que el software recibe los datos de prueba, observamos la salida, el comportamiento del programa dada la salida.

¿La pregunta entonces es?

¿Qué hacemos con esto? ¿Está bien?

¿El software ha proporcionado el resultado correcto dado los datos de prueba?

Algo o alguien tiene que hacer esa comparación, llamado “Oracle”, que viene a ser el desarrollador o probador que ejecuta las pruebas.

Se tiene el software funcionando, ingresan los datos, mira lo que pasa y decide si el comportamiento coincidió o no con lo que esperaban.

Ahora, como se puede saber o no de que puede existir un error humano en esto, ¿Los humanos somos particularmente confiables? ¿Puede decidir la diferencia entre el número “uno” 1 y la “letra l minúscula” cuando se escribe?

Es así que lo que se está empezando a ver y utilizar son los casos de “Oracle” automatizados que comparan algún resultados esperados y conocidos, determinados o recuperados a través de frameworks como ser: JUnit, PyUnit, etc.

El framework verificará automáticamente que ciertos casos sean verdaderos o falsos y proporcionará un informe visual, comparando esto con la salida que generó el software.

1.11 ¿QUE HABILIDADES NECESITA UN TESTER?

todo buen tester tiene ciertas habilidades que lo hacen ser un analista o ser una persona con un pensamiento bastante crítico.

Algunas de las más importantes son:

Que habilidades necesita un tester?



Pensamiento Lógico

Debe saber como desglosar un Sistema en unidades más pequeñas para poder crear casos de prueba



Ser organizado y metódico

Esto es clave para poder ejecutar los casos de prueba en un orden y poder encontrar la mayor cantidad de errores



Muy buena comunicación

Excelente comunicación verbal y escrita para comunicar los errores y documentarlos.



Apasionado por la tecnología

Todo Tester tiene que tener un gusto por la Informática, ser curioso y creativo



Atención a detalles, curiosidad y sentido común

Se requiere tener un pensamiento crítico con atención a detalles desde el punto de vista de un usuario final



Paciencia y persistencia

El Desarrollo de software es un proceso que requiere flexibilidad y mucha paciencia.

✓ Tener un pensamiento lógico.

Poderes de ser capaz de desglosar una aplicación, un sistema en unidades más pequeñas para poder crear casos de prueba. Entonces tú tienes que saber cuándo estás probando una aplicación, la tienes enfrente tuyo, ver la aplicación, hacer un análisis funcional de la aplicación y saber cómo se hace para sus opciones, para que sirve cada una y qué impacto tiene cada una de esas opciones para el negocio.

Se tiene que pensar como desde el punto de vista del negocio y también como desde el punto de vista de usuario y poder desglosar todas estas funcionalidades en unidades más pequeñas y poder así crear casos de prueba, entonces debes de tener un pensamiento muy lógico.

✓ Tener también muy buena comunicación

Una excelente comunicación verbal y escrita para comunicar los errores y comentarlos.

Como testers, nosotros estamos en constante comunicación con diferentes personas dentro de nuestro equipo, especialmente con los desarrolladores y con los análisis de requisitos y también con el dueño del producto. Entonces debemos de ser capaces de poder comunicar todo lo que vamos encontrando de una manera muy efectiva.

Hay personas que se les dificulta explicar cómo reproducir un error o como escribir los pasos para reproducir cierto escenario o en ciertos casos de prueba, o también se la dificultad explicárselo a una audiencia más grande, entonces nosotros tenemos que tener estas habilidades de muy buena comunicación.

✓ Atención al detalle

Esto es fundamental, los tester son atentos a los detalles. Deben de ser curiosos, y deben tener un sentido común. Cuando nosotros estamos probando una aplicación, se requiere tener un pensamiento crítico con atención a detalles desde el punto de vista de un usuario final. Si digamos que es una aplicación web y nosotros como tester tenemos que simular que somos un usuario final.

Entonces cualquier cantidad de detalles, por más mínimos que sean, nosotros tenemos que revisarlos y verificar que funcionen perfectamente. Hay que mirar detalles, hay que ser bastante críticos con todo lo que estamos mirando, porque dependiendo de lo que se esté probando, si no tenemos esta atención ante esos detalles, cualquiera de los pasos se puede convertir en un defecto y generar hasta un fallo total del sistema y la aplicación del módulo puede caerse o no funcionar cuando ya esté funcionando en producción.

✓ Ser muy organizados y metódicos

Esto es clave para poder ejecutar los casos de prueba en un orden y poder encontrar la mayor cantidad de errores ya que cuando nosotros estamos probando algo tiene que tener un cierto orden y hemos de seguir

como una metodología o los casos de prueba deben de estar organizados de una manera lógica y secuencial, porque para probar un caso de prueba hay algunos requisitos.

Muchas veces cuando vamos a hacer, por ejemplo, una prueba de una página de autenticación de usuario y tenemos que ingresar un usuario y una clave, antes de tener acceso a esa clave, deberíamos de registrarnos en otra plataforma y para registrar esa plataforma tendríamos que tener unos datos de prueba. Es así que debemos tener un orden lógico de los pasos que hay que hacer para poder ejecutar un caso de prueba.

Al ser organizados en ese sentido y de la misma manera, debemos ser muy organizados en la documentación de las evidencias de la prueba, estamos ejecutando unos casos de prueba y no tiene, y tenemos las evidencias como las fotos o los vídeos de cada uno de esos casos. Están desordenados, están por todas partes no vamos a poder encontrarlos fácilmente o reportarse a un desarrollador en caso de que haya un problema. Hay que ser muy organizados en la documentación, en la ejecución de cada uno de los pasos.

✓ **Ser apasionados por la tecnología**

Los testers tienen que tener un gusto por la informática, tiene que ser curioso y creativo, revisar las opciones y ser capaces también de poder configurarlas fácilmente para poder encontrar la mayor cantidad de errores a partir de toda esa curiosidad que debe tener un buen tester.

✓ **paciencia y la persistencia**

El desarrollo de software es un proceso que requiere flexibilidad y mucha paciencia.

Porque cuando se está probando una aplicación vamos a encontrarnos con una cantidad de problemas que no nos tienen que parar ya que seguramente no nos van a permitir continuar fácilmente. Entonces, como está en nosotros, tenemos que tener mucha paciencia y entender que el desarrollo del software es un proceso muy complejo y que requiere de repetición.

Hay que tener mucha paciencia y persistencia, porque hay que probar y probar y volver a repetir los mismos casos, hasta que todos los errores de todos los casos te pasen completamente y responden exitosamente de manera que no tengamos ningún otro problema.

1.12 RESPONSABILIDADES DEL TESTER O QA EN UN EQUIPO DE DESARROLLO DE SOFTWARE



la diferencia entre tester y QA en la práctica es lo mismo. En las empresas los tratan de igual manera, pero en la teoría un analista de QA difiere un poco de lo que es el tester, porque **el analista de Aseguramiento de Calidad o Quality Software Assurance se encarga más del aseguramiento de la calidad y de todo el proceso de desarrollo del software viéndolo a manera macro, mientras que el tester solamente se encarga de la ejecución de las pruebas como tal, pero eso es un concepto que vamos a mirar más adelante.**

En la práctica las empresas se refieren al tester o QA como la misma persona.

Algunas de las responsabilidades más importantes son:

✓ **el diseño de un plan de pruebas como tester**

Son los encargados de realizar un plan de pruebas. Eso depende mucho del tipo de proyecto que estamos trabajando, si es un proyecto muy grande, vamos entonces a elaborar un plan de pruebas, que es como si

fuerza un mapa o una hoja de ruta en lo que se va a hacer y la estrategia que, a usar, como se va a probar, con qué tipo de datos probar, etcétera.

✓ **Definir los casos de prueba con base a los requisitos del cliente o el negocio**

Son los encargados de hacer un análisis funcional de la aplicación y entender muy bien qué es, lo que hace y cuáles son los requisitos que la empresa está solicitando al equipo de desarrollo.

Interpretar esos requisitos y crear una cantidad de escenarios de prueba que va a permitir encontrar los errores después. (diseñar casos de prueba y ejecutarlos).

✓ **Gestionar el ambiente y los datos de prueba**

Cuando se está probando una aplicación, ya sea en una empresa o para un cliente externo, se tiene que hacer una instalación de nuestro sistema. Si creemos que es una aplicación que va a estar instalada en nuestro computador, entonces tenemos que gestionar todo un ambiente de pruebas para que esta aplicación pueda correr correctamente en nuestro computador.

Si es en una red interna de una empresa, tenemos que gestionar todos los permisos necesarios, accesos de la base de datos o todo lo que sea necesario para que esa aplicación pueda correr exitosamente y podamos efectivamente ejecutarla y probarla. Para eso debemos guiarlos con la ayuda de un desarrollador o ingeniero de infraestructura o con cualquier persona dentro de la empresa que nos ayude a configurar la aplicación para que podamos ejecutar las pruebas.

A eso se refiere toda la gestión del ambiente y de igual manera a los datos de prueba, etc.

✓ **Ejecutar los casos de prueba**

Obviamente, ya después de haber diseñado todos estos casos, es quien está encargado de ejecutar estos casos de prueba con todas las combinaciones posibles para poder encontrar todos los errores el tester los ejecuta y también responsable de documentar todas las evidencias de las pruebas. Ej. una aplicación en un celular, vamos a estar probando todas las opciones que nos ofrece una aplicación, estar grabando todo lo que está pasando allí para poder tener una evidencia por donde se pasó y que se probó. Cuando encontramos un error poder mirar cómo ser por medio de un video, por ejemplo, de cómo se genera el error, poderlo replicar nuevamente y poder dar la muestra a un programador para su futura corrección.

✓ **Documentar todo ese tipo de cosas**

y tenerla en un repositorio de evidencias y que se pueden ver en un documento de Word o un documento Excel como un administrador de casos de prueba.

✓ **Reportar los errores encontrados y realizar seguimiento para su corrección y revalidación**

Una vez que se ha encontrado una cantidad de bugs o errores en el sistema se va a reportar al programador y tenemos que reportarlo de una manera específica. Se puede utilizar un formato específico o formato que ya exista en un sistema de manejo o administración de efectos de errores. También es el encargado de hacerle seguimiento a este error hasta que sea corregido y se pueda probar nuevamente hasta que ya el error sea corregido y termine todo su ciclo de vida.

✓ **Participar en todas las reuniones de seguimiento que tenga el equipo**

En los eventos que dictamina la metodología Scrum o la metodología que se tenga definido.

Ej. metodología Scrum, en el Desarrollo de software para poder construir software de una manera ágil. El test es participar en todos estos eventos que se hacen diario y semanalmente.

✓ **Realizar informes de calidad del producto una vez se han ejecutado**

De todo un set de pruebas, el tester debe crear un informe de calidad del producto con todos los hallazgos que ha encontrado y lo provee al equipo de desarrollo para que posteriormente se puedan corregir todos los defectos, todos los errores que se han encontrado y también que puedan tomar decisiones acerca de la calidad del producto.

Esta información la usan mucho los Managers o los dueños del producto (product owner) para tomar decisiones sobre si un producto o un sistema completó cumplir con las expectativas para poder pasar a producción o poderla liberar al público en general.

✓ **Ayudar a resolver las dudas a los analistas de requisitos y a los dueños del producto (PO)**

Como son los testes los que están probando y están adquiriendo el conocimiento del sistema que se está probando en el momento, es responsabilidad del tester resolverles dudas a los analistas de requisitos, inclusive a los desarrolladores o a otros tester en cuanto a la funcionalidad o a la manera como se ve, cómo funciona la aplicación que estamos probando, porque nos volvemos como unos expertos en lo que estamos haciendo de la repetición y por la experiencia que tenemos con la aplicación.

Los testers son las primeras personas que están consumiendo una funcionalidad y por tal motivo nos volvemos como los expertos en ello. Entonces son un punto de referencia para el resto del equipo.

✓ **Ayudar a los programadores a replicar los errores y a investigar su solución**

Cuando los testers encuentran defectos, muchas veces cuando los reportan a los programadores, el reporte debe tener un formato donde el programador pueda leer la cantidad de pasos que ejecutar para llegar al error, pero muchas veces estos reportes no son claros o no son obvios o fáciles de entender para el

programador, entonces nosotros tenemos que ir a ayudarles a los programadores a replicar los errores para que ellos puedan corregirlos.

Trabajamos en equipo con ellos y les indicamos paso a paso de manera más detallada de cómo replicar el error fácilmente para que ellos rápidamente lo puedan corregir y nosotros lo podamos probar nuevamente cuando ya esté arreglado

✓ **Implementar prácticas de aseguramiento de calidad para prevenir errores en el código**

Dentro del proceso de desarrollo de software podemos implementar prácticas para prevenir los errores. Por ejemplo, hacer una sugerencia de cómo implementar pruebas unitarias, pruebas a los requisitos, como se está definiendo los requisitos con los analistas de requisitos.

Como nosotros somos expertos en el producto, podríamos implementar sugerencias de al momento de escribir los criterios de aceptación, porque como ya conocemos tanto el producto, podemos ayudarles a los analistas de requisitos, como es que deberían ser las funcionalidades para que estas no sean ambiguas para los desarrolladores difíciles de entender y eventualmente ellos resulten modificando algo que no va a funcionar o que no tiene sentido desde ningún punto de vista.

PROCESO DE PRUEBA DE SOFTWARE





Proceso Fundamental de Testing

Planificación y Control

Análisis y Diseño

Implementación y Ejecución

Evaluación de criterio de salida y
Reportes

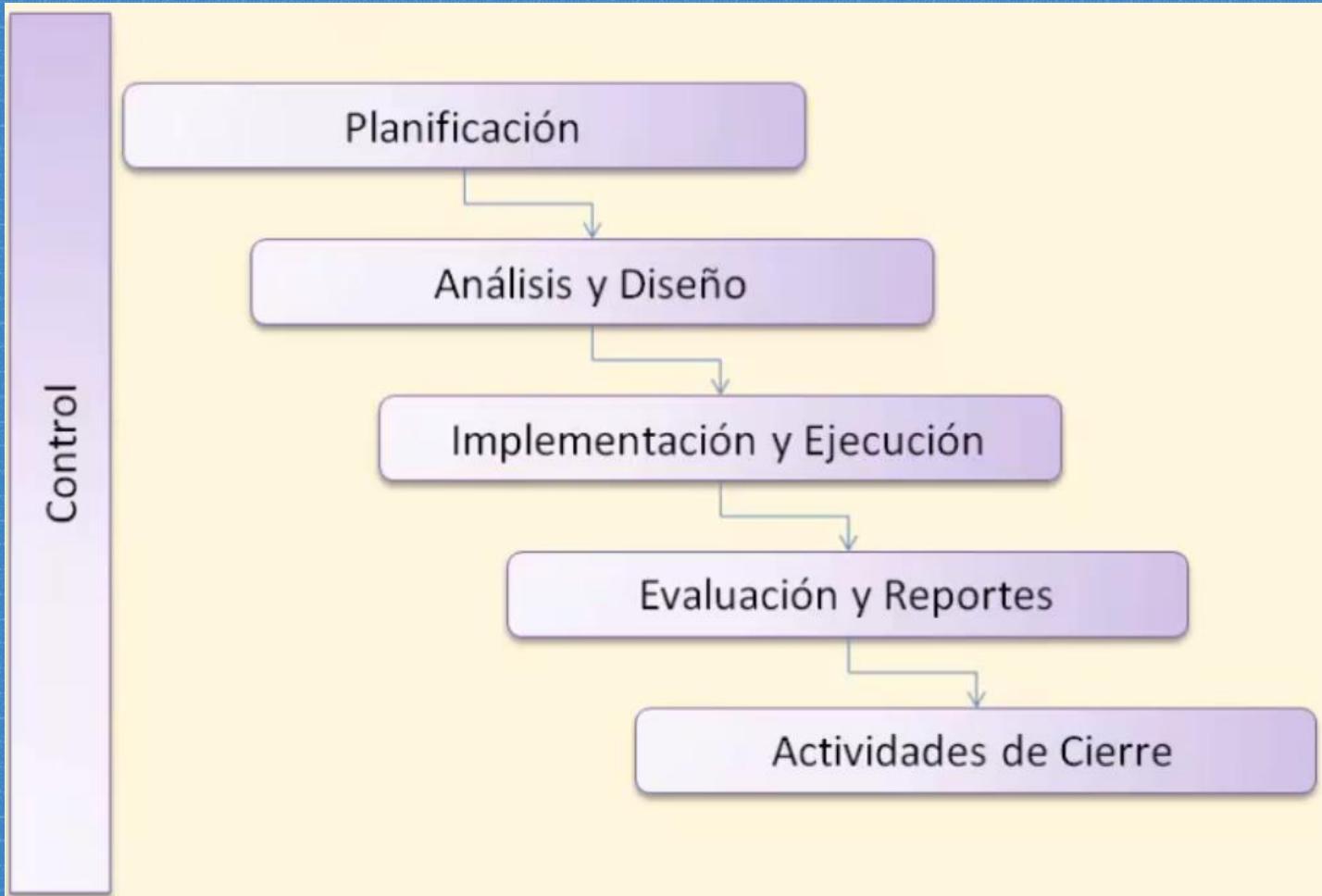
Actividades de cierre



*Testing no sólo es ejecutar pruebas, esta formado por pasos previos y posteriores a la ejecución.
Dependiendo del enfoque, se va a realizar en diferentes puntos del proceso de desarrollo.*



Proceso Fundamental de Testing



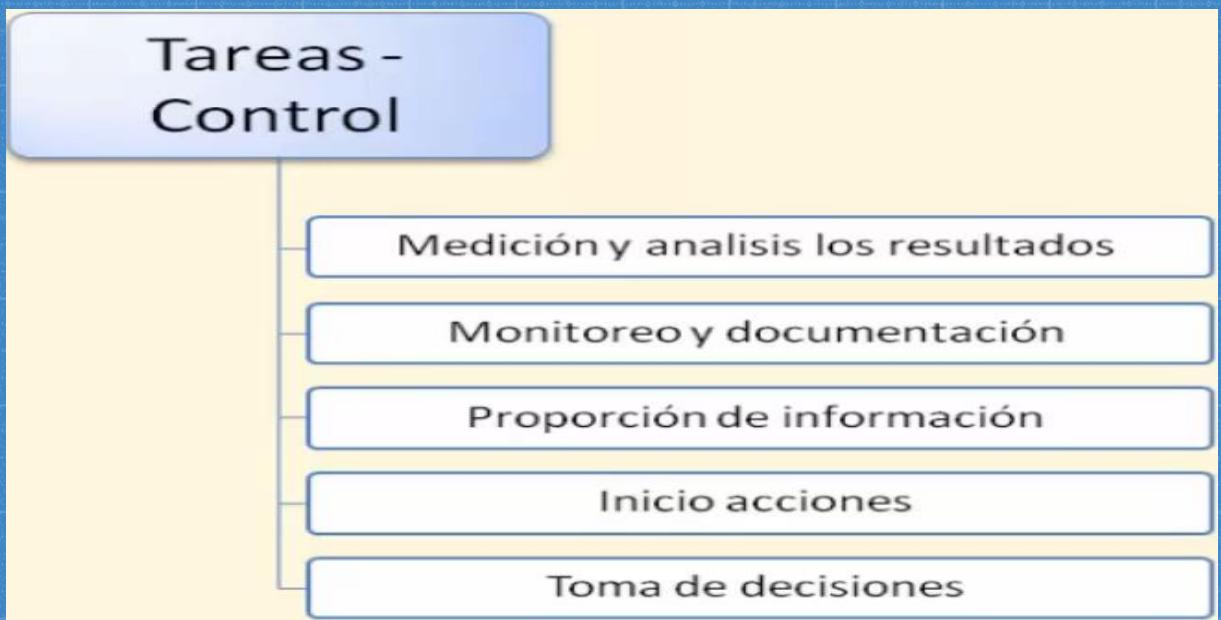
Si bien se tiene diferentes fases, estas pueden superponerse como la fase de control que se realiza en todas las fases.

FASE 1 – Planificación y Control Proceso de Testing



Control de pruebas es la actividad continua porque se realiza durante todas las fases, pero influye específicamente durante la **planificación** de las pruebas.

- El “plan de pruebas maestro” puede ser modificado en función de la información que obtengamos a partir del control de pruebas.
- Para saber el estado del proceso de nuestras pruebas vamos a comparar el progreso logrado con respecto a lo que teníamos planificado.



FASE 1 – Planificación y Control

Proceso de Testing



Tareas - Planificación

Alcance / Riesgos

completo o parte

Objetivos de pruebas

criterios de salidas

Enfoque de pruebas

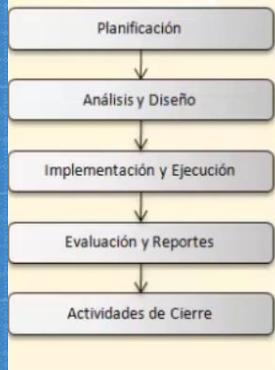
como? que técnica?

Adquisición de recursos

personas, pcs?, etc.

Condiciones de pruebas

entorno estable



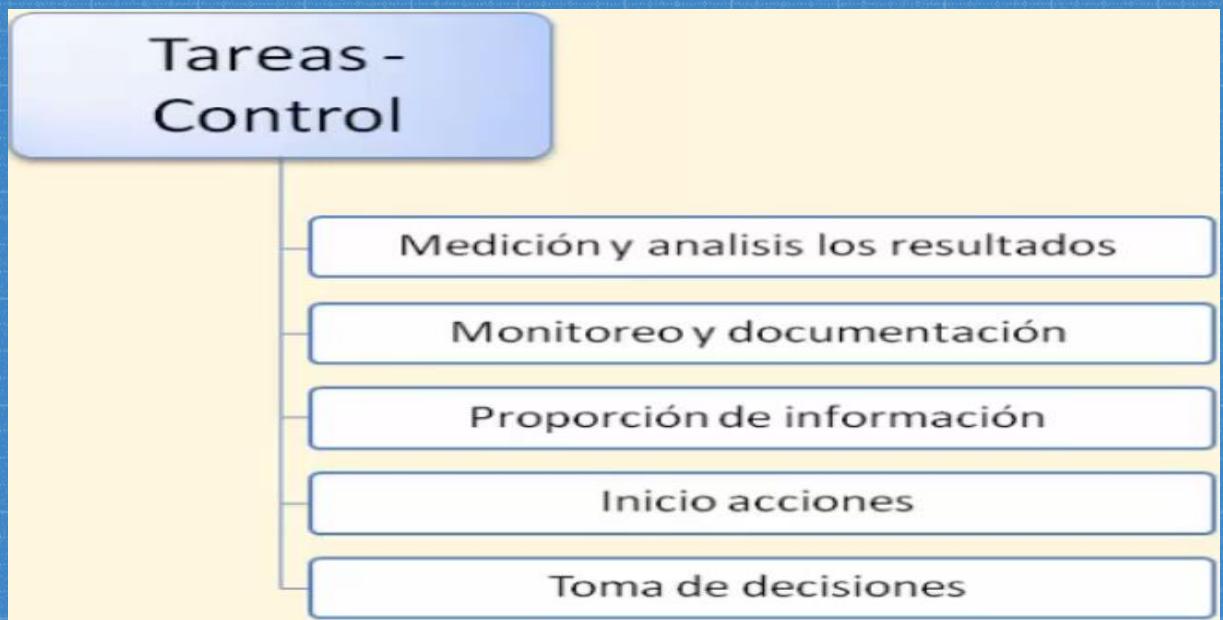
CONTROL

FASE 1 – Planificación y Control Proceso de Testing



Control de pruebas es la actividad continua porque se realiza durante todas las fases, pero influye específicamente durante la **planificación** de las pruebas.

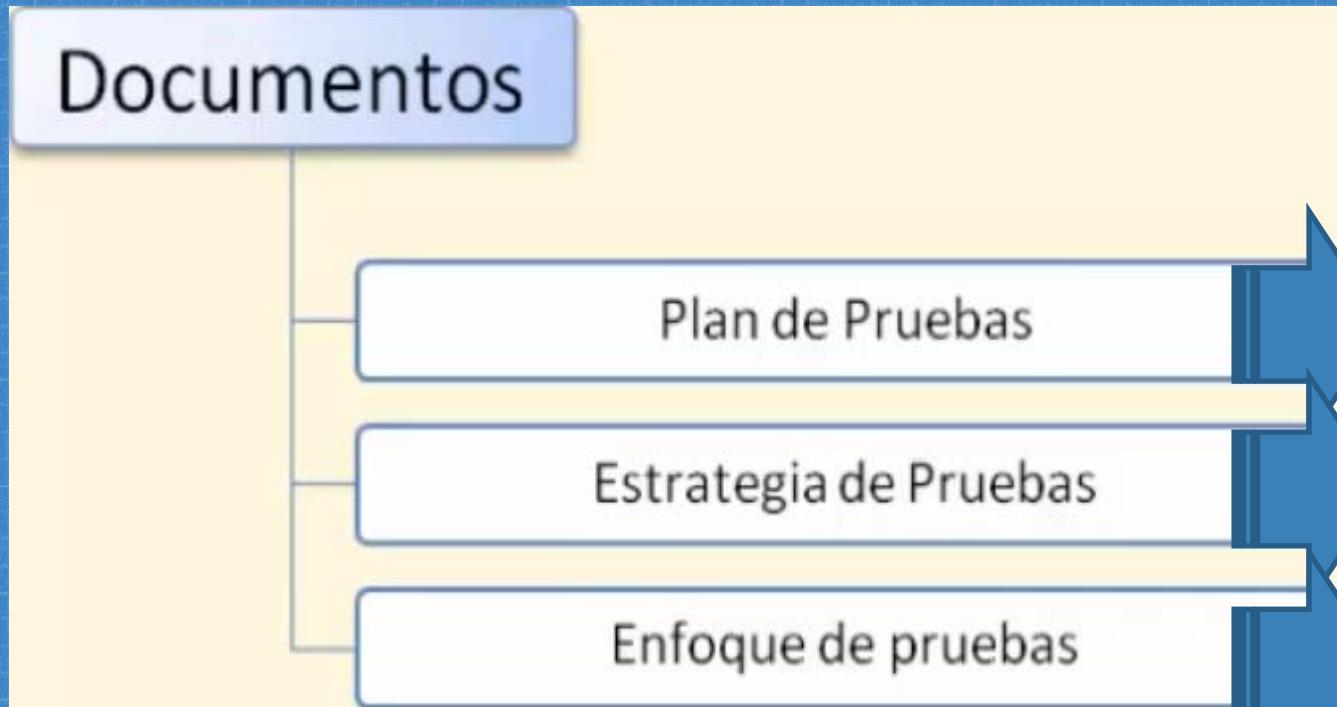
- El “plan de pruebas maestro” puede ser modificado en función de la información que obtengamos a partir del control de pruebas.
- Para saber el estado del proceso de nuestras pruebas vamos a comparar el progreso logrado con respecto a lo que teníamos planificado.



FASE 1 – Planificación y Control Proceso de Testing



Lo que se concluyan en las tareas anteriores se va a plasmar en documentos. Que se van a utilizar en todas las fases del proceso de testing.



Estos criterios de salida son condiciones acordadas con la gente involucrada y nos van a dar las condiciones por las cuales se va a considerar que el proceso esta formalmente concluido.

FASE 2 – Análisis y Diseño

Donde los objetivos generales de las pruebas se transforman en condiciones de pruebas que sean tangibles como son los **casos de prueba**.

Proceso de Testing



Tareas

Revisión de elementos

Testeabilidad

Condiciones de Pruebas

Diseño de Casos

Entorno

Herramientas

Requerimientos, especificaciones

Evaluar elementos basicos

Lista de lo vamos a probar

Positivos - negativos

Que este disponible

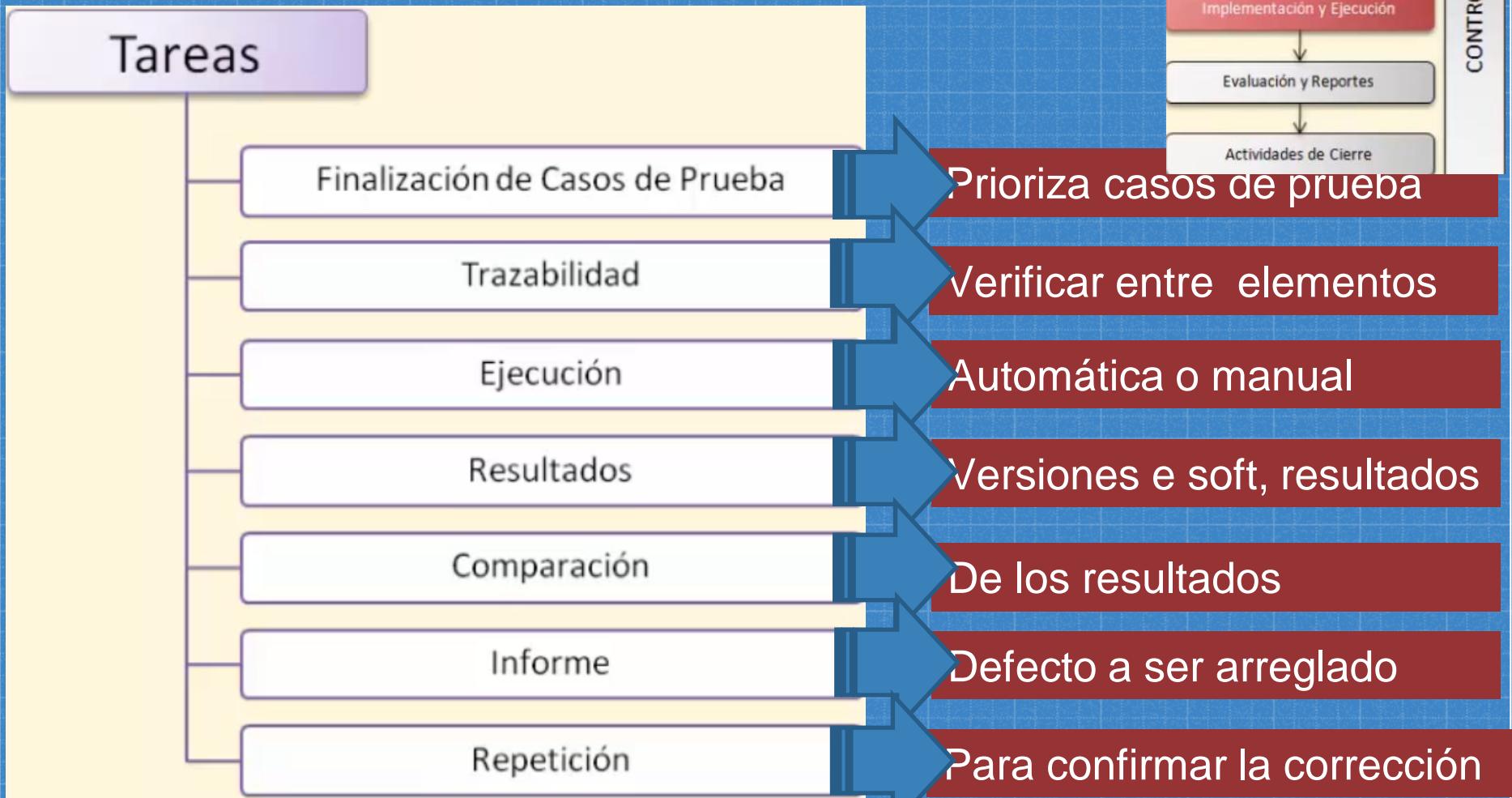
Procesos, procedimientos





FASE 3 – Implementación y Ejecución

La ejecución de las pruebas se la realiza ya sea manualmente o con alguna herramienta.

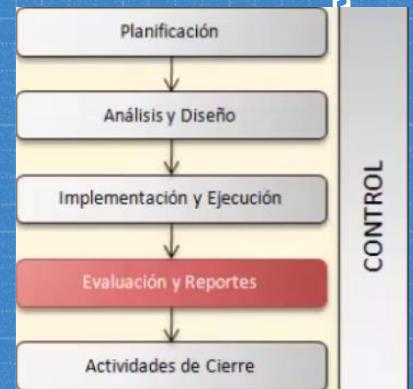


Proceso de Testing



FASE 4–Evaluación de Criterios de Salida y Reportes

La ejecución de las pruebas es evaluado contra los objetivos que se han definido.



Tareas

Comparación

Registros de prueba contra los criterios de salida

Evaluación

Evaluamos si se necesita ejecutar mas pruebas

Reporte

Resumen de las pruebas ejecutadas

Proceso de Testing



FASE 5 – Actividades de Cierre

Nos sirve para concluir con todas nuestras actividades de pruebas.



Tareas

Recolección

Verificación

Documentación

Análisis de lecciones aprendidas

Información de las activ

Todo entregado y prob

Todos corridos

Para futuros proyectos

UNIDAD 2

FUNDAMENTOS DEL TESTING

2.1 INTRODUCCIÓN

Veremos cómo crear planes de prueba, como diseñar los casos de prueba y como reportarlos. Veremos algunos ejemplos donde se va a tener la oportunidad de practicar con sitios web que tenemos en internet de manera que sea más práctico, entretenido e interactivo. Inicialmente recordar que cuando vamos a planificar las pruebas de software, es importante tener en cuenta los principios del software testing.

Este es un tema que se evalúa en la certificación de software testing como lo es FOUNDATION LEVEL DE LA FUNDACION ISTQB que significaría **International Software Testing Cualificación Board** que es como la entidad que certifica software testing a nivel mundial que recomiendo hacerla si se quieren profesionalizar como testers. empezar a ver el detalle cada uno de sus principios con algunos ejemplos.

2.2 PRINCIPIOS DEL SOFTWARE TESTING

2.2.1 Ejecutar pruebas nos muestra la presencia de defectos, pero no puede probar que no los hay. ¿Y qué significa esto?

Aunque se realice una cantidad considerable de pruebas, no es posible asegurar que un sistema se encuentre libre de defectos. Bueno, para la muestra tenemos el Samsung Galaxy Note, que fue liberado por la empresa Samsung en el año 2016 y que tuvo que ser retirado del mercado, porque se incendiaba tanto en reposo como en uso. Entonces, Samsung, siendo una empresa tan importante que tuvo suficiente tiempo para probar este dispositivo tanto en su hardware como en su software, con personas que estoy seguro muy profesionales pudieron haber encontrado los defectos, sin embargo, el dispositivo salió con errores cuando se liberó al mercado.

Entonces, esto es un ejemplo del principio número 1 que las pruebas nos muestran que hubo objetos, pero no pueden probar que no los hay.

2.2.2 El testing exhaustivo es imposible. ¿Qué quiere decir esto?

Probar todas las combinaciones de entrada y precondiciones no es posible, excepto en casos triviales o como muy fáciles, como un formulario con dos campos o más o tres campos. Es muy fácil probarlos, pero imaginémonos que es un formulario con muchos campos, demasiadas las combinaciones que habría que hacer de casos de prueba es casi que imposible, debido a esa situación, lo correcto es realizar un análisis de riesgo y así priorizar y ejecutar los casos de prueba que tienen mayor importancia.

Por ejemplo, el caso de la página de <https://weather.com> en una página que muestra el estado del tiempo a nivel mundial, de todas las ciudades del mundo, y si miramos en el planeta tierra, en el mundo hay 195 países y cada uno de los países tiene muchas ciudades. Si hacemos el cálculo rápido, da más o menos. **quinientas mil ciudades** las que hay en el mundo y nos ponen a probar este software, sería casi imposible probar con cada una de las ciudades cómo se comporta el clima y convivir con sus diferentes combinaciones. Entonces esa prueba sería muy larga, muy costosa.

Por esta razón, en lugar de probar excesivamente esta página del clima es mejor hacer un análisis de riesgos, utilizar algunas técnicas de pruebas y establecer prioridades para enfocar los esfuerzos de las pruebas.

2.2.3 La verificación de la calidad de un sistema debe empezarse lo antes posible. ¿Qué quiere decir esto?

Las pruebas de software deben empezarse lo más temprano posible durante el ciclo de vida del desarrollo del software, **en el control de eventos en etapas tempranas del proyecto**.

Equivale a raíz costo de tiempo y dinero. Mientras más tarde se encuentra defectos, más costoso es arreglarlos, así como la imagen que tenemos. Este bicho representa los bugs, a medida que el **bug** se va encontrando en una etapa posterior el desarrollo del software se va volviendo más costoso.

En el eje X tenemos especificaciones, diseño, codificación, pruebas y la liberación, y a medida que el software va cambiando de etapa se va volviendo más grande y en el eje G tenemos el costo de arreglarlo en el tiempo.

Es mejor encontrarlo durante las especificaciones, durante la negociación de los requisitos del software o al menos durante la codificación, cuando el software, cuando el bug se encuentra durante la etapa de desarrollo, es mucho más barato de corregir.

Hay una investigación muy buena que hizo IBM y dice que el costo de eliminación de un defecto aumenta con el tiempo y, por ejemplo, un defecto encontrado en producción vale 30 veces más que encontrado en etapa de diseño.

2.2.4 La mayoría de efectos relevantes suelen concentrarse en un grupo muy determinado de módulos de nuestro producto y existen distintas razones por la que esto sucede.

Una de ellas son los cambios se hacen regularmente en un solo módulo o en una sola opción del sistema que estás probando. O un módulo o una página o sitio en una aplicación móvil en una de las opciones de la aplicación **puede que sea una opción muy crítica** por ahí en la página web o en la aplicación móvil, o donde se concentra toda la lógica de la aplicación.

Entonces, al realizar una opción muy importante, ahí es donde se hacen la mayoría de cambios y en con la introducción de nuevos cambios hay una probabilidad alta de que se introduzcan errores en la programación. Si miramos un ejemplo, imaginemos que tenemos un sistema de reservas en línea de una aerolínea y los defectos se tienden a encontrar en uno o dos, dos de las opciones dentro de sistema de reservas en línea.

Este fenómeno está relacionado con la regla 80/20, llamado el **principio de Pareto**, que nos indica que el 80 por ciento de los defectos se encuentra en el 20 por ciento del sistema que se está probando por los y por cual, si queremos descubrir una mayor cantidad de bugs, debemos enfocarnos en estos módulos que son más importantes o en un buen análisis de riesgos y en esas funcionalidades que son más importantes. Eso no quiere decir que dejemos de ignorar los demás, pero siguiendo el principio de Pareto, la mayoría de los **bugs** que van a encontrar en esos 20 por ciento el sistema que se está probando.

2.2.5 La paradoja del pesticida

Cuando ejecutan los mismos casos de prueba una y otra vez y sin ningún cambio. Eventualmente estos dejarán de encontrar defectos nuevos, o sea, nuevos bugs. Es necesario que cambiemos las pruebas y los datos de prueba para poder encontrar nuevos defectos.

En la imagen tenemos el software tester en esta persona que está rociando con pesticida sobre un insecto que sería el insecto sería el bug y explica claramente que pasa con el bug y como no le hace ningún efecto el pesticida. Entonces cuando usamos este mismo pesticida a los mismos insectos, estos se vuelven resistentes y no tienen el mismo efecto para detectar uno de los bugs.

2.2.6 El testing es totalmente dependiente del contexto

Es necesario adecuar las pruebas según el contexto del objeto que se está probando. Por ejemplo, no es igual probar una aplicación web para uso médico que maneja datos muy confidenciales a una aplicación móvil de diversión o un sistema bancario, son dos aplicaciones o dos sistemas totalmente distintos, y la gente tiene que adecuarse a ese contexto. El riesgo es un factor crítico a definir para las pruebas. Entonces hay que hacer un análisis de riesgos, pensar que hay más riesgo cuando se trata de vidas humanas, pérdida económica, etc. y por consiguiente, se deben ejecutar más pruebas o unas pruebas con mayor rigor.

2.2.7 La falacia (mentira) de ausencia errores. ¿Qué quiere decir esto?

Es posible que la mayoría de los defectos críticos de una aplicación hayan sido corregidos. Pero esto no asegura que el software sea exitoso. Por ejemplo, podríamos probar todos los requisitos del sistema de una aplicación de comercio electrónico, una página que vende productos que podemos probar todos los casos y de acuerdo a las especificaciones de del negocio.

Digamos que encontramos los errores y los desarrolladores los probaron, los arreglaron, nosotros lo probamos nuevamente y todo ya funciona perfectamente. Aun así, el software que vamos a liberar puede que sea difícil de usar o que en realidad ese sistema sea ineficiente o no se ajuste a la necesidad del mercado o que el sistema tenga una calidad inferior a los sistemas de la competencia.

Entonces eso es un ejemplo claro de la falacia de errores. No quiere decir que porque nosotros proveemos mucho el sistema va a tener alta calidad.

UNIDAD 3

PROCESO DE PRUEBA DE SOFTWARE

Normalmente cuando hablamos de testing pensamos sólo en ejecutar las pruebas, pero en realidad está formado por muchos pasos previos y posteriores a la ejecución de pruebas.

La ejecución de pruebas es solo una parte del proceso de pruebas, dependiendo del enfoque que seleccionemos el proceso de pruebas se va a realizar en diferentes puntos del proceso de desarrollo. Además del proceso de desarrollo tenemos un proceso específico para el testing.

3.1 FASES DEL PROCESO DE PRUEBA

Si bien tenemos diferentes fases y estas fases pueden superponerse como ser la fase de control que se realiza en todas las fases para poder saber en cada momento en que estado nos encontramos. Por lo que vamos a tener: planificación, análisis y diseño, implementación y ejecución, evaluación y reportes y las actividades de cierre.



3.1.1 FASE 1. Planificación y Control, el control de las pruebas es una actividad continua porque se realiza durante todas las demás fases, pero influye específicamente durante la planificación de las pruebas. El “plan de pruebas maestro” puede ser modificado en función de la información que obtengamos a partir del control de pruebas.

Para saber el estado del proceso de nuestras pruebas vamos a comparar el progreso logrado con respecto a lo que teníamos planificado, en base a eso se inicia las medidas correctivas y se preparan y se toman decisiones, entonces la tarea del control va a ser:

- **medir y analizar los resultados de las pruebas**, para saber cuántas fallaron, cuantas pasaron, cantidad, el tipo y la importancia de los defectos que hemos encontrado.
- **monitorear y documentar el progreso**, descubrir cuantas pruebas se completaron y cuantas faltan por completar, cuales son los resultados y cuáles son los riesgos que hemos encontrado.
- **proporcion de información**, brindar información de las pruebas, reportes a las personas interesadas sobre el estado de las pruebas para que tomen las acciones necesarias. Por Ej. Si no hemos tenido el tiempo necesario para realizar todas las pruebas los “Stakeholder” (la parte interesada) pueden decidir que se trabaje horas extras o recortar la cantidad de pruebas que se va a ejecutar.
- **inicio de acciones**, correctivas dependiendo de lo que se necesite corregir es posible que se tenga que ajustar algunos criterios de salida o efectos bloqueantes.
- **toma de decisiones**, se define si se sigue o no con las pruebas.

Tareas de planificación y control

- **Alcance y los Riesgos**, determinar si se va a probar un software completo, un componente o algún otro producto.

-**Objetivos de las pruebas**, identificar estos objetivos y los criterios de salida. Ej. Queremos prevenir defectos, que el software cumple con los requerimientos.

-**Enfoque de pruebas**, ¿Cómo ejecutaremos las pruebas?, ¿qué técnicas vamos a usar? ¿qué es lo que se va probar y cuan extensamente? Es decir: vamos a definir la cobertura de las pruebas, ¿quiénes van a participar en el equipo y durante cuanto tiempo? Se va a establecer nuestra estrategia de pruebas.

-**Adquisición de recursos**, obtener y programar los recursos requeridos por las pruebas. Necesitamos determinar la cantidad de recursos a nivel de personas, computadoras, de software, del entorno de pruebas, del presupuesto. También se debe establecer las fechas para las demás fases del proceso.

-**Condiciones de pruebas**, se seleccionará las condiciones de entrada y salida. Ej. Que el entorno de prueba esté listo y sea estable, que las herramientas necesarias estén instaladas, el equipo de prueba este completo. Ej. Que todos los casos de prueba diseñados sean ejecutados, que todos los documentos estén actualizados.

Una vez que logremos estas “salidas” es que vamos a dejar de probar.

Lo que se concluya en estas tareas se va a plasmar en “**Documentos**” que se va a realizar en esta fase y que se va a utilizar en todas las fases del proceso fundamental de testing.

Por un lado, el documento de “**Plan de pruebas**” describe el alcance, enfoque, los recursos y el calendario de las actividades de prueba que están previstas. Por otro lado, el documento “**Estrategia de pruebas**” describe a alto nivel los niveles de prueba que se van a realizar.

Otro documento importante es el “**Enfoque de pruebas**”, incluye análisis de riesgo, puntos de inicio respecto del proceso de pruebas, técnicas de diseño de pruebas que se va a aplicar, los criterios de salida y los tipos de prueba a ejecutar.

Estos criterios de salida son condiciones acordadas con la gente involucrada y nos van a dar las condiciones por las cuales se va a considerar que el proceso está formalmente concluido.

3.1.2 FASE 2. Análisis y Diseño, donde los objetivos generales de las pruebas se transforman en condiciones de pruebas que sea tangibles como son los casos de prueba.

Las tareas más importantes son:

- **Revisión de elementos básicos para las pruebas**, revisar los requerimientos, las arquitecturas, las especificaciones de diseño, las interfaces, etc. Se usa estas bases para poder comenzar con el diseño de las pruebas.

-**Analizar la testeabilidad**, evaluar si los elementos básicos pueden testearse o permiten generar casos de prueba en base a ellos.

-**Identificación y revisión de las condiciones de pruebas**, basándonos en las tareas anteriores vamos a obtener una lista de lo que nos interesa probar realmente.

-**Diseño de casos de pruebas, positivos**: son aquellos que dan muestra de la funcionalidad en sí, **negativos**: comprueban situaciones en las que no hay tratamientos de errores.

-**Entorno**, involucra poner a punto el entorno de las pruebas. Es decir que esté disponible, que se pueda administrar los usuarios, la carga de datos.

-**Determinar las herramientas necesarias**, ya sea procesos, procedimientos o responsabilidades necesarias. Lo que se hará es seleccionar, proveer, instalar y operar las herramientas de prueba.

3.1.3 FASE 3. Implementación y Ejecución, se realiza la ejecución de las pruebas ya sea manualmente o mediante el uso de alguna herramienta. Las tareas que abarca esta etapa son:

-**Finalización de casos de prueba**, muchas veces no se pudo terminar con el diseño de prueba en la fase anterior, ya sea por lo tiempos o la complejidad, entonces se lo realiza aquí donde se va a finalizar, implementar y priorizar los casos de pruebas y procedimientos de las pruebas.

El procedimiento de prueba consiste en crear los datos de pruebas y los scripts para la automatización en los casos que corresponda. También se va a crear un conjunto de casos pruebas: Ej. Crear lista de productos, logearse, desloguearse. Un conjunto de pruebas lógico sería: logearse, crear una lista de productos y desloguearse. Se están juntando los tres casos de prueba de una manera lógica pero ordenada.

-**Trazabilidad**, verificar que se pueda realizar una trazabilidad entre los elementos básicos de prueba y los casos de prueba. (los elementos básicos los vimos en la etapa de planificación, eran por Ej. Requerimientos, arquitectura, los diseños, los prototipos, etc.) tiene que haber una trazabilidad entre estos elementos básicos y los casos de prueba, a través de uno debo poder llegar al otro.

-**Ejecución**, ya teniendo los casos de prueba, el conjunto de casos de prueba y todo ya definido se ejecuta los casos de manera automática o manual.

-**Registro de los resultados**, una vez que hayamos ejecutado entonces podemos registrar los resultados obtenidos, incluye versiones del software, herramientas de prueba utilizados, los productos de soporte de pruebas, incluye comparar los resultados reales con los resultados esperados.

-**Comparación**, de los resultados. Los actuales obtenidos de las pruebas con los esperados, ósea lo que debería haber sucedido.

-**Informe**, en caso de que haya habido alguna discrepancia reunir los detalles del defecto y los reportamos para que pueda ser arreglado por el desarrollador.

-**Repetición**, repetir las actividades de prueba para confirmar una corrección luego de que el defecto haya sido corregido que hemos informado en el punto anterior (retesting)

3.1.4 FASE 4. Evaluación de Criterios de Salida y Reportes, es la actividad donde la ejecución de las pruebas es evaluada contra los objetivos que hemos definido previamente. Ej. Si tenemos criterios de salida definidos en el plan de pruebas aquí es donde los usaremos comparándolos con los resultados obtenidos actuales. Las tareas que abarca son:

-**Comparación**, los registros de prueba contra los criterios de salida especificados en la fase de planificación. Evaluamos la evidencia que se tiene de las pruebas que se ejecutaron de las fallas reportadas y solucionadas o pendientes para así confirmar si se completaron los criterios de salida que dan fin a las pruebas.

-**Evaluación**, en base a los resultados de la tarea anterior, evaluamos si se necesita ejecutar más pruebas o si el criterio de salida debe modificarse. Ej. que no se haya ejecutado la cobertura deseada o porque nuevos riesgos se han descubierto

-**Reporte**, preparar un resumen de las pruebas ejecutadas para que las partes interesadas estén al tanto del trabajo y el avance.

3.1.5 FASE 5. Actividades de Cierre, nos sirven para concluir con todas nuestras actividades de pruebas. Está compuesto por las siguientes tareas:

-**Recolección de información**, de las actividades de pruebas completadas.

-**Verificación**, que todo haya sido entregado y haya sido probado. Que todo lo indicado en el plan de pruebas se haya cumplido.

-**Documentación**, todos los casos de prueba corridos, todos los resultados, los bugs encontrados, etc.

-**Análisis de lecciones aprendidas**, sirve para futuros proyectos, Ej. mejoras a los procesos del ciclo de desarrollo de software, donde hubo más problemas o fallas, para mejorar e diseño, etc.

3.2 CICLO DE VIDA DE DESARROLLO DE SOFTWARE

El ciclo de vida del desarrollo de software es la estructura que contiene los procesos, actividades y tareas relacionadas con el desarrollo y mantenimiento de un producto de software, abarcando la vida completa del sistema, desde la definición de los requisitos hasta la finalización de su uso.

Se trata de evitar los costes de rectificar errores de implementación mediante un método que permita a los programadores adelantarse para mejorar sus resultados finales.

Este sistema de desarrollo (o ciclo de vida del proceso de software), necesita de varios pasos imprescindibles para garantizar que los programas ofrezcan una buena experiencia al usuario, seguridad, eficiencia, estabilidad y fiabilidad de uso.

El software se construye por medio de la ingeniería de software, en un proceso **multifacético** que conlleva numerosos pasos previos al lanzamiento de un programa. Es por esto que la comunidad del desarrollo de software a nivel mundial ha creado una metodología de trabajo que administren el ciclo de vida de un proyecto.

Los distintos modelos son utilizados para el proceso de desarrollo de software, y también incluyen actividades del proceso de las pruebas.

Significa que las pruebas no existen de manera aislada, siempre están relacionadas con las demás actividades de desarrollo. Dependiendo el modelo que elijamos en nuestro proyecto va a tener un distinto enfoque hacia las pruebas.

Entonces vamos a ver varias.

3.2.1 MODELO DE DESARROLLO EN CASCADA

Se llama el modelo de cascada y el ciclo de desarrollo de software define los pasos involucrados en el desarrollo de software en cada fase. Entonces esto cubre un plan detallado para construir, implementar y mantener el software.

El propósito general de este modelo en cascada es de entregar un producto de alta calidad que cumpla con los requisitos del cliente. Veamos un ejemplo: Amazon.

Imaginemos que Amazon desea desarrollar una funcionalidad de la subasta en su página web. La empresa de desarrollo de software tiene un equipo el cual va a desarrollar esa funcionalidad utilizando el modelo de desarrollo de cascada. (caso hipotético).

El equipo (ellos) empiezan haciendo un análisis de toda la funcionalidad, los analistas de requisitos con unas personas que van y hablan, con las personas de Amazon que desean esta funcionalidad y ellos recogen esos requisitos y los escriben en los documentos.

Entonces hacen todo el análisis, levantan todos los requisitos y de ahí de esta fase sale lo que se llama el documento de diseño.

Este documento se lo pasan a otras personas, que son los arquitectos de software, que son personas con mucho conocimiento técnico, que hacen todo lo que es la arquitectura para esa funcionalidad. Digamos que en esta etapa se demoran dos semanas (en la etapa de análisis se demoran dos semanas y en el diseño otras dos semanas).

El entregable de esta etapa es un documento de arquitectura de software.

Este documento se lo pasan a los programadores, que van a recoger la arquitectura del software, la van a entender y la van a programar con toda la funcionalidad de la de la subasta y la van a hacer en la página web. La implementación se va a demorar por decir cuatro semanas, entonces, durante todo ese mes ellos terminan el software y se la entregan al equipo de testing.

El equipo de testing va a hacer toda la verificación del software, va a comprobar que todo funciona de acuerdo a las especificaciones del documento de diseño y cuando ya ellos dan el visto bueno y van a proceder a pasar el software a producción, lo van a instalar y lo van a poner en producción para su posterior mantenimiento.

- *El modelo de desarrollo de cascada es un proceso muy lento para una funcionalidad como lo que es la subasta.*
- *Amazon tuvo que esperar por lo menos dos o tres meses y ellos tuvieron que esperar mucho tiempo en un escenario optimista.*
- *Demoraron tres meses en entregarle la funcionalidad.*

Desventajas del modelo en cascada

- **No permite cambios de requisitos.**

No les permite porque, si ellos están en el momento de la implementación y a alguien de Amazon le da por hacer un cambio en la funcionalidad de la subasta ya no quieren que se realice con ciertos parámetros sino con algún criterio distinto. Ya cambiarlo dentro no les permite, pues implicaría que se cambie la arquitectura o cambiar documentos de diseño, entonces es muy estricto y el modelo cascado muy rígido, no deja cambiar fácilmente los requisitos porque ya puede estar en una etapa posterior y no va a ser fácil cambiarlo.

- **El producto no es funcional sino hasta que se termina**

Ej. la gente de Amazon solamente va a poder ver el software una vez esté funcionando y este haya sido terminado (al tercer o cuarto mes).

Esta es una gran desventaja porque los ejecutivos van a querer ver rápidamente y poder dar un feedback. Entonces esta es una gran desventaja, solo lo ven al final.

- **Los fallos solo detectan una vez de finalidad el producto**

Para que los testers puedan encontrar los errores tienen que esperar mínimo dos meses a que se termine el análisis, diseño y la implementación. Solo entre el segundo y el tercer mes van a empezar a probarlo.

Entonces les toca esperar mucho tiempo para poder encontrar los fallos.

- **El usuario final no se integra en el proceso de producción hasta que no se terminan la programación**

Los usuarios finales de Amazon, o sea los clientes o inclusive los ejecutivos que son los que están solicitando este requisito de la subasta, solamente pueden proveer o encontrar este feedback o dar como una retroalimentación solo hasta el final. Es decir, les toca esperar todo el tiempo a que la cascada termine.

Entonces este modelo es muy rígido y muy demorado y no es muy recomendado en la actualidad.

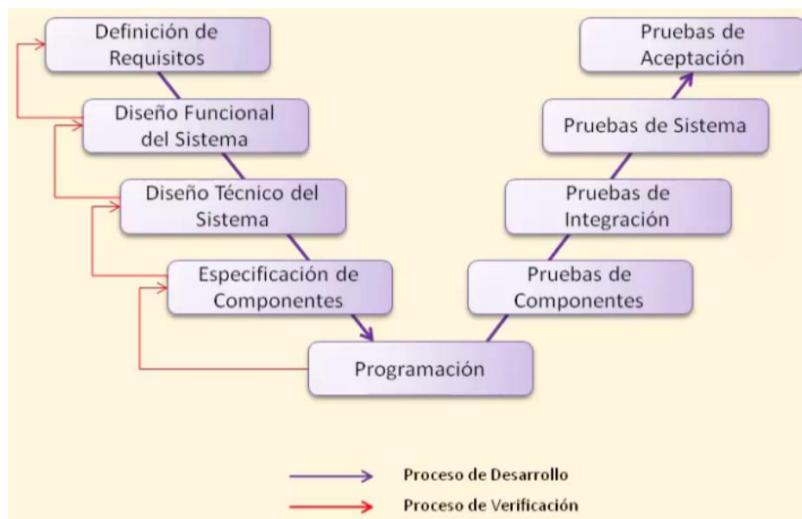
3.2.2 MODELOS DE DESARROLLO V

describe los niveles de desarrollo y niveles de prueba como dos ramas que están relacionadas.

Busca regular el proceso de desarrollo de software y minimizar los riesgos del proyecto en general. Es un proceso ideal por su robustez para pequeños proyectos, es decir para equipos de uno a cinco personas.

El modelo representa las relaciones temporales entre las distintas fases del ciclo de desarrollo en un mismo proyecto, también se lo llama modelo de **Verificación y Validación**, este modelo considera al testing como una actividad paralela al desarrollo de software, por lo tanto, las actividades de testing se realizan en cada una de las fases de proceso de desarrollo. El modelo parece una V, tenemos dos secuencias de fases, la primera se corresponde con la secuencia de fases del desarrollo del proyecto y la segunda con la secuencia de fases del testing del proyecto.

Las fases de un mismo nivel se realizan en paralelo Ej. Especificación de componentes con prueba de componentes.



Modelo V

La rama de a izquierda comienza con la **definición de requisitos**, básicamente revisar toda la documentación con la especificación del cliente. El **diseño funcional** del cliente que es el diseño del flujo funcional del programa, terminado esto realizamos en **diseño técnico** del sistema que básicamente es definir la arquitectura y las interfaces que van a formar parte. Luego la **especificación de componentes** aquí lo que hacemos es especificar la estructura de cada uno de los componentes que van a formar parte de nuestro programa y por último la **programación** que es la creación del código ejecutable en sí.

Cada nivel de desarrollo se verifica respecto de los contenidos del nivel que le precede, para entender esto según la definición, **Verificación** es: comprobar la conformidad de los requisitos establecidos. Nos hacemos la pregunta... ¿lo hemos realizado correctamente?

En la rama derecha, comienza con la **prueba de componentes** que prueba la funcionalidad de cada uno de los componentes, las **pruebas de integración**, lo que hace es probar todas las interfaces de esos sus componentes y como se relacionan, las **pruebas de sistema** una vez que tenemos todo el sistema integrado probamos todo el sistema en si, y las **pruebas de aceptación** que son las pruebas formales de los requisitos que el cliente nos ha dado. En esta rama lo que se hace es **Validar**, que se refiere a la corrección de cada nivel de desarrollo, es decir vamos a comprobar que los resultados de un nivel de desarrollo sean los adecuados. Por ej. en la prueba de componentes **VALIDAMOS** que sean adecuados los resultados de las especificación de componentes, en las pruebas de integración **VALIDAMOS** que sean adecuado el diseño técnico, en las pruebas de sistema **VALIDAMOS** que sean adecuados los resultados del diseño funcional y las pruebas de aceptación **VALIDAMOS** que sean adecuados los requisitos del cliente.

3.2.3 MODELOS ITERATIVOS (Scrum)

-**Modelos Iterativos**, donde los más importantes son: El RUP, el XP y Scrum (metodología).

Los modelos iterativos derivan de lo que es el modelo en cascada, por lo tanto, busca reducir el riesgo que existe entre lo que el usuario realmente necesita y el producto final.

¿Qué es una iteración? Es una secuencia de actividades que se organiza con el objetivo de entregar parte de la funcionalidad de un producto al cliente, es decir en una primera iteración voy a entregar al cliente parte de una funcionalidad que él me pidió, en la segunda iteración voy a entregar una mejora ya sea de la misma funcionalidad que se le entrego antes o una nueva funcionalidad, y así sucesivamente hasta que se logra el producto final.

El cliente una vez que esté terminado cada una de las iteraciones va a evaluar el producto que se le entrega para poder proponer mejoras.

Las actividades de: Definición de requisitos, de Análisis, de Diseño, de Desarrollo, de Prueba y de Implementación se van a segmentar en pasos más chicos pero reducidos y se van a ejecutar de manera continua, en cada iteración se le está agregando una característica adicional al sistema que se está desarrollando, y tenemos otro tema importante en cada una de las iteraciones vamos a aplicar lo mismo que se aplicó en el modelo V, la verificación y la validación.

- **Verificación:** relación con el nivel precedente
- **Validación:** el grado de corrección de producto en el nivel actual

Algunos de los modelos iterativos que se están utilizando son:

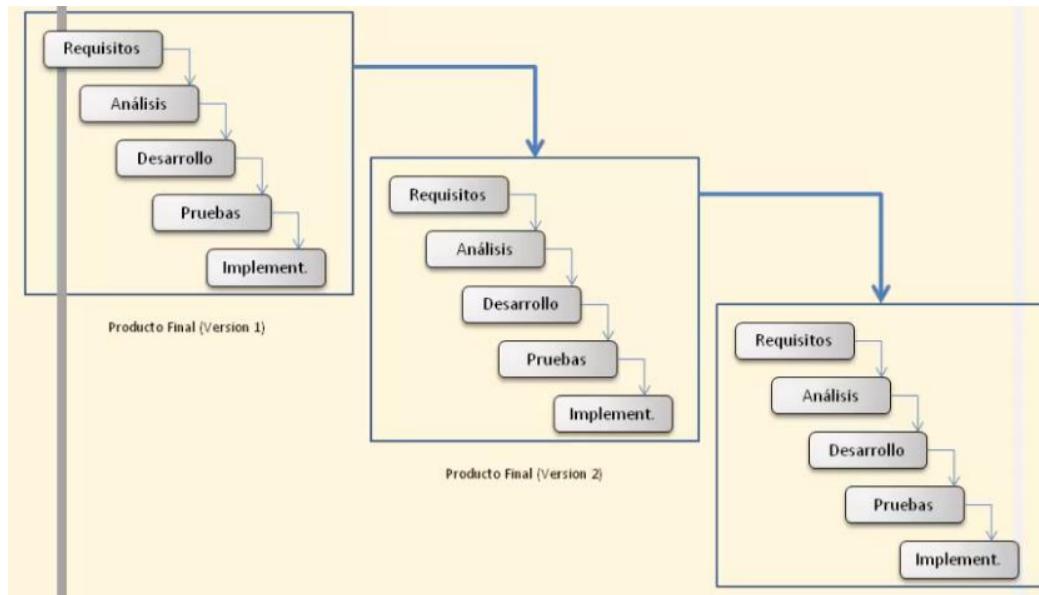


Proceso Unificado - RUP. - Modelo orientado a objetos y pertenece a Rational/IBM, se basa en que aporta un lenguaje de modelado especial llamado UML y da soporte a todo el proceso unificado.

Programación Extrema – XP. - Es un modelo en el cual el desarrollo y las pruebas tienen lugar sin una especificación de requisitos formalizada, todo se define en las reuniones, no hace falta ningún tipo de documentación en específico.

Scrum. - Es un proceso en el que se aplica de manera regular un conjunto de buenas prácticas para poder trabajar en equipo y realizar el mejor producto posible del proyecto en el que se encuentren trabajando. En esta metodología ágil se realizan entregas parciales de lo que es el producto final y se lo utiliza mucho en proyectos que tienen entornos muy complejos o donde se necesite tener resultados pronto para el cliente o donde los requisitos son constantemente cambiantes o poco definidos.

Esta metodología es la que actualmente se está usando en la mayoría de las empresas a nivel mundial, básicamente tenemos un proyecto que se ejecuta en iteraciones que hemos definido previamente, estas iteraciones van a durar desde los 15 días hasta 1 mes (no más de eso), y cada de estas iteraciones va a proporcionar un resultado completo es decir un incremento del producto final.



Modelos Iterativos

Se toma como inicio una lista de requerimientos el cual ha sido priorizado, llamada “**product backlog**” y es el cliente quien va definiendo esta priorización de acuerdo a lo que aportaría o cree que aportaría al producto final.

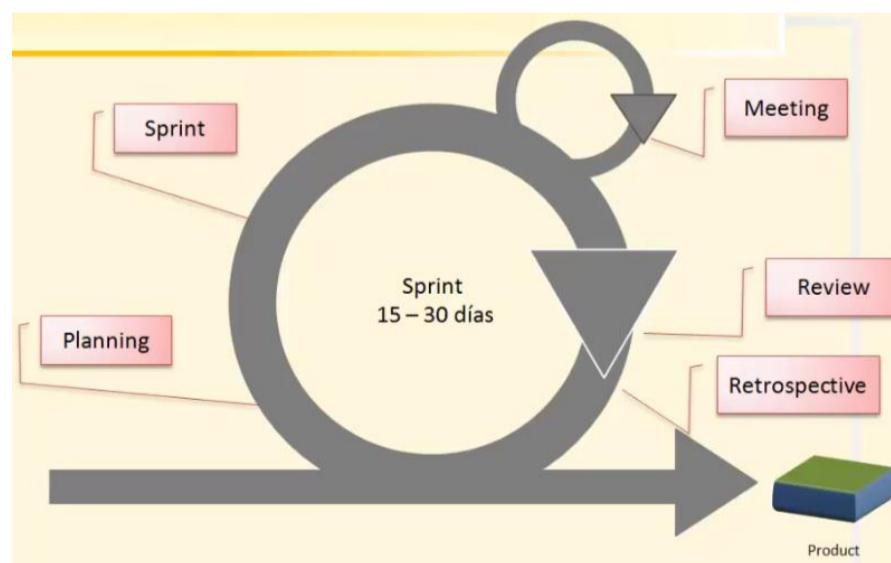
Entonces, todo empieza con el “**spring planning**”, es la reunión que se realiza en primer día de la iteración y tiene dos partes importantes: la selección de los requisitos que se van a implementar en esta iteración y la planificación propiamente dicha.

La selección de requisitos se realiza con la lista de requerimientos y así se seleccionan los que se van a implementar en esta iteración, y la otra parte importante, **la planificación** se basa en la elaboración de la lista de tareas de cada uno para la iteración, aquí es donde se realizan las estimaciones y la asignación de los recursos necesarios que se van a utilizar.

Como segundo paso, tenemos el **spring**, que vendría a ser *la ejecución*, a partir de este momento es que ya se comienza a trabajar en lo que se había planificado el primer día, y todos los días donde todo el equipo se toma 15 minutos para reunirse y así realizar entre todos una de sincronización, llamada “**scrum meeting**”, se realiza para que cada miembro del equipo sepa en qué está trabajando los demás miembros, cada uno va a responder a **tres preguntas**: Que hice desde la última reunión, Que voy a hacer a partir de este momento y Que impedimentos tengo o voy a tener para poder cumplir con mis asignaciones.

El último día se enfrenta lo que se denomina “**Spring review**” es donde el equipo presenta al cliente los requisitos que se han completado en esta iteración, así el cliente puede exponer sus comentarios si le gusto o no, que cambiaria y que no, etc.

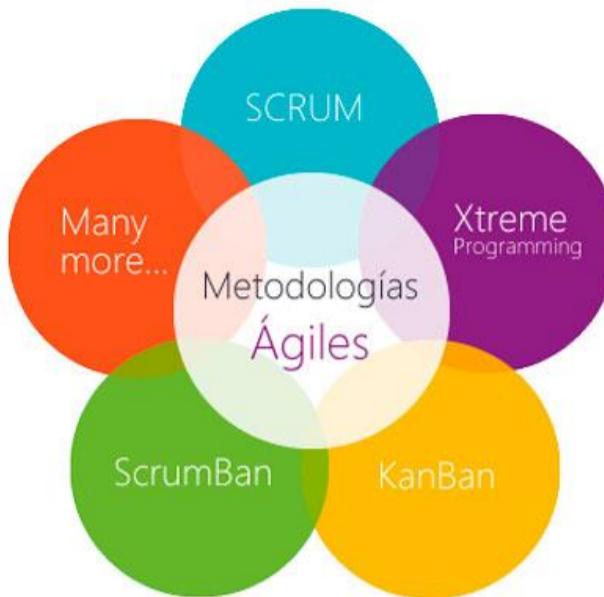
Por último, se realiza la “**retrospective**” el cual permite al equipo analizar cómo fue su trabajo y cuáles son los problemas que podrían impedirle progresar en las próximas iteraciones. Es decir que teniendo esto podemos realizar los cambios necesarios para que no ocurran.



Modelo iterativo - Scrum

Básicamente **Scrum** es una metodología ágil donde tenemos una **Planning** para comenzar y planificar nuestros 15-30 días de iteración, una **Meeting**, todos los días para que cada uno exponga en que está trabajando, una **Spring Review** donde se participa al cliente para que exponga sus opiniones y por ultimo una **Retrospective** para mejorar constantemente.

3.3 METODOLOGIAS DE DESARROLLO AGIL

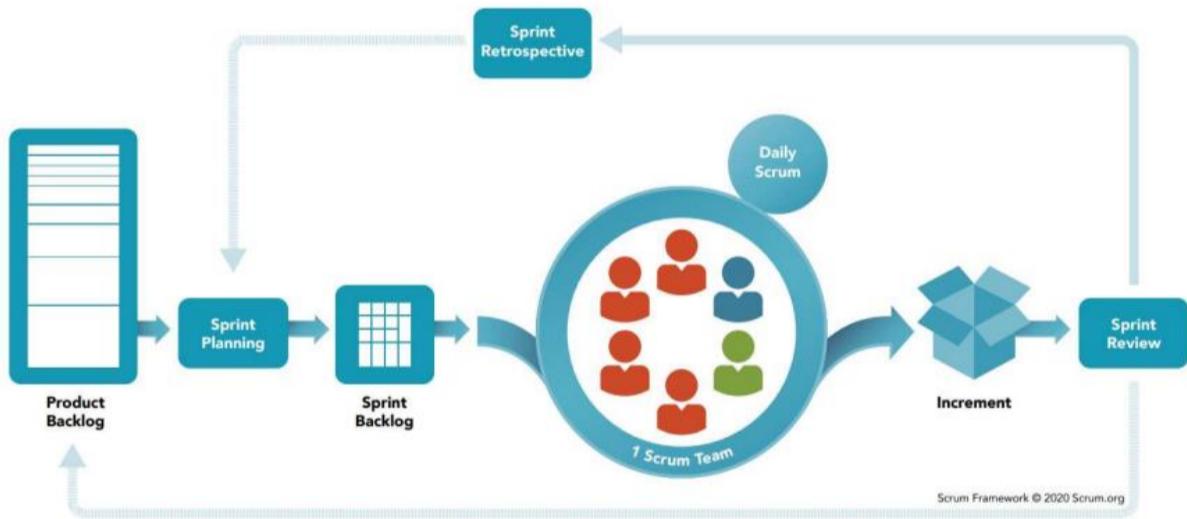


¿Qué es una metodología ágil en el desarrollo?

Por definición, las metodologías ágiles son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno.

Una de las más famosas que se llama Scrum, aunque hay varias.

3.3.1 Scrum



Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente en equipo y obtener el mejor resultado posible de un proyecto. En el desarrollo de la aplicación o a lo que se esté haciendo se divide en pequeñas iteraciones que se llaman sprint.

Esto es una mejor metodología de desarrollo debido a su planificación. Prueba, integración, evaluación de riesgos y control continuo del proceso del proyecto y por lo tanto reduce la posibilidad de falla del mismo.

Para explicar un poco, aunque Scrum sería motivo de un curso completo, pero lo veremos de una manera muy práctica.

Cuando un equipo de desarrollo está trabajando con este marco de trabajo y el equipo lo hace por medio de Springs. **¿Qué es un Sprint?**, es un periodo de tiempo que puede ser de dos o de cuatro semanas, donde el equipo con diferentes roles dentro del equipo de desarrolladores, de análisis de requisitos, dueño de producto, arquitectura, etc. se compromete a entregar la funcionalidad en cierto tiempo, o sea, en el periodo del **sprint** que puede ser de dos semanas o cuatro semanas.

Durante ese mes se va a celebrar ciertas reuniones y utilizando las mejores prácticas de desarrollo de software donde van a comprometerse con un objetivo claro y al finalizar ese sprint entregar esta aplicación esta funcionalidad que están pidiendo.

Si miramos el ejemplo de Amazon nuevamente con la necesidad que tienen de implementar la subasta, con el método en cascada se demoraban entre tres o cuatro meses, pero con Scrum este mismo equipo que va a trabajar en esta funcionalidad va a demorar un sprint, o sea, un mes. **Entonces a eso se refiere con agilidad.**

El equipo posiblemente no va a entregar toda la funcionalidad completa de la subasta en la página, sino que va a implementar una primera versión que sea funcional, que le sirva a Amazon para poder desde ahí empezar a mejorarla, mejorarla en los siguientes sprints, o sea, en el mes dos y en el mes tres, pero ya entregara algo en el mes uno.

3.3.2 Ventajas de manejar Scrum

- **Hay mucha flexibilidad y adaptación a un mercado cambiante**

¿Por qué? Ej. de Amazon. Si deciden durante el primer mes de desarrollo ya no quieren implementar la subasta en el lado derecho de las páginas sino en el lado izquierdo. Entonces lo pueden hacer, pero el equipo puede ir trabajando en la funcionalidad, pero ya para el mes siguiente primero lo van a poner en el lado derecho, que fue donde lo vieron inicialmente, además, si ellos deciden cambiar algunas reglas de negocio, la función de los colores de algún aspecto que quieran cambiar la funcionalidad. Por lo que no tienen que esperar hasta el final final, como en el modelo en cascada hasta que pase los 3 o 4 meses. Para hacer este cambio lo pueden hacer en cada sprint.

- **Resultados anticipados**

Amazon va a tener resultados desde el primer mes de cómo se ve la funcionalidad y lo van a tener de manera anticipada, lo que constituye muchas ventajas para los clientes y también para la empresa. Van a poder tener un **feedback** más rápido, van a poder empezar a evaluar qué tan buena es la opción, que tan bueno el comportamiento de los clientes con la funcionalidad, Que tanto la usan, si la usan más con un browser específico como por ejemplo Chrome, Firefox o si lo usan más en sus celulares como Android, es así que van a tener resultados anticipados porque desde el primer mes van a tener esta información.

- **Obtención de un mínimo producto mínimo (pero) viable**

Cuando nos referimos a producto es a la necesidad como tal, que puede ser la pantalla o la aplicación web o la página web. Todo esto es un producto en desarrollo de software. El producto se refiere al sistema que se está desarrollando para nuestro ejemplo sería a la funcionalidad de la subasta en la página de Amazon.

O sea que es una pequeña versión de la subasta que sea funcional y a medida que va pasando los meses se va a ir mejorando y mejorando. A eso se le llama iterar, mejorar y mejorar la primera versión con una versión mejor, con una versión mejor.

- **Feedbacks rápidos y precisos**

- **Fecha de entrega de proyecto realista**

Si se tiene un sprint, con el que se está trabajando, entonces el equipo va a tener máximo un mes y en ese mes van a ver y tener esa primera versión lista.

- **Rápido aprendizaje y autonomía responsabilidad**

Son cosas que el equipo como tal, con sus roles, van a estar trabajando paralelamente y van a estar aprendiendo de lo que están desarrollando. Van a estar muy unidos, trabajando muy, muy bien como equipo, utilizando las mejores prácticas de desarrollo de software.

Entre ellos van aprendiendo y van a tener mucha autonomía y responsabilidad, con esto nos referimos, con autonomía que muchas veces para cuando se está trabajando durante el proceso del desarrollo, ciertas personas del negocio como ser, por ejemplo, el presidente o el cliente de tecnologías de Amazon decide hacer algún cambio sobre las funcionalidades de lo que se está desarrollando y lo hace a medio del camino, Entonces, si el cambio que se está solicitando es durante el sprint, no se puede, el equipo no se puede desviar del camino, tienes que esperar hasta la siguiente iteración para implementar estos cambios, esta es una autonomía es que equipo debe tener, ya que debe concentrarse en lo que se está haciendo durante ese primer sprint y hacer cambios a medias no se puede.

Eso pasa mucho cuando se está desarrollando software, que la gente o diferentes actores empiezan a querer meter cambios lo que va a provocar cambiar el curso de lo que se está desarrollando. Pero eso no se puede hacer siempre y cuando se esté trabajando sobre este primer mes.

Los cambios, si se quieren hacer cambios, van a tener que ser introducidos para el siguiente sprint por medio de un acto muy importante a que te llama el **dueño del producto**.

3.3.3 Roles en Scrum

Hay ciertos roles que tienen cada uno de los miembros del equipo. Un equipo típico Scrum está compuesto por entre 3 y 9 personas de acuerdo a la teoría de Scrum, y cada uno en este grupo trabajan juntas, normalmente están dentro de la empresa o están trabajando en el mismo espacio y trabajan juntas durante todo el tiempo que dure toda la duración del sprint y tienen unos roles muy específicos.

- El Scrum Master

El Scrum Master, es la persona experta que ayuda al equipo a trabajar con el marco Scrum, es un facilitador y ayuda a resolver los impedimentos. No es un **Project Manager**, es la persona experta en el manejo de Scrum es la persona que guía a todo el equipo de desarrollo y también guía al dueño del producto a ejecutar todas las ceremonias de todos los eventos que deben de hacerse durante el sprint, o sea, durante ese primer mes.

Facilitador también es la persona que le ayuda al equipo a resolver impedimentos cuando ellos tienen algún problema técnico o algún problema con el negocio. El **scrum master** es el que aboga por el equipo y también ayuda a que todos los impedimentos se vayan resolviendo rápidamente para que el equipo pueda lograr el objetivo de finalizar ese sprint.

- El dueño del producto

PO, en general el dueño del producto es una persona que representa la voz del cliente o el negocio dentro del equipo de desarrollo. Es el encargado de identificar qué se debe hacer y controlar el flujo del trabajo. Esta persona es un intermediario dentro del equipo y dentro de la organización.

Si vamos al ejemplo de Amazon, el dueño del producto es una persona que está dentro del equipo Scrum, la persona que sabe muy bien que es lo que Amazon les está pidiendo. Es él que va y habla con los gerentes de tecnología o con los ejecutivos, gerentes, etc. de Amazon.

Recoge esa necesidad que el negocio necesita y se la lleva al equipo de Scrum para que el equipo lo pueda desarrollar, o sea decide **QUE se debe de hacer el producto**. Y ya el equipo se encarga de decir el **COMO lo van a realizar**, (el equipo se encarga de desarrollar el cómo de la funcionalidad). Entonces, en este caso, el PO es la primera persona que recibe esa necesidad en nuestro caso, implementar la subasta en la página web. A ellos, o a él, es al que le dicen primero qué es lo que necesita. El habla con gente del negocio de Amazon y él es el que decide qué es lo que necesita y baja al equipo de Scrum.

Entonces es un rol bastante importante, pero hay todavía más. Otro de los roles más importantes es el equipo de desarrollo como tal.

- El equipo de desarrollo

Son las personas que van a hacer **el cómo**, son las personas que van a desarrollar la funcionalidad de la subasta por ej.

Dentro del equipo de desarrollo hay varios subroles, hay desarrolladores, hay testers, hay analista de requisitos o analista de negocio, hay arquitectos de software. Puede haber también analistas de diseño. Bueno, puede haber varias personas dentro del equipo de desarrollo y en general estas personas son las que van a definitivamente desarrollar la funcionalidad.

Estas personas se van a reunir mediante algunas reuniones que tienen scrum, se van a reunir a discutir la necesidad y van a trabajar conjuntamente en desarrollar todo como si fuera una misión. Ellos son los que van a encargarse de hacer lo que el negocio necesita y ese sería como el objetivo del sprint, o sea, entregar al negocio lo que necesitan en el periodo de tiempo que el negocio lo necesita.

Ahí está el tester o QA, los developers, analista de negocio, arquitectos, desarrolladores está el Scrum Master, que es el que los guía a ellos en todas las prácticas de Scrum y está también el dueño del producto, y por fuera del scrum hay otras personas como el dueño del negocio o a usuarios finales, pero ellos ya están por fuera del equipo. Los más importantes son los que están dentro de la caja gris.

3.3.4 Eventos de Scrum

Se va a tratar de explicar lo más claramente posible mediante el ejemplo de Amazon.

Es importante que entiendan al menos algunos conceptos, o al menos la terminología que se usa para que cuando estén trabajando en empresas no esté como muy dudas.

El corazón del scrum es el sprint en los equipos de trabajo de un modelo Scrum es un periodo de tiempo. Es un ciclo que puede ser de dos o de cuatro semanas.

Recomendablemente y durante ese mes se van a llevar a cabo ciertos eventos o ciertas reuniones.

Entonces, ¿cuáles son estas reuniones?

La primera sería la planeación del sprint. Esa es la primera reunión que se realiza al inicio del sprint y sirve para seleccionar los ítems en los que deba trabajar y como se van a hacer.

Entonces, volviendo al ejemplo de Amazon, con la subasta, el dueño, el producto ya digo que tiene que se necesita una subasta en la página web, una funcionalidad que se ponga al lado del botón de compra donde los usuarios de Amazon puedan bueno, los vendedores puedan subastar artículos y los compradores puedan participar en esa subasta de comprar esos artículos.

Entonces ese es el QUE es desde la necesidad. Eso es lo que se necesita. Esa necesidad está escrita el dueño. El producto escribe todas las necesidades de algo que se llama el **sprint backlog**. Es como una pila de trabajo.

Se escribe todas las necesidades en una plataforma o en cualquier manejador de sprint varios que puede ser JIRA, Trello o puede ser Excel. Las empresas manejan diferentes sistemas para almacenar esas necesidades y lo que hace durante la planeación es que el equipo **se reúne el equipo de trabajo**, o sea el SCRUM TEAM con en PRODUCT OWNER y el SCRUM MASTER. Ellos se reúnen en un sitio y van a trabajar en cómo desarrollar esa funcionalidad y cómo lo van a hacer.

Ahí es donde se define y cuáles son las actividades que se necesitan para llevar a cabo esa funcionalidad, para que al final del Sprint, al final del mes se pueda terminar el producto. Entonces aquí van a estar los desarrolladores, van a decir que van a tener que programar ciertos campos, que van a tener que hacer una a una, por ej. un cuadro de input text o donde van a poner los botones, donde las validaciones para los valores de la subasta, etc. En qué base de datos será la subasta, como la van a conectar, con qué servicios, etc. Los desarrolladores van a escribir sus actividades.

Los testers van a definir también lo que se necesita probar de los diseñadores de interfaces gráficas, quienes van a definir qué se necesita hacer, entonces estas personas van a tener y van a planear cómo se va a hacer eso y lo van a incluir en algo que se llama el sprint backlog.

O sea, van a poner todas las actividades en un tablero que se llama un tablero scrum.

Y ese tablero generalmente lo tienen de manera virtual o también lo tienen físicos y están dentro de la empresa y van a poner desglosar en un conjunto de actividades para poderles hacer un monitoreo y un seguimiento de en **qué, como, en que van, en que etapa** se encuentra cada una de esas actividades

Una vez que se han terminado estas cuatro semanas.

Al finalizar la cuarta semana, el último día del sprint se reúnen para celebrar el **Spring Review** o la revisión, el equipo de desarrollo, el equipo Scrum se reúne con el dueño del producto (PO) también invitan a algunas personas de la empresa, en este caso de Amazon puede ser el gerente de ventas o el gerente de operaciones. Algunas personas importantes de otras empresas que son las que están interesadas en ver cómo está funcionando, lo que ellos solicitaron, lo que ellos le pidieron, En esta reunión van a poder obtener esta información por medio del equipo.

UNA persona del equipo les va a hacer una demostración de la funcionalidad en una computadora proyectando en una pantalla donde van a poder ver, siendo ellos los primeros en ver cómo funciona. Entonces van a tener la oportunidad y van a poder dar un feedback al equipo de algunas correcciones o algunas cosas que se puedan mejorar para la siguiente iteración o ya para el siguiente sprint.

Es una reunión importante donde el equipo toma todas esas observaciones para el **feedback** de las personas del negocio y las van a trabajar en un siguiente sprint.

FINALMENTE SE TIENE La retrospectiva es una reunión que ocurre al final del sprint, donde se hace una reflexión del sprint y se discuten oportunidades de mejora.

Se habla en la retrospectiva de las acciones a tomar para mejora, o para que cuando empiecen el nuevo sprint que será de cero, tengan en cuenta todos esos problemas y se corrijan.

Entonces, al finalizar de estas cuatro semanas, el equipo va a entregar la funcionalidad al negocio. El negocio la va a instalar en producción y los clientes de Amazon van a poder empezar a consumir la funcionalidad de la subasta.

Amazon va a poder obtener información instantánea de lo que ellos solicitaron y van a poder empezarán a dar reportes, cómo crear métricas y mirar a ver que está funcionando y que no está funcionando con la aplicación en internet ya lista.

Las personas del negocio van a darle toda esa información al dueño del producto para que defina cuáles son las mejoras que le van a hacer para que los siguientes sprint implementen y se mejoren.

Esto sería como el resumen de todas las ceremonias.



- ❖ **Planeación Sprint:** Es la primera reunión que se realiza al inicio del sprint, sirve para seleccionar los ítems en los que se va a trabajar y como se van a hacer
- ❖ **Reunión Diaria (Daily):** Es una reunión diaria de 15 minutos en la que cada miembro del equipo de Desarrollo da un update de lo que está haciendo como También que impedimentos tiene
- ❖ **Revisión Sprint:** Es una reunión que ocurre al final del sprint donde el PO y el equipo presentan a los usuarios (stakeholders) el incremento terminado del product para su inspección y adaptación.
- ❖ **Retrospectiva:** Reunión que ocurre al final del sprint donde se hace una reflexión del sprint y se discuten oportunidades de mejora para el próximo sprint

UNIDAD 4

DISEÑO DE LAS PRUEBAS

4.1 Concepto de Caso de Prueba

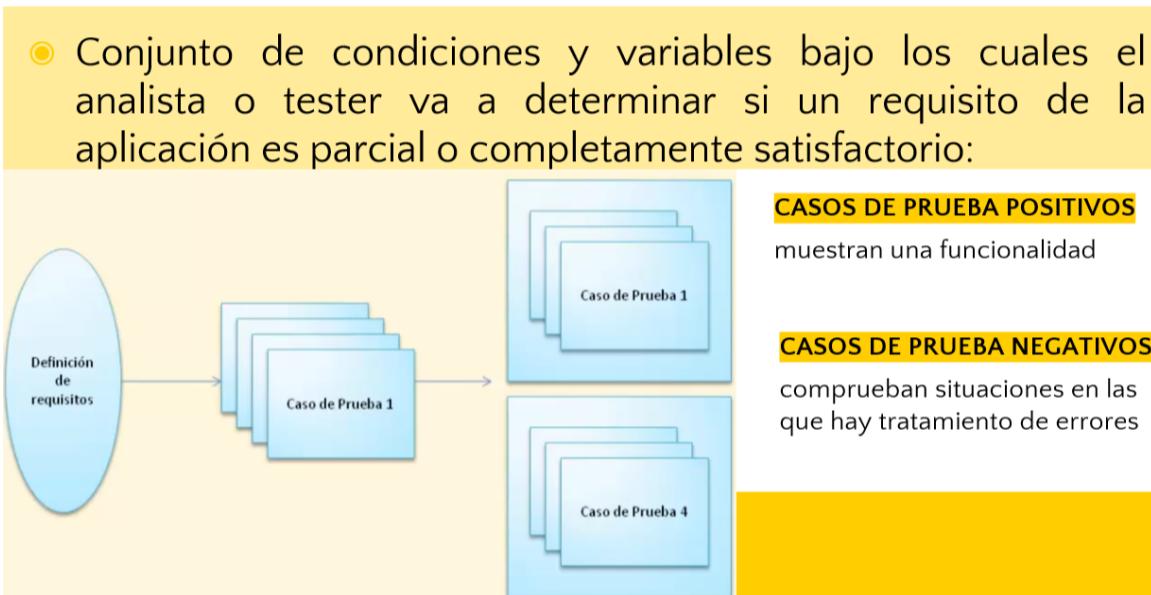
Un caso de prueba es un conjunto de condiciones y variables bajo los cuales el analista o tester va a determinar si un requisito de la aplicación es parcial o completamente satisfactorio. Se va a tener:

-**casos de prueba positivos**, muestran una funcionalidad.

-**casos de prueba negativos**, comprueban situaciones en las que hay tratamiento de errores. Ej. si tengo que mi requisito dice que el precio debe ser mayor que cero, mi caso de prueba positivo será precio igual a cero, a diez, a quince, etc. Y mi caso de prueba será negativo si el precio igual a menos cinco, en este caso el sistema debe poder determinar que es una situación de error y enviar el error a la aplicación.

Un requisito puede tener varios casos de prueba, Ej. precio igual a cero, a diez, a quince, a menos cinco. (al menos va a tener un caso de prueba) ósea hay una trazabilidad. Un requisito es uno o más casos de prueba. La característica más importante es que vamos a tener una entrada conocida y una salida esperada, Ej. el precio igual diez, que sería la entrada. La salida esperada es que permita continuar con la aplicación. Deben ser trazables la trazabilidad es una de las características más importantes entre los requisitos y los casos de prueba.

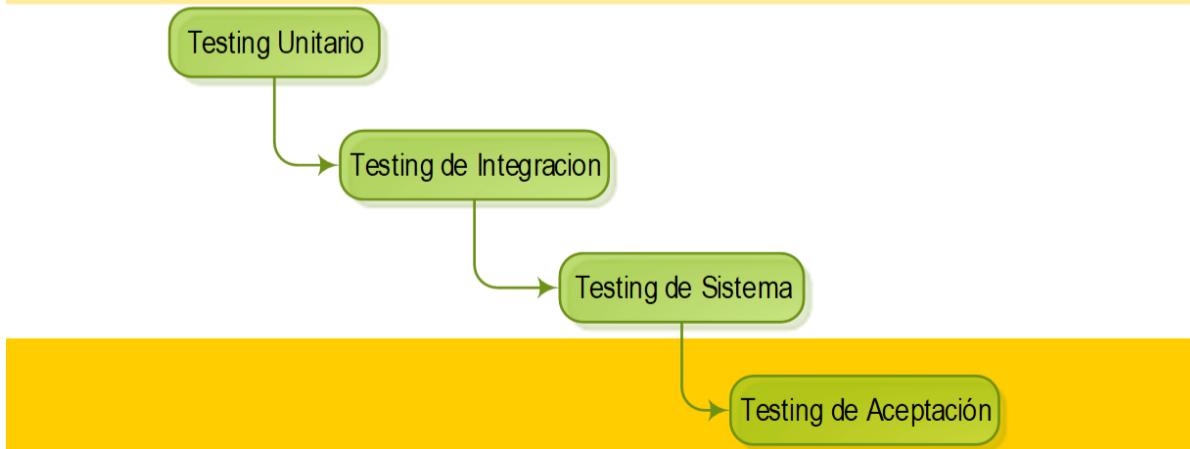
En nuestro caso precio igual a menos cinco, que sería la entrada y la salida un error.



4.2 Niveles de prueba

Dependiendo en qué momento del proceso de desarrollo son ejecutadas las pruebas vamos a tener distintos niveles de test.

- Dependiendo en que momento del proceso de desarrollo son ejecutadas las pruebas vamos a tener distintos niveles de test:



-**test unitario**, o pruebas de componentes, este nivel de pruebas es el más bajo y el primero que se realiza. Es la **prueba de cada componente después de su construcción** de manera individual e independiente. Es decir el desarrollador termina de construir este componente y es ahí donde se desarrolla las pruebas unitarias. Los componentes también pueden ser mencionados como módulos, clases o unidades de acuerdo al lenguaje que se haya usado para el desarrollo del mismo. Es decir, las pruebas unitarias nos permiten asegurar que los pequeños fragmentos de código funcionen bien de manera aislada.

Otra característica es que las pruebas unitarias **incrementan la confianza en el mantenimiento del código**. Si los test son correctos y se corren con cada cambio que se realice en el código, entonces tenemos más probabilidades de que se encuentren los posibles errores en este nivel, y no en un nivel más avanzado, cual es el beneficio, simplemente que el costo de corregir un error en los niveles bajos es mucho menos caro que en los niveles de aceptación, por ejemplo.

Las pruebas unitarias normalmente son **realizadas por los desarrolladores**, pero pueden ser ejecutadas por el equipo de pruebas también. Los casos de prueba se van a obtener a partir de las especificaciones del componente, del diseño del software y del modelo de datos. Nos preguntamos... ¿Se cumple las especificaciones?, vamos a necesitar al menos un caso de prueba que corrobore que la funcionalidad se realiza correctamente. ¿Tenemos resistencia a datos de entrada inválidos? Un defecto muy común en este tipo de pruebas es el procesamiento de datos, ej. valores límite, si el precio tiene que ser mayores que cero, que pasa si es cero.

Se utilizan **métodos de caja blanca**, que ya veremos más adelante. Uno de los frameworks más conocidos que se usa para las pruebas unitarias se llama JUNIT.

-**test de integración**, o de interfaz, es la prueba que comprueba la interacción entre los elementos del software, es decir **comprueba la interacción entre los componentes** una vez que han sido integrados. (integración significa construir grupos de componentes). Cada componente ya ha sido probado en su funcionalidad interna, **se ha realizado el test unitario** o prueba de componente para ese componente y ahora estas pruebas se encargan de comprobar la función externa. Es decir, como interactúan estos componentes entre sí. Puede ser **desarrollado por desarrolladores o por testers** y los casos de prueba van a ser obtenidos a partir de especificación de interfaces, el diseño y el modelo de datos.

El objetivo de las pruebas de integración es básicamente detectar defectos en las interfaces, entre los defectos más comunes son: perdida de datos, manipulación errónea de datos, los componentes interpretan los datos de entrada de manera diferente. Hay varias estrategias para poder abordar este nivel de testing, tenemos: estrategia ascendente o descendente que son las más utilizadas o la bing bang que significa todos los test unitarios se combinan y se prueban.

Para este tipo de pruebas se pueden utilizar los métodos de caja blanca o caja negra, que se van a ver más adelante.

-**test de sistema**, nos referimos al comportamiento de todo el sistema o el producto, que se define en el alcance del proyecto. Ahora se va a probar al sistema como un todo, de manera completa. Probar los requisitos funcionales (adecuación, exactitud, interoperabilidad, cumplimiento de la funcionalidad, seguridad) y los requisitos no funcionales (fiabilidad, usabilidad, eficiencia, mantenibilidad, portabilidad), los casos de prueba pueden ser obtenidos desde la especificación de los requisitos, los procesos de negocio, casos de uso, evaluación de riesgos, todos documentos que tengamos acerca del sistema en sí.

Vamos a probar el sistema desde un punto de vista del usuario, que se hayan implementado correctamente todos los requisitos, (funcionales y no funcionales) el entorno donde estamos probando que sea muy similar con el entorno real.

Se usarán los métodos de caja negra.

-**test de aceptación**, son realizadas para verificar de manera formal la conformidad del sistema con los requisitos. Aquí lo que hacemos es validar que el sistema cumple con el funcionamiento esperado, que está listo para su uso en producción. Básicamente se determina si se cumple los requisitos contractuales (los requisitos que se han firmado con el cliente)

Hay dos tipos de pruebas de aceptación: **testing de aceptación interno**, que es realizado por miembros de la organización donde se ha desarrollado el software; testing de aceptación externo que es realizado por gente externa a donde se ha desarrollado el software, que puede ser el cliente o usuarios finales del sistema.

4.3 Tipos de Prueba

- Funcionales

- No Funcionales
- Estructural
- Relacionada a cambios

2.4.3.1 Pruebas Funcionales, es un tipo de pruebas que se puede realizar en todos los niveles (de prueba vistos anteriormente), el objetivo principal es que el objeto de prueba sistema, programa, etc., funcione realmente.

¿Para que usamos las pruebas funcionales? Sirven para verificar que existen los requisitos funcionales, que son los que están establecidos en las especificaciones, en los casos de uso, en las reglas de negocio, o en cualquier documento que sea relacionado. Se prueba básicamente cinco características que son las más importantes en este tipo de pruebas:

- **Adecuación**, para probar esto nos preguntamos ¿Las funciones implementadas son adecuadas para su uso?
- **Exactitud**, ¿Las funciones presentan los resultados correctos que hemos acordado con el cliente?
- **Interoperabilidad**, ¿La interacción con el entorno del sistema presenta algún problema?
- **Cumplimiento de funcionalidad** ¿El sistema cumple con las normas y reglamentos que se deben aplicar en este caso?
- **Seguridad**, ¿Tenemos los datos o programas protegidos contra acceso no deseado o contra perdidas?

2.4.3.2 Pruebas No Funcionales, el objetivo principal es saber cómo funciona un sistema, se van a describir las pruebas necesarias para medir las características que se puedan cuantificar en una escala. Ej. el tiempo de respuesta para una prueba de rendimiento.

El problema con este tipo de pruebas es que la mayor parte de los proyectos están especificados de manera muy escasa, no están especificados de manera adecuada, y este tipo de pruebas también se puede llevar a cabo en todos los niveles de prueba vistos anteriormente. Dentro de las pruebas no funcionales vamos a tener las pruebas de:

- **Volumen**, el procesamiento de grandes cantidades de datos. ¿Qué pasa cuando mi sistema maneja gran cantidad de datos?
- **Estabilidad**, que sucede cuando el sistema trabaja en un modo de operación continua.
- **Robustez**, es la reacción a entradas erróneas.
- **Usabilidad**, si es estructurado, comprensible, fácil de aprender por el usuario final.
- **Rendimiento**, la rapidez con la cual un sistema ejecuta una determinada función. Ej. si para lograrme necesito 30 segundos, entonces mi rendimiento no es tan bueno.
- **Carga**, hace referencia a que pasa cuando el sistema tiene que trabajar bajo cierta carga, Ej. si tiene muchos usuarios, muchas transacciones cómo reacciona ante eso.

Los Requisitos No Funcionales son muy difíciles de lograr la conformidad porque generalmente no están definidos o no están bien definidos. Ej. el cliente va a decir mi sistema tiene que ser fácil de usar, tiene que tener una interfaz de usuario bien estructurada, pero como logramos eso. ¿Qué es fácil de usar para el usuario?

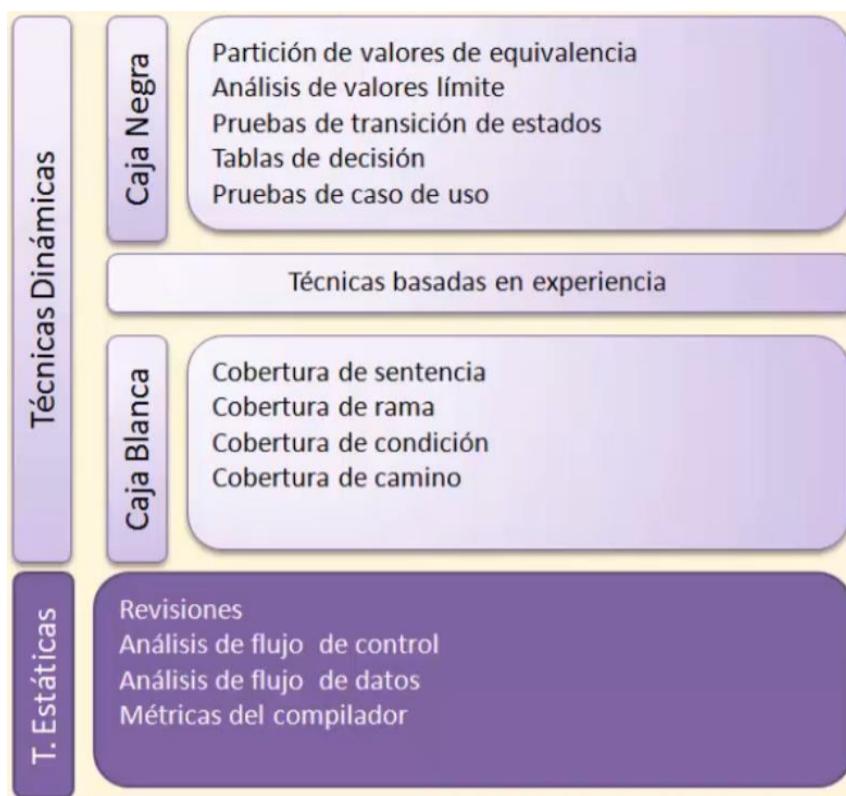
2.4.3.3 Prueba Estructural, que tiene la finalidad de medir el grado en el cual la estructura del objeto de prueba ha sido cubierta por los casos de prueba y el grado en el que ha sido cubierto por los casos de prueba, también denominados pruebas de caja blanca y pueden ser realizados en todos los niveles de prueba, pero sobre todo cuando estamos haciendo las pruebas de componentes (Unit Test) o de integración.

2.4.3.4 Pruebas Relacionadas a Cambios, el objetivo es repetir una prueba de funcionalidad que ha sido verificada previamente, que puede ser de:

- Confirmación, cuando hemos detectado un defecto y los desarrolladores lo han arreglado entonces el software tiene que ser probado de nuevo para comprobar que el defecto original ha sido retirado con éxito, a esto se llama confirmación.
- Regresión, es el conjunto de pruebas repetidas de un programa que ya ha sido probado pero que ha sufrido alguna modificación, y sirve para determinar si se ha introducido un nuevo error.

Es decir, **confirmación** se hace cuando tenemos un bug reparado, tenemos que corroborar que ha sido realmente eliminado y **regresión** es que cuando haya habido un cambio cualquiera sea de entorno o porque se corrigió algún defecto hacemos pruebas repetidas para determinar que no se haya introducido algún error adicional.

4.4 Tipos de Técnicas



Ambas técnicas son complementarias, ya que detectan diferentes tipos de errores, el test estático lo que hace es encontrar defectos y el test dinámico encuentra fallas en el resultado esperado.

Nota. – Repasando lo que se vio anteriormente, fallo era una manifestación física o visible del defecto, era cuando se daba durante la ejecución de la aplicación.

2.5.1 Técnicas Estáticas, es el examen manual, análisis automatizado del código y se basa en cualquier otra documentación del proyecto sin que tengamos que ejecutar el código propiamente dicho.

Normalmente este tipo de técnicas como son: Revisiones, Análisis del Flujo de Control, Análisis del flujo de datos o las métricas del compilador, suelen realizarse antes de las técnicas dinámicas porque los defectos que encontramos con este tipo de técnicas van a ser al principio del Ciclo de Vida, por lo tanto, van a ser menos costosos para corregirlos que los detectados durante la ejecución de las pruebas.

- **Revisiones**, (*una de las formas más utilizadas en las técnicas estáticas*), según el estándar EEE1028 que habla sobre las revisiones en el Software, una “Revisión” es la **Evaluación de un producto para poder detectar discrepancias respecto de los resultados planificados y de esta forma recomendar mejoras**.

Consiste en analizar un determinado objeto de prueba, pero sin tener que ejecutarlo. Ej. un script, un código fuente, un requisito, una especificación, etc. lo que se debe evaluar son los estándares de programación, los flujos de datos, **no el resultado que vamos a obtener luego de ejecutar el código**.

El principal objetivo es que nos permitirá descubrir defectos en las especificaciones, en el diseño, en las especificaciones de interfaces cualquier otro tipo de documento. Las revisiones pueden ser:

- a) **Revisión Formal**, tiene ciertas etapas o actividades que se deben cumplir o completar las cuales son:
 1. **Planificación** donde se deben definir los criterios a utilizar, (que tipo de revisión a aplicar, cual es la lista de comprobación, quienes van a participar, que rol va a tener cada participante, y los tiempos en los cuales se va a realizar).
 2. **Criterios de Entrada/Salida**, que es lo que estamos revisando y cuando vamos a parar de revisar esto.
 3. **Reunión de inicio de proceso y distribución de documentos (Kick-Off)**, donde definen que es lo que debe hacer cada uno.
 4. **Identificación de defectos**, sobre los documentos, armar preguntas, consultas, comentarios, etc.
 5. **Reunión de revisión**, donde se va a discutir los resultados obtenidos, el resultado de los comentarios, la preguntas, etc. y de va a registrar los **resultados** y recomendaciones para mejorar.
 6. **Reconstrucción**, la persona encargada va a corregir los defectos, evaluar las mejoras si hubo y las va a implementar, luego nuevamente una reunión de seguimiento.

7. Comprobación de los criterios de salida, si se cumplen o no, en caso de que no se cumplan se inicia nuevamente con la reunión, etc., hasta que se termine con la revisión.

- b) **Revisión Informal**, (o revisión entre pares), es iniciada por el autor o creador de documento del código, de la especificación, etc. y hay uno o dos desarrolladores colaboradores normalmente se realiza para la revisión de código. La documentación es informal y los resultados pueden ser registrados en forma de una lista de acción o de actividades a realizar.
El objetivo principal de esta revisión es invertir poco para obtener un posible beneficio.
- c) **Revisión Guiada**, es un tipo de revisión liderada por el autor del objeto que se está presentando y a lo largo de la presentación este autor irá mostrando el objeto a revisar, habrá un par de revisores que harán preguntas y comentarios para tratar de comprender el objeto y detectar desviaciones o áreas que representen algún tipo de problema. El objetivo es ganar conocimiento del objeto a probar y entender el funcionamiento y detectar defectos. Ej. diseño preliminar de interfaz de usuarios, modelo de datos, etc.
- d) **Revisión Técnica**, puede ser formal o informal con la diferencia que aquí participan técnicos expertos, preferentemente externos a la organización y además es liderada por otra persona que no es el autor denominada **Moderador**. Ej. chequear estándares, etc.
- e) **Revisión de Inspección**, es la revisión más formal porque contiene proceso de preparación, ejecución, documentación, de seguimiento, tiene roles bien específicos, listas de comprobación ya definidos, se utiliza métricas, criterios de entrada y salida para aceptar o no el producto. Contiene todas las etapas que hemos visto previamente.

Nota. - El propósito principal de esto es detectar defectos utilizando un método estructurado.

2.5.2 Técnicas Dinámicas, estos tipos se diferencian básicamente en la forma en la que se obtienen los casos de prueba, y para tener una **buena cobertura** deberíamos poder aplicar las tres técnicas porque cada una va a descubrir defectos diferentes.

2.5.2.1 Técnicas de Caja Blanca, se realizan cuando el tester puede acceder al código fuente de la aplicación y también a sus algoritmos y estructura de datos.

La persona que está probando necesita tener habilidades de programación, conocer el framework de desarrollo que se está usando, tiene que saber también que es lo que hacen las distintas clases y métodos. Esta técnica se realiza sobre las funciones internas de cada módulo, lo que tiene mayor foco es la estructura del programa (*la parte interna vemos lo que hay adentro sin importar las entradas y salidas*).

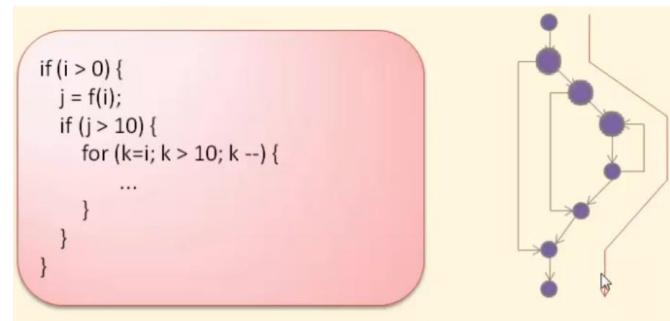
Los casos de prueba aquí van a garantizar que todos los caminos que existen en el módulo hayan sido ejecutados tanto las decisiones positivas como las negativas. El objetivo principal es: *poder alcanzar el código que no se puede alcanzar con caja negra, poder encontrar inconsistencias, código que no se utiliza o no tiene razón de ser, permite optimizar el funcionamiento de la aplicación desde adentro*.

Se puede utilizar tanto en los niveles de componentes, de integración y de sistema.

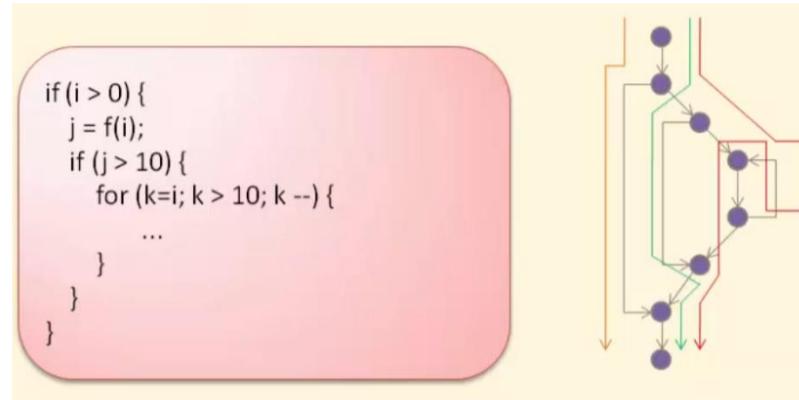
Una de las ventajas es que los métodos pueden aplicarse en etapas tempranas del sistema, y encontrar fallas en etapas tempranas hace que el costo de corregir sea mucho menor a etapas finales, y se tiene una cobertura casi total a lo que es la estructura del sistema.

Tipos de técnicas de caja blanca: ¿Qué es cobertura? Es la medida en que un conjunto de pruebas ha probado una estructura en particular, esta medida va a ser expresada en porcentaje.

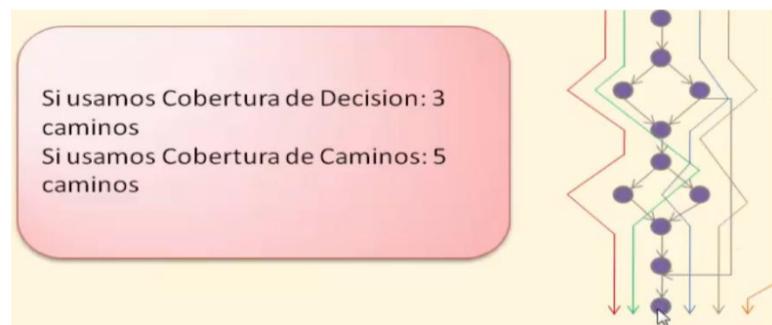
-**Cobertura de sentencia**, comprobamos el número de sentencias ejecutadas, el foco es la sentencia de un programa. → ¿Qué caso de prueba vamos a necesitar? Será satisfactorio si todas las sentencias son ejecutadas por lo menos una vez. Se utiliza un gráfico de flujo de control (las instrucciones o las sentencias se representan mediante nodos, y el flujo de control se representan por las aristas o la flecha), el cálculo de la cobertura es: **Cobertura = (Nro. de Sentencias ejecutadas/Nro. total de sentencias) *100**. El objetivo principal de estas pruebas es detectar aquellos bloques de código que no se utilizan. (código muerto) Ej.



- **Cobertura de Decisión**, comprobamos el número de aristas ejecutadas. (una decisión es un punto en el código en el cual se va a producir una ramificación) entonces vamos a satisfacer el criterio de cobertura de decisión si todas las condiciones del programa son ejecutadas al menos una vez por verdadero o por falso. También utiliza el grafico de flujo de control, pero se centra en el flujo de control del segmento de un programan (en las aristas del diagrama de flujo). Nos preguntamos qué caso de prueba necesito para cubrir por lo menos una vez una de estas aristas. **Cobertura = (Nro. de Decisiones ejecutadas/Nro. total de decisiones) *100.**



- **Cobertura de Camino** (combinación de segmentos de programa, en un diagrama de flujo), es una determinada secuencia de nodos y aristas alternados. Primero comprobar el número de caminos linealmente independientes que se han ejecutado en el diagrama de flujos de la unidad o componente que se está testeando. Nos centramos en la ejecución de todos los posibles caminos que se puedan lograr través del programa. Ej. un camino va a ser una vía única desde el inicio hasta el final del programa del diagrama de flujo. El objetivo es alcanzar un porcentaje definido previamente de cobertura de camino. Cobertura= (Nro. de caminos ejecutados/Nro. total de caminos) *100. La cobertura de camino es mucho más exhaustiva que la cobertura de sentencia y de decisión y no es fácil lograr una cobertura de camino de 100% (solamente se logra en programas que son muy simples).



- **Pruebas de Condición y cobertura**, es una técnica un poco más complicada que no evalúa la sentencia en sí, sino lo que hay dentro de la sentencia, de las condiciones múltiples que existen dentro de las sentencias. Van a ser el porcentaje de todos los resultados individuales de condición que afectan al resultado de la decisión lo que se hace aquí es detectar defectos que provienen de implementar condiciones múltiples (constituidas por combinación de condiciones atómicas) S e combinan mediante el uso de operadores lógicos, OR, AND, etc. Hay tres tipos de técnicas:

a) cobertura de condición simple:

Cobertura de Condición Simple:		Ejemplo: $a > 1 \text{ OR } b < 10$
$a = 6 \text{ (true)}$	$b = 15 \text{ (false)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 0 \text{ (false)}$	$b = 2 \text{ (true)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$

Las subcondiciones deben tomar al menos una vez el valor verdadero y falso.

Combinamos estos valores con dos casos de prueba. Vamos a lograr toda la cobertura porque cada condición atómica o subcondición ha tomado dos valores diferentes (verdadero/falso).

b) cobertura de condición múltiple:

Cobertura de Condición Múltiple:		Ejemplo: $a > 1 \text{ OR } b < 10$
$a = 6 \text{ (true)}$	$b = 15 \text{ (false)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 6 \text{ (true)}$	$b = 5 \text{ (true)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 0 \text{ (false)}$	$b = 5 \text{ (true)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 0 \text{ (false)}$	$b = 15 \text{ (false)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (false)}$

En este caso usaremos todas las combinaciones posibles que tengamos como casos de prueba (siempre y cuando sean reales), será mejor que el anterior porque vamos a cubrir todas las sentencias y decisiones, pero nos vamos a encontrar con que las ejecuciones de algunas decisiones no son posibles.

c) mínima cobertura de condición múltiple:

Mínima Cobertura de Condición Múltiple:		Ejemplo: $a > 1 \text{ OR } b < 10$
$a = 6 \text{ (true)}$	$b = 15 \text{ (false)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 6 \text{ (true)}$	$b = 5 \text{ (true)}$	$a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 0 \text{ (false)}$	$b = 5 \text{ (true)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (true)}$
$a = 0 \text{ (false)}$	$b = 15 \text{ (false)}$	$\rightarrow a > 1 \text{ OR } b < 10 \text{ (false)}$

Se basa en crear la menor cantidad de casos de pruebas necesarias, para lograr se debe va a tomar en cuenta una sola regla:

SOLO SE VA A CONSIDERAR COMO CASO DE PRUEBA, CUANDO EL CAMBIO DEL RESULTADO DE UNA SUBCONDICION (CONDICION ATOMICA), CAMBIA EL RESULTADO TOTAL DE LA CONDICION

COMBINADA.

2.5.2.2 Técnicas de caja negra

Esta estrategia de testing se centra principalmente en la **verificación de la funcionalidad de la aplicación**, vamos a estar viendo los datos que entran, los resultados que se obtienen, la interacción, el funcionamiento de la interfaz de usuario, en general todo aquello que suponga estudiar el correcto funcionamiento del sistema.

Como es impracticable realizar las pruebas sobre todas las posibilidades el objetivo es encontrar una serie de entradas donde el comportamiento permita encontrar la mayor cantidad de errores, con esta técnica vamos a observar al módulo, al programa, al componente como una caja, el sistema es tratado como un sistema cerrado, donde no conocemos nada de cómo fue desarrollado, vamos a trabajar con entradas y salidas.

La principal diferencia entre la técnica de caja negra y caja blanca, en caja negra no se trabaja con el código fuente sino con el programa en sí. **Se debe elegir la técnica que mejor se aadecue al proyecto en el que se está trabajando, la técnica de caja negra es la técnica más cercana a la experiencia del usuario**, el objetivo principal es asegurarnos que el usuario ve cómo funciona de acuerdo a los requerimientos y que además cumple con las expectativas del usuario, *también se le llama test funcional o de comportamiento*.

Es muy importante que las especificaciones del sistema sean de calidad, es decir, como el resultado de las pruebas de caja negra dependen de las especificaciones del sistema, si las especificaciones son erróneas entonces los casos de prueba también lo serán.

Qué tipo de errores vamos a encontrar: errores de interfaz, errores de comportamiento, etc.

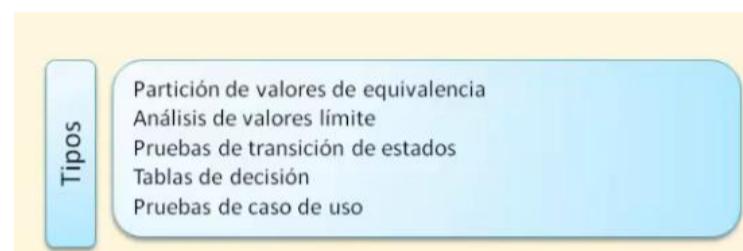
En que niveles puedo utilizar este método, en todos. En el testing unitario, en el testing de integración, y en el testing de aceptación.

Ventajas:

- las pruebas se realizan desde un punto de vista del usuario buscando discrepancias con las especificaciones.
- El tester no necesita conocimiento del lenguaje de programación
- Las pruebas pueden diseñarse a medida que se termina las especificaciones, no hace falta esperar a que el desarrollador termine con el código.

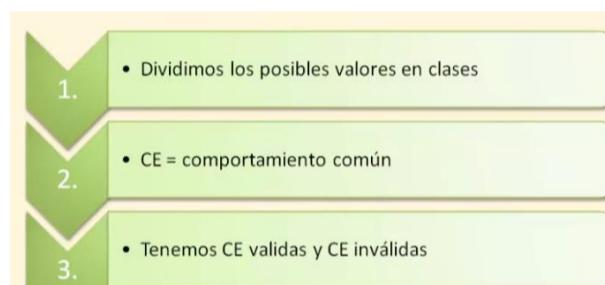
Desventaja. No hay forma de probar todos los caminos posibles en el sistema, si los especificadores no son claras o falta algo, los casos de prueba no van a ser buenos y tendrán falta de cobertura.

Vamos a ver:



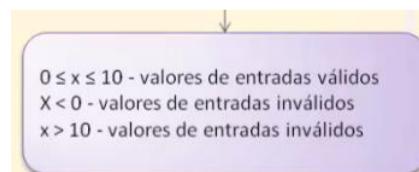
Partición de equivalencias:

Es lo que la mayoría de los tester usamos de manera intuitiva, lo que hacemos es dividir los posibles valores en clases, mediante esto observamos valores de entrada de un programa y valores de salida. La partición de valores de equivalencia se va a dirigir a una definición de casos de prueba que descubran clases de errores, reduciendo así el número total de casos de prueba que hay que desarrollar, **entonces tenemos que con una mínima cantidad de casos de prueba se puede esperar un valor cobertura específico** y además es aplicable a todos los niveles de prueba, de componentes, de integración, de sistemas y de integración. CE=clases de equivalencias



En rango de valores definido que tenemos se va a agrupar en clases de equivalencia, donde todos los valores para los cuales se espera que el programa tenga un comportamiento común se agregan a una determinada clase de equivalencia o CE, y además las clases de equivalencia puede consistir en un rango de valores por ej. que $x < 0$ y $x < 10$ o en un valor aislado por ej. que x sea verdadero, entonces teniendo esto en cuenta que las clases de equivalencias puede ser: clase de equivalencias válidas y clases de equivalencias invalidadas donde nos encontramos con valores que están fuera del rango o con un formato incorrecto.

Tenemos que nuestro valor x está definido que como x es mayor o igual que 0, y es mayor o igual que 10, entonces como clase de equivalencia básica y mínima vamos a tener 3:



Cuando x está dentro del rango entre 0 y 10 van a ser los valores de entrada válidos, como también necesitamos clase de equivalencias inválidas tenemos que x menor a 0 o cuando x es mayor que 10, van a ser valores de entrada invalidadas, ESTAN SERIAN LAS BASICAS, también podemos determinar otro tipo de valores por ej. Que x no sea numérico o que tenga un formato numérico no admitido.

Entonces con estas clases de equivalencias vamos a generar nuestros casos de prueba, para ello armaremos básicamente la siguiente tabla:

Variable	Clase de Equivalencia	Representante
Valor válido	EC1: $0 \leq x \leq 10$	+5
	EC2: $x < 0$	-15
	EC3: $x > 10$	+20
	EC4: x no entero	Hola

Tenemos que para nuestra primera clase de equivalencia, vamos a tener nuestro primero test case, vamos a tener como representante en número +5, para formar un caso de prueba donde $x < 0$ usaremos como representante el -15 y para $x > 10$ usaremos el número +20 y como otra clase de equivalencia aparte nuestro representante será “Hola”.

Como conclusión podemos obtener que a partir de una expresión $0 \leq x \leq 10$ vamos a obtener todas nuestras clases de equivalencias, vamos a determinar rangos de valores donde estos valores se comporten de manera similar, por ej. **1er caso valores de entrada validas**, con $x < 0$ deberían tener un comportamiento similar, lo mismo cuando $x > 10$, y el mismo caso cuando x es “no entero”, a apartir de estas clases de equivalencias vamos a ELEGIMOS UN REPRESENTANTE y es lo que forma nuestros casos de prueba.

Ejemplo 2

Parte del código de un programa trata el precio final de un artículo en base a su precio de venta al público y determinado un descuento en %.

↓

0 ==< PrecioVenta
0 ==< Porcentaje ==< 100

Variable	Clase de Equivalencia	Estado	Representante
Precio de Venta	EC1: $x \geq 0$	Válido	1000
	EC2: $x < 0$	No Válido	-10
	EC3: x no numérico	No Válido	“Test”
Descuento	EC4: $0 \leq x \leq 100$	Válido	15
	EC5: $x < 0$	No Válido	-20
	EC6: $x > 100$	No Válido	200
	EC7: x no numérico	No Válido	‘Hola’

Casos de Prueba para CE Válidas

Variable	Clase de Equivalencia	Estado	Representante	TC1
Precio de Venta	EC1: $x \geq 0$	Válido	1000	*
	EC2: $x < 0$	No Válido	-10	
	EC3: x no numérico	No Válido	“Test”	
Descuento	EC4: $0 \leq x \leq 100$	Válido	15	*
	EC5: $x < 0$	No Válido	-20	
	EC6: $x > 100$	No Válido	200	
	EC7: x no numérico	No Válido	‘Hola’	

Para formar los casos de prueba para las clases de equivalencia no validas se hace es la combinación de una clase de equivalencia valida con una no valida, siempre tiene que haber una clase de equivalencia VALIDA con una NO VALIDA.

Casos de Prueba para CE NO Válidas

Variable	CE	Estado	Representante	TC1	TC2	TC3	TC4	TC5	T6
Precio de Venta	EC1: $x \geq 0$	Válido	1000	*	*	*	*		
	EC2: $x < 0$	No Válido	-10					*	
	EC3: x no numérico	No Válido	"Test"						*
Descuento	EC4: $0 \leq x \leq 100$	Válido	15	*				*	*
	EC5: $x < 0$	No Válido	-20		*				
	EC6: $x > 100$	No Válido	200			*			
	EC7: x no numérico	No Válido	'Hola'				*		

Análisis de valores límite

Permite ampliar o complementar la técnica anterior introduciendo una nueva regla para seleccionar los representantes, acá probamos con más énfasis los valores que están al límite de cada una de las clases, porque son los que frecuentemente no están definidos o el desarrollador no los implementó.

Al igual que el anterior esta técnica puede utilizarse en todos los niveles de pruebas.

En esta técnica también vamos a seleccionar representantes, por lo tanto, vamos a tener 2 partes:

- La partición en clases de equivalencias, donde vamos a evaluar un representante de cada clase
- y además vamos a hacer un análisis de valores límite y su entorno