Clase Testing Software

El **testing** NO puede asegurar ni calidad en el software ni software de calidad, ya que, la calidad de software no abarca sólo aspectos del producto.

El testing no se incluye en el plan de proyecto como un requerimiento, sino que yo decido hacer testing y lo incluyo en el plan de proyecto porque es una actividad necesaria.

TESTING no es igual a CALIDAD de PRODUCTO (este concepto va mucho más allá de lo que tiene que ver con el testing) y TESTING no es igual a CALIDAD de PROCESO (no tienen nada que ver).

Nuestro objetivo cuando planteamos hablar de testing o ejecutar las actividades de testing tienen que ver con que suponemos o sabemos que vamos a encontrar defectos de software. Proceso DESTRUCTIVO de tratar de encontrar defectos.

Principios de testing

El testing exhaustivo es imposible. No es viable en términos de costo, esfuerzo y tiempo testear el software de forma completa, la idea es abarcar la mayor cantidad de casos posibles dentro del testing con el menor esfuerzo.

Testing temprano: comienza desde que empezamos a escribir los casos de prueba. Desde el momento que comenzamos a planificar y definir su área de cobertura.

Decidir cuánto testing es suficiente depende de:

• Evaluación del nivel de riesgo. A mayor nivel riesgo, mayor testing.

• Costos asociados al proyecto

Error vs Defecto: Cuando en el testing hablamos de que encontramos defectos, estos fueron generados en una etapa anterior. En la implementación cuando se escribió el código se generó el error, pero cuando pasa a la etapa siguiente y lo detecto ahí, ese error se convierte en defecto.

Severidad

1 – Bloqueante: No me permite seguir ejecutando el caso de prueba.

2 – Crítico: A nivel de la funcionalidad que estoy ejecutando, es necesario corregir ese defecto sino la funcionalidad va a tener un funcionamiento que no es el adecuado.

3 – Mayor: Escalón que sigue siendo grave.

4 – Menor

5 - Cosmético: Tiene que ver con etiquetas, errores de ortografía, etc.

Prioridad: Relacionado con la prioridad de la funcionalidad y del caso de prueba en el contexto de esa funcionalidad dentro del negocio, de acuerdo al uso de la funcionalidad para el negocio voy a clasificarlo en niveles de tratamiento para determinar cuál resolver primero.

Niveles de prueba: Tiene que ver cómo yo voy a escalando en término de lo que voy probando, haciendo un enfoque desde menor granularidad a mayor granularidad.

Primer nivel de prueba: Testing unitario. Hago foco sobre un componente, cuando se termina de construir el componente de manera individual, lo testeo de manera independiente. No vinculandolo con otros componentes. Es normalmente ejecutado por el mismo desarrollador.

Testing de integración: Como su nombre lo indica, se comienza a juntar los diferentes componentes probados antes de manera aislada. Se van integrando de manera incremental, partiendo de los módulos críticos para el negocio y luego descendiendo en prioridad a las secundarias.

Testing de sistema: Aquí tenemos el sistema completo, debemos testear la aplicación funcionando como un todo. Se deben investigar los requerimientos funcionales tanto como no funcionales (seguridad, carga, performance, etc). Se trata de probar en un entorno que sea lo más parecido a uno de producción, con el objetivo de encontrarnos en un escenario similar al que nos vamos a encontrar cuando despleguemos el software.

Testing de aceptación: Es realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades. Tiene como objetivo que el usuario nos de el okey.

Cuando hablamos en términos de metodologías tradicionales o procesos definidos, la prueba es también un proceso con distintas etapas.

Planificación: Armamos el plan de prueba un partecita del plan de proyecto, definimos el criterio de aceptación, etc.

Identificación y especificación: De los casos de prueba, definimos qué es lo que vamos a probar. En esta etapa no hay necesidad de código sino que los requerimientos funcionales y no funcionales están definidos.

Ejecución: Ejecutamos los casos de prueba especificados anteriormente.
Análisis de fallas: Verificamos los resultados, si hay defectos o no llegamos al criterio de aceptación volvemos a ejecutar las pruebas hasta cumplirlo y llegar al momento de terminar las pruebas.
Para lo que es el caso de metodologías ágiles, el testing no es un proceso definido sino que queda embebido dentro del sprint tratando de encontrar los defectos lo antes posibles para poder corregirlos.
Caso de prueba: Escribir una secuencia de pasos que tienen condiciones y variables con la que yo voy a ejecutar el software que me van a permitir saber si este está funcionando correctamente.

Ciclo de test o prueba: Abarca la ejecución de la totalidad del ciclo de pruebas establecidos, en el caso de que haya errores el paso siguiente será arreglarlos para luego volver a ejecutar los casos de prueba para ver si se solucionan correctamente, esta se considera una nueva iteración del ciclo de test.
Testing con regresión: Plantea que cuando se corrigen errores, se deben volver a ejecutar la totalidad de los ciclos de prueba establecidos en la iteración anterior, ya que es posible que al resolver un problema en uno de los casos de prueba estemos introduciendo otro en otro caso de prueba.

El testing y el ciclo de vida: Una de las tareas importantes dentro del ciclo de vida tiene que ver con verificación y la validación, el testing permite cubrir una parte.
Dependiendo del ciclo de vida, el testing se inserta de manera distinta pero la idea es buscar aquellos en los que se inserte de una manera temprana.

Clase testing caja negra

Criterios para determinar cuándo finalizar las pruebas de testing:
Good enough: A medida que vamos encontrando y resolviendo defectos, va a haber una tendencia a disminuir los efectos encontrados. Corta cuando la cantidad de defectos encontrados llegue a cierto término.
Presupuesto: Corta cuando la cantidad de dinero invertido en pruebas llega a tal monto.

Como el testing no es exhaustivo es necesario aplicar estrategias para maximizar la cantidad de defectos minimizando el esfuerzo requerido para hacerlo. Estas estrategias surgen para el diseño de casos de pruebas, artefacto por excelencia del testing, que contiene un conjunto de condiciones y un conjunto de pasos que se deben ejecutar para poder garantizar que el resultado esperado sea o no igual que el resultado obtenido y de esa forma encontrar defectos.
Estrategias de caja blanca: Podemos ver el detalle de la implementación de la funcionalidad, código o estructura interna, pudiendo diseñar el casos de prueba para poder garantizar la cobertura (testing de coverage).
Estrategias de caja negra: No conozco el código, solo lo puedo analizar en términos de entradas y salidas. Se identifican las diferentes entradas que una funcionalidad puede tener, voy a elegir los valores con los cuales puede ejecutar esa funcionalidad para finalmente obtener el resultado.
Dentro de la implementación de la estrategia existen diferentes métodos:

- Basados en la experiencia
- Basados en especificaciones

Partición de equivalencias: Primero analiza cuales son las condiciones externas que van a estar involucradas en el desarrollo de una funcionalidad, tanto entradas como salidas. Luego se analiza para las condiciones externas el subconjunto de valores posibles que pueden tomar cada una de esas condiciones externas que producen un resultado equivalente.
La división de partición de equivalencia se basa en cuales son los resultados posibles que vas a tener.
Una partición de equivalencia es un subconjunto de valores que puede tomar una condición externa, de forma tal que si tomo cualquier valor posible de ese subconjunto el resultado en la ejecución de esa funcionalidad es equivalente.
Como segundo paso se diseñan los casos de prueba. Tomo un valor representativo de cada una de esas clases de equivalencia y escribo los pasos ordenados para ejecutar la funcionalidad especificando cada uno de los valores que se ingreso.
La prioridad se basa en poder decidir que casos de prueba se van a ejecutar antes que otros, es importante priorizar aquellos que nos permitan encontrar la mayor cantidad de defectos y que garantice que la funcionalidad pueda cumplirse. Donde:

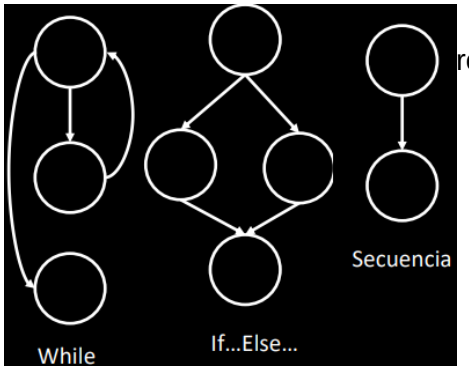
-Prioridad alta: Combinación de todos aquellos escenarios que presentan caminos felices de todas las clases de equivalencia válidas.

-Prioridad baja: Validaciones relacionadas con formato no esperado o no ingresa valor
-Prioridad media: Combinación de valores que pueden ejecutarse bajo ciertas condiciones que producen alguna falla o camino no feliz
Precondiciones: Conjunto de valores o de características que tiene que tener mi contexto para que yo pueda llevar adelante el caso de prueba en particular.

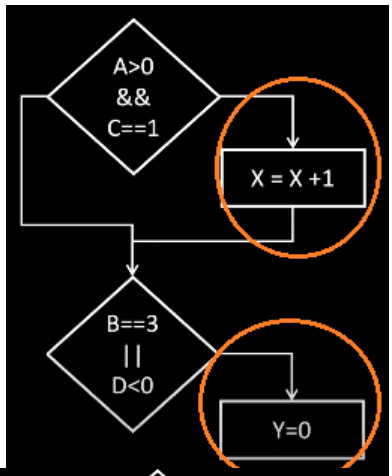
Ejemplo de utilización de mapa iterativo: Las coordenadas del GPS son 32° 23’ 234’
No ingresa valores: Combinación de todos los campos sin ingresar valores.

Dentro de esta estrategia se puede implementar el análisis de valores límites, se basa en que la mayor cantidad de defectos se suelen encontrar en los extremos de los intervalos. Entonces al escribir los casos de prueba en vez de tomar cualquier valor de la clase de equivalencia vamos a tomar uno que se encuentre en el extremo.

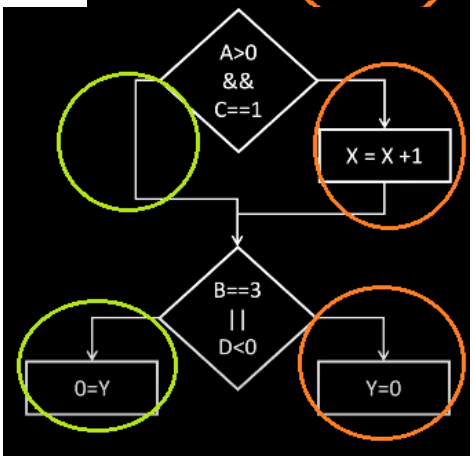
Clase testing caja blanca
Método de caja blanca:
Cobertura se entiende por formas o estrategias de recorrer los distintos caminos que nuestro código nos provee para desarrollar una funcionalidad.
Cobertura de enunciados o caminos básicos: Intenta garantizar que vamos a recorrer todos los caminos independientes que tiene nuestra funcionalidad. Intenta representar el conjunto de caminos bajo una métrica llamada complejidad ciclomática, nos da una idea de cuántos caminos independientes una funcionalidad tiene y cuantos casos de prueba vamos a tener que hacer para recorrerlos.



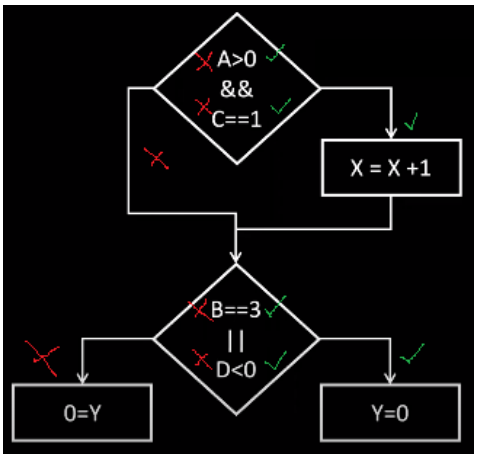
La ejecución de los casos de prueba se representa mediante grafos de flujo.



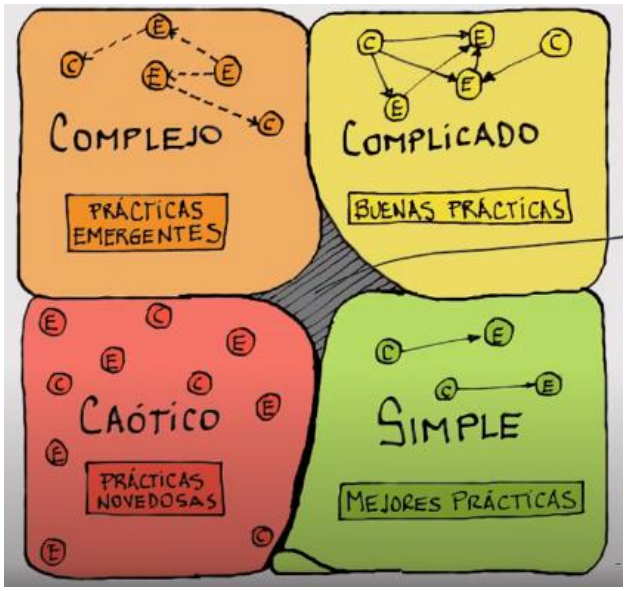
Cobertura de sentencias: Una sentencia es cualquier instrucción como asignación de variables u invocación a métodos, etc.
Trata de encontrar la cantidad mínima de casos de prueba que permitan cubrir todas las sentencias.
En este caso, para poder cumplir con ambas sentencias X = X + 1 e Y = 0, solo hará falta elegir una combinación de valores para las variables que permita ir por las 2 ramas del true.
Cobertura de decisión: Una decisión es una estructura de control completa.
Trata de encontrar la cantidad mínima de casos de prueba que permitan cubrir todas las decisiones.
En este caso, escribimos un caso de prueba que mediante la combinación de valores para las variables nos permita cubrir los 2 caminos por true y otro que nos permita cubrir el camino por false.



Cobertura decisión/condición: Busca no solamente evaluar las decisiones sino que también valorar todas las condiciones.
Haciendo que todas las condiciones sean verdaderas, logramos que las decisiones sean verdaderas. Si hacemos que todas las condiciones sean falsas, logramos que las decisiones sean falsas. Logramos así que se cubran todas.
Cobertura múltiple: Busca evaluar el combinatorio de todas las condiciones en todos sus valores de verdad posibles.



Clase Frameworks para Escalar SCRUM
La escalabilidad es la habilidad para reaccionar y adaptarse sin perder calidad o bien para estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos.
Para darle un contexto a lo que tiene que ver con la complejidad vamos a usar un framework propuesto por Snowden, llamado Cyefin donde se plantea que de acuerdo a los entornos donde nos encontramos vamos a tener distintas formas de resolver o encarar la complejidad.
Se parte de los simple o conocidos donde se tiene una causa que desencadena una única consecuencia y que implementando buenas prácticas es fácil de resolver
Luego se encuentran los escenarios complicados, donde se tiene que una causa que puede terminar con varios efectos. Requiere de utilizar las buenas prácticas para abarcar la variedad de respuestas correctas.



Dentro de los escenarios complejos no se puede asociar una causa a una efecto fácilmente, no hay respuestas correctas. La clave en este tipo de escenarios es aplicar prácticas emergentes relacionadas con lo que propone scrum y los frameworks ágiles como la inspección y la adaptación.
Esto requiere que se explore todo lo que pueda sobre el problema, inspeccione qué funciona o no y luego se adapte en función de lo que ha aprendido. Scrum aprovecha los beneficios de los pequeños ciclos de desarrollo (sprints) para contener la generación de este conocimiento.
Y finalmente el último de los escenarios, el caótico, es donde se intenta buscar prácticas novedad u ideas para poder resolverlo. Requiere una solución rápida, por eso se utiliza la práctica novedosa para poder salir de ese caos donde la situación se encuentra.
La situación borrosa que se encuentra en el medio, es el desorden, donde tengo falta de información para poder catalogar el problema dentro de un escenario entonces no puedo trabajarlo.

Construimos productos con la ayuda de un proyecto como unidad de gestión que sirve para guiar la construcción de este producto. Scrum es un framework que sirve para gestionar proyectos para crear productos de software.

Nexus: Rasgos a simple vista
Posee un único product backlog.
La ceremonia de Product Backlog Refinement es obligatoria
El incremento del producto es uno para todos los equipos que están trabajando y no en función de partes de cada equipo.
La planificación del sprint se basa en cómo vamos a particionar la parte del product backlog en cada uno de los Equipos Scrum en un Nexus y luego hacer su propia planning de forma separada.
Daily una para cada equipo individual y otra nexus donde cada equipo inspecciona su avance sobre el incremento integrado.
La retrospectiva es una combinación de cada equipo por si mismo y en términos de Nexus
Review hay una sola, y no individualmente por equipo.

Roles
Nexus Integration Team (NIT): Tiene como función resolver los aspectos que tienen que ver con la integración y con las dependencias que surgen como de la cantidad de equipos trabajando sobre un mismo product backlog para obtener un único incremento del producto. Estas dependencias ocasionadas por un equipo podrían condicionar el avance de otro equipo. Formado por: El Product Owner, Scrum Master y miembros de los Equipos Scrum individuales en ese Nexus.
Product Owner en NIT:
Scrum Master en NIT
NIT Members

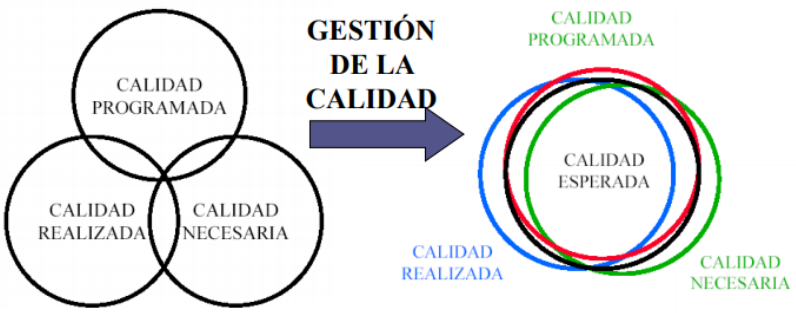
SHU-HA-RI - Madurez Lean-Ag
Cual seria el mecanismo o camino que se debería seguir para ser capaces a partir de nuestro aprendizaje de definir nuevas reglas.
Entonces antes de pensar modificar o adaptar lo que está definido, primero hay que aprender bien lo que está definido. Dominarlo, entenderlo y usarlo. Incorporar la experiencia que implica seguir las reglas.
Una vez superada esa instancia, puedo “romper las reglas”, decidir que usar y que no dependiendo del contexto. En esta etapa se puede comenzar a plantear que cosas modificar para adaptarlas a tu realidad .
Y finalmente en un futuro poder desarrollar la capacidad de plantear prácticas emergentes que resuelvan nuevos escenarios complejos. Es el estado en donde podemos hacer nuestra propia versión de una metodología.

Clase Aseguramiento de Calidad de Proceso y de Producto

Calidad: La definición que se le da muchas veces tiene que ver con la percepción o la mirada que tiene los distintos actores que trabajan sobre ella. Cuando se habla sobre que la calidad del producto o servicio tiene que ver con características o aspectos que estén relacionados con la habilidad de alcanzar las necesidad que se habían planteado, el punto fundamental nace en ¿según las necesidades de quien? dando lugar a varias variables sobre la “calidad”.
La calidad no es única en el tiempo, se puede tener diferentes medidas de la calidad del producto de acuerdo al contexto en el que nos encontremos y debe satisfacer las expectativas y necesidades de todos los involucrados.

Hacer mas testing no le agrega mas calidad al producto. La calidad tiene que estar embebida a lo largo de todo el proceso. Clave para trabajar en el área de calidad: Las personas involucradas son la clave para lograrlo y se logra mediante la capacitación.

La calidad se podría plantear desde 3 conceptos:
Calidad programada: Es la que se espera que tenga el producto de software que se está construyendo como equipo de desarrollo
Calidad necesaria: Calidad mínima que debería tener el producto de software
Calidad realizada: Es la calidad que realmente tiene el producto desarrollado



Lo que se busca con la gestión de la calidad es que la intersección entre estos 3 conceptos sea lo más coincidente posible. Esto implica que cuanto más lejos estén una de otra más dificultades y más problemas se presentan.

Un proceso se instancia o cobra vida a través un proyecto que conlleva a la construcción de un producto.
En los procesos de empíricos para tener un producto de calidad es la gente que trabaja en el proyecto, al instanciarlo se puede evaluar el proceso y mejorarlo.
Cuando se define el proceso es importante determinar qué actividades me permiten lograr el aseguramiento de la calidad.

Modelos contra los que me comparo para definir la calidad del producto: calidad en Uso (ISO 25010: 2011), calidad de producto (ISO 25010:2011) y calidad de Datos (ISO 25012:2008)

El único aspecto en el podemos influir es en el proceso por lo que el aseguramiento de la calidad se centra en este contexto.

Para el aseguramiento de la calidad de software debemos tener un estándar definido y tengo que comparar contra ese estándar para saber si lo alcance.

Clase de Lean y Kanban

Lean el trabajo continuo que se mantiene a lo largo del tiempo.
Principios:
Eliminar desperdicio: El software funcionando es lo que da valor, entonces el desperdicio hace referencia al retrabajo o construir cosas que no vamos a utilizar.

Kanban

En desarrollo de software el principal referente de la filosofía Lean es Kanban, framework de mejora de proceso.
Significa: signal card, permite tener visible la señal que necesito para saber cual es el estado o en qué momento se encuentra mi proceso.

Se basa en el concepto de just in time, teniendo solamente lo que se necesita así reducir los costos.
También se maneja con el concepto de administración de colas que hacen cosas diferentes haciendo foco que en ninguna de ellas haya tiempo ocioso ni sobrecargo de tareas. Separarlos por la cola permite que se absorba la demanda variable.
No tiene una definición del proceso que hay que seguir, sino que el framework mejora el proceso que está definido completando la mejora continua.

Principios:
• Visualizar el Flujo: El flujo de trabajo entre las distintas colas debe ser visible.
• Limitar el Trabajo en progreso (WIP): Limita cuando trabajo en progreso podemos tener en cada uno de los carriles. Hace foco en limitarse en un momento del tiempo a hacer una única cosa y no multitasking.
• Administrar el flujo
• Hacer explícitas las políticas
• Mejorar colaborativamente: No hay persona que asigne la tarea y priorice lo que se tiene que hacer eso se resuelve en términos del equipo

¿Cómo aplicar este framework?

Parte de entender el proceso actual, cómo funciona, qué tareas se realizan, etc. Una vez incorporado se deciden los límites, el work in progress de cada una de las etapas y finalmente se prueba para obtener un feedback.
El trabajo se divide en pieza de trabajo. Se establecen las políticas. Se mapea el proceso, en columnas de izquierda a derecha mostrando cómo el proceso fluye. Cada uno de los equipos se autoasigna la tarea en la cual trabajará.

Métricas fundamentales

Lead Time = Vista del cliente, representa el tiempo que transcurre desde que ingresa la pieza de trabajo hasta que esta terminada. Para mi cliente la entrega de valor tiene que ver con esta métrica.
Cycle Time = Vista interna, saliendo de esa cola de acumulacion inicial tiempo que sucede entre el inicio y el final del proceso, para un ítem de trabajo dado
Touch Time = tiempo específico en el que cada uno de los recursos que está en mi proceso está trabajando en una determinada pieza de trabajo, cuanto tiempo paso el ítem en columnas “trabajado en curso”.

Herramientas de Scrum para las métricas ágiles
Taskboard (tablero): Permite tener visibilidad en el ambiente de trabajo. Story con sus tareas planificadas en el sprint backlog, to do, doing, to verify y done. Cada una con su estimación en horas.
Definition of done: Para saber si la feature esta terminada.
Burndown chart: Es descendente, ya que va decrementando los puntos de historias que me quedan para quemar.

Métricas ágiles:
Métrica ágil por excelencia la velocidad: Cuantos puntos quemamos al finalizar el sprint. La unidad de tiempo es un sprint. Solo cuenta el trabajo completado y aceptado por el product owner en la review.
Capacidad: Horas de trabajo disponibles por dia x días disponibles iteración. La capacidad se determina fundamentalmente en función de la velocidad, nos basamos en lo que realmente paso.