

TESTING DE SOFTWARE:

En el contexto de “Asegurar la calidad vs controlar la calidad”

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:

- La calidad no puede “inyectarse” al final
- La calidad del producto depende de tareas realizadas durante **todo** el proceso
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos
- La calidad no solamente abarca aspectos del producto sino también del proceso y como estos se pueden mejorar, que a su vez evita defectos recurrentes.
- El testing NO puede asegurar ni calidad en el software ni software de calidad.

El **testing** no es el responsable ni asegurar la calidad ni desarrollar software de calidad.

Es una *disciplina* en la que se contemplan aspectos con respecto al producto y con los requerimientos de calidad, pero no nos asegura que nuestro proceso para construir el SW sea de calidad.

No abarca solamente aspectos del producto

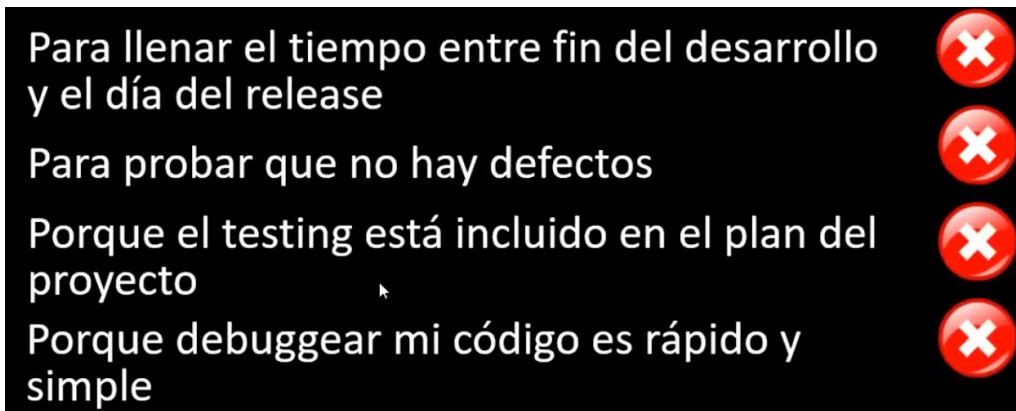
Hay otras disciplinas o actividades que se ejecutan antes que el testing y que permiten *asegurarnos la calidad en el producto* que vamos a construir. Esas actividades abarcan como por ejemplo la “inspección” y que están fuera de la prueba de software o testing.

¿Por qué el testing es necesario?

- ✓ **Porque la existencia de defectos en el SW es inevitable:** el testing es un proceso que iniciamos asumiendo que va a haber defectos en el SW (ya que esta desarrollado por personas).
- ✓ **Para evitar ser demandado por mi cliente**
- ✓ **Para reducir riesgos**
- ✓ **Para construir confianza en mi producto**
- ✓ **Porque las fallas son muy costosas**
- ✓ **Para verificar que el software se ajusta a los requerimientos y validar que las funciones se implementan correctamente**

Algunas frases falsas:

El testing no se incluye como un requerimiento, sino que mas bien uno decide hacer testing ya que es una actividad necesaria.



- **El testing es una etapa que comienza al terminar de codificar:** puede comenzar mucho antes, incluso desde el momento en el que se empiezan a definir los requerimientos o el plan de proyecto
- **El testing es probar que el software funciona:** el testing es un proceso destructivo cuyo objetivo es encontrar defectos en el SW, no probar que el SW funcione.
- **Testing = Calidad del Producto**
- **Testing = Calidad de Proceso**
- **El tester es el enemigo del programador.**

Definición de “Testing”:

- Proceso **DESTRUCTIVO** de tratar de **encontrar defectos (cuya presencia se asume)** en el código.
- Se debe ir con una actitud negativa para demostrar que algo es incorrecto.
- Testing exitoso **es aquel que encuentra defectos**
- Mundialmente el testing supone **del 30 al 50% del costo de un SW confiable**

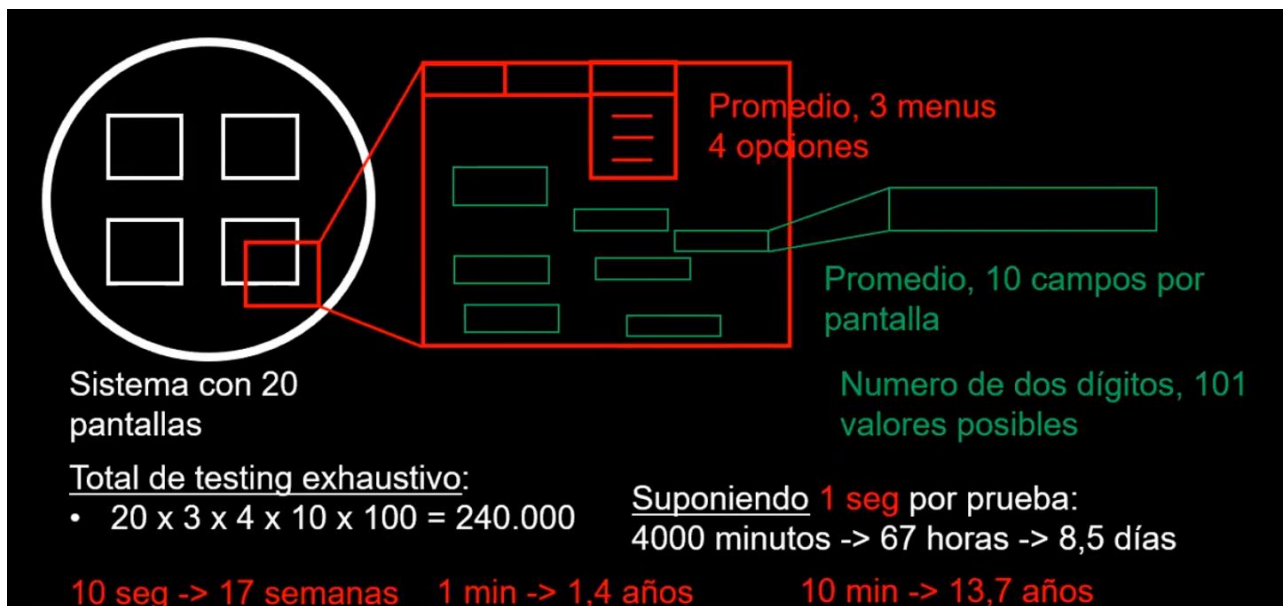
El **testing** es un *proceso destructivo*, porque su objetivo es *encontrar defectos*, o porque intenta romper el SW. Los defectos que tratamos de encontrar se basan en la premisa de que se asume de que *van a existir defectos*, ya que es inevitable.

Es un proceso destructivo también porque el testing es una *actitud negativa*, para lo cual uno se enfoca en encontrar los defectos en el SW sabiendo que van a haber defectos.

Principios del Testing:

- **El testing muestra presencia de defecto**
- **El testing exhaustivo es imposible:** no es viable en términos de costos y esfuerzo, no se puede probar el SW de manera completa, por lo que se abarcan la mayor cantidad de casos posibles con el menor esfuerzo posible. Nunca se va a poder probar el SW de manera completa.
- **Testing temprano:** probar el software con el SW funcionando, también empieza al escribir casos de prueba, definir el nivel de cobertura del testing, etc.
- **Agrupamiento de Defectos**
- **Paradoja del Pesticida:** cuando uno utiliza mucho un pesticida, uno se concentra mucho en eliminar la plaga que el pesticida puede eliminar, pero perdemos el foco de otros bichos o plagas que están ahí pero no estaríamos viendo. Quizá se logran armar casos de prueba que se ejecuten de manera automática, pero dejar cosas afuera que no estoy viendo por estar concentrado en eliminar defectos en un cierto contexto (generalmente pasa con el testing automatizado)
- **El testing es dependiente del contexto**
- **Falacia de la ausencia de errores.**
- **Un programador debería evitar probar su propio código**
- **Una unidad de programación no debería probar sus propios desarrollos**
- Examinar el software para probar que no hace lo que se supone que debería hacer es la mitad de la batalla, la otra mitad es ver que hace lo que lo que no se supone que debería hacer.
- **No planificar el esfuerzo de testing sobre la suposición de que no se van a encontrar defectos.**

¿Cuánto testing es suficiente?



En este ejemplo, se indica porque el *testing exhaustivo* no es posible, ya que para una sola persona le llevaría un año y medio probar todos los casos posibles para ese caso, sin considerar el testing de regresión.

- ✓ **Cuando se tiene la confianza de que el sistema funciona correctamente:** hay que determinar cuando hay que dejar de testear.
- ✓ **Depende del riesgo de tu sistema:** va a depender de cada aplicación y de su respectiva importancia

Algunas frases falsas:

Cuando se terminó todo lo planificado



Nunca es suficiente



Cuando se ha probado que el sistema funciona correctamente



Conclusiones:

- El testing exhaustivo es imposible
- *Decidir cuanto testing es suficiente depende de:*
 - **Evaluación del nivel del riesgo:** cuanto mayor sea, mas testing hay que realizar, por ende más esfuerzo y dinero.
 - **Costos asociados al proyecto:** disponibilidad monetaria del presupuesto del proyecto
- *Usamos los riesgos para determinar:*
 - Que testear primero
 - A que dedicarle más esfuerzo de testing
 - Que no testear (por ahora)

Criterio de Aceptación:

El **criterio de aceptación** es algo que yo acuerdo con mi cliente, hasta donde se va a probar, para lo que eso va a quedar como respaldo por si aparecen defectos más adelante. Debe ser una **variable medible**.

Es lo que comúnmente se usa para resolver el problema de determinar cuando una determinada fase de testing ha sido completada.

Al ser una variable medible, *puede ser definido en términos de:*

- Costos
- % de tests corridos sin fallas
- Fallas predichas aun permanecen en el SW
- No hay defectos de una determinada severidad en el SW

La Psicología del Testing:

La búsqueda de fallas puede ser visto como una critica al producto y/o su autor, o como una crítica al programador

La construcción del SW requiere otra mentalidad a la de testear el software.

Es una actividad que se hace orientada a la calidad del producto, asumiendo que siempre van a existir defectos sin importar quien programe.

Error vs Defecto:

En términos de testing, usamos el termino “**defecto**”, esto es porque cuando en el testing encontramos *defectos*, en esa etapa de testing, fueron generados en una etapa anterior (en este caso en la etapa de implementación).

Cuando se escribió el código se genero el **error**, y cuando ese *error* paso a la etapa siguiente el *error* se convirtió en *defecto*.

El **error** se detecta en la misma etapa en la que se genera, en cambio el **defecto** se lo detecta en una etapa posterior.

Clasificación de Defectos:

Se clasifican según las siguientes variables:

Severidad

1. **Bloqueante:** no se puede continuar con el caso de prueba debido al defecto
2. **Critico:** es necesario corregirlo sino la funcionalidad va a tener un error o funcionamiento que no es el adecuado.
3. **Mayor:** lo mismo que critico pero de menor gravedad.
4. **Menor:** una funcionalidad que se ejecuta por un curso alternativo que no permite seguir avanzando y el mensaje que muestra no es el adecuado
5. **Cosmético:** tiene que ver con etiquetas, errores de ortografía, etc.

Tiene que ver con que pasa si el defecto permanece en el SW, no se piensa con respecto al negocio.

Prioridad

1. **Urgencia**
2. **Alta**
3. **Media**
4. **Baja**

Se define según la prioridad de esa funcionalidad para el negocio, es decir, que tan importante es la funcionalidad para el negocio.

Niveles de Prueba:

Los niveles de prueba tienen que ver en como voy escalando en termino de lo que se va probando desde la menor granularidad hacia la mayor granularidad.

I. Testing Unitario

Se hace foco sobre un componente, se termina de construir y se testea de manera independiente sin vincularlo con otros componentes (es el testing que suele ejecutar el mismo programador que desarrollo el código).

- Se prueba cada componente tras su realización/construcción
- Solo se prueban componentes individuales
- Cada componente es probado de manera independiente
- Se produce con acceso al código bajo pruebas y con el apoyo del entorno de desarrollo, tales como un framework de pruebas unitarias o herramientas de depuración
- Los errores se suelen reparar tan pronto como se encuentran, sin constancia oficial de los incidentes.

II. Testing de Integración

Se plantea empezar a juntar los componentes que se probaron de manera aislada, generalmente al integrar los componentes se lo realiza de manera incremental, es decir, de a poco, y a medida que se van integrando se los va testeando y se corrigen errores. Siempre se prueban primero los módulos críticos para el negocio, y a partir de allí se lo extiende a funcionalidades que son de menor prioridad para el negocio.

- Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
- Cualquier estrategia de prueba de versión o de integración debe ser **incremental**, para lo que existen dos esquemas principales:
 - Integración de arriba hacia abajo (top-down)
 - Integración de abajo hacia arriba (bottom-up)
- Lo ideal es una combinación de ambos esquemas
- Tener en cuenta que los módulos críticos deben ser probados lo más tempranamente posible
- Los puntos clave del test de integración son simples:

- Conectar de a poco las partes más complejas
- Minimizar la necesidad de programas auxiliares

III. Testing de Sistema

Se tiene el sistema completo, entonces se prueba la aplicación funcionando como un todo, se busca sobre todo el cumplimiento de requerimientos funcionales y no funcionales (seguridad, carga, performance, etc.), y se busca un entorno lo mas parecido al de producción (es decir, lo más similar al despliegue).

- Es la prueba realizada cuando una aplicación esta funcionando como un todo (Prueba de construcción Final)
- Trata de determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, stress, performance, etc.)
- El entorno de prueba debe corresponder al entorno de producción tanto como sea posible para reducir al mínimo el riesgo de incidentes debidos al ambiente específicamente y que no se encontraron las pruebas.
- Deben investigar tanto requerimientos funcionales y no funcionales del sistema.

IV. Testing de Aceptación:

Es el testing que hace el **usuario**, no es un testing técnico, para ver si la aplicación se ajusta a sus necesidades o si es lo que estaba esperando, y se hace foco en que el sistema hace lo que el espera que haga.

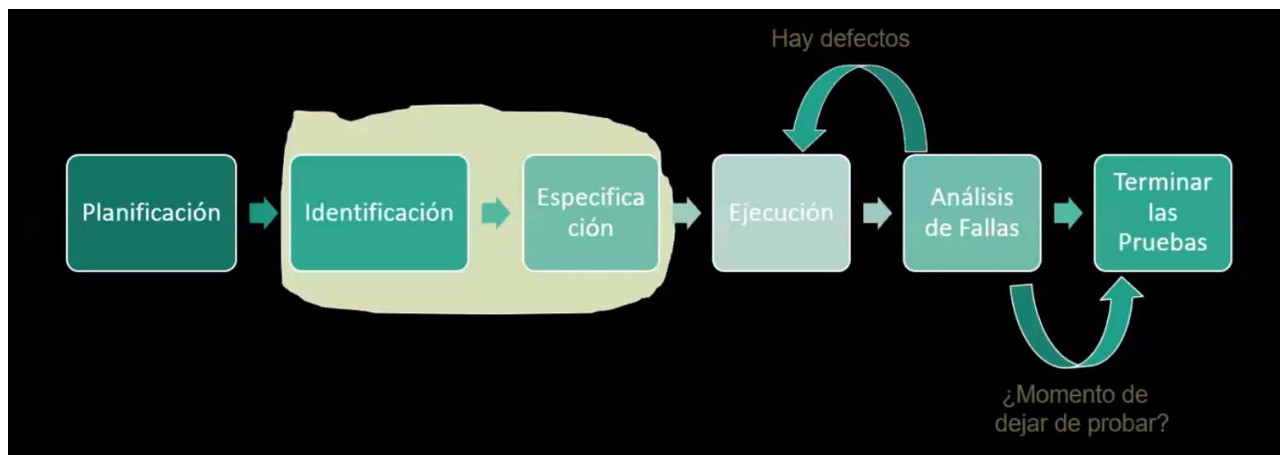
- Es la prueba realizada por el usuario para determinar si la aplicación se ajusta a sus necesidades.
- La meta en la prueba de aceptación es el de establecer confianza en el sistema, las partes del sistema o las características especificas y no funcionales del sistema.
- Encontrar defectos no es el foco principal en las pruebas de aceptación.
- Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta).

Se realizaría en la **Review** (donde nuestro cliente o PO mira las features e indica si era lo que esperaba o no), dentro de lo que sería el ciclo de vida del producto en el framework de SCRUM

Proceso de Pruebas

La prueba también es un proceso y dispone de distintas etapas, dentro de lo que sería un proceso definido.

En metodologías ágiles este proceso no ocurre.



- a. **Planificación:** cuando armamos el plan de pruebas, definimos el criterio de aceptación, es un plan subsidiario del plan de proyecto. Se acuerdan las metas y objetivos del cliente, estrategias de testing, que pruebas van a ser manuales y cuales automatizadas, etc.

- La Planificación de las pruebas es la actividad de verificar que se entienden las metas y los objetivos del cliente, las partes interesadas (stakeholders), el proyecto, y los riesgos de las pruebas que se pretende abordar.
- Construcción del Test Plan:
 - Riesgos y Objetivos del Testing
 - Estrategia de Testing
 - Recursos
 - Criterio de Aceptación
- Controlar:
 - Revisar los resultados del testing
 - Test coverage y criterio de aceptación
 - Tomar decisiones



b. Proceso

- a. Identificación: de los casos de prueba
- b. Especificación: definimos que es lo que vamos a probar

- Revisión de la base de pruebas
- Verificación de las especificaciones para el software bajo pruebas
- Evaluar la testeabilidad de los requerimientos y el sistema
- Identificar los datos necesarios
- Diseño y priorización de los casos de las pruebas
- Diseño del entorno de prueba

c. Ejecución: ejecutamos esos casos de prueba

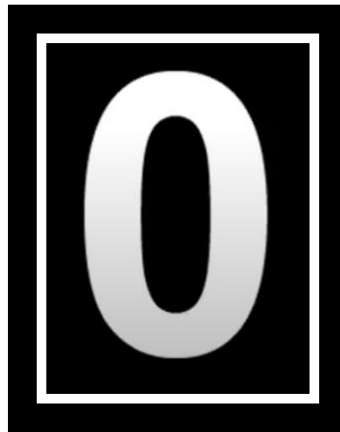
- Desarrollar y dar prioridad a nuestros casos de prueba
- Crear los datos de prueba
- Automatizar lo que sea necesario
- Creación de conjuntos de pruebas de los casos de prueba para la ejecución de la prueba eficientemente.
- Implementar y verificar el ambiente.
- Ejecutar los casos de prueba
- Registrar el resultado de la ejecución de pruebas y registrar la identidad y las versiones del software en las herramientas de pruebas.
- Comparar los resultados reales con los resultados esperados.

d. Análisis de fallas: se analizan los resultados de la ejecución de casos de pruebas, si hay defectos o si no se llega al criterio de aceptación, se vuelve a ejecutar y se vuelven a analizar las fallas, hasta que se cumpla con el criterio de aceptación

- Evaluar los criterios de Aceptación
- Reporte de los resultados de las pruebas para los stakeholders.
- Recolección de la información de las actividades de prueba completadas para consolidar.
- Verificación de los entregables y que los defectos hayan sido corregidos.
- Evaluación de cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

e. Terminar las pruebas: se concluye la prueba una vez llegado al criterio de aceptación.

¿Cuántas líneas de código necesito para empezar a hacer testing?



¿Y en SCRUM?

El testing se hace pero queda embebido dentro de las actividades del sprint, con las mismas características, tratando de encontrar los defectos lo antes posible y corregirlos lo antes posible. No está establecido como un proceso definido.

Casos de Prueba:

Tiene que ver con escribir una secuencia de pasos que tienen condiciones y variables con las que voy a ejecutar el SW y me van a permitir saber si el SW está funcionando correctamente o no.

- Set de condiciones o variables bajo las cuales un tester determinara si el software está funcionando correctamente o no.
- Buena definición de casos de prueba nos ayuda a **reproducir** defectos.

Si el resultado esperado no se logra quien tiene que reproducir un defecto para corregirlo, simplemente tiene que seguir los pasos que están enunciados en el caso de prueba.

Es fundamental para que el tester sepa que es lo que está testeando y que es lo que está ejecutando para validar si el resultado es el esperado o si no es, como se hizo para llegar a ese resultado.

Condiciones de prueba:

Esta es la reacción esperada de un sistema frente a un estímulo particular, este estímulo está constituido por las distintas entradas.

Una condición de prueba debe ser probada por al menos un caso de prueba

Características del Caso de Prueba:

Un caso de prueba es la unidad de la actividad de la prueba.

Consta de tres partes:

- **Objetivo:** la característica del sistema a comprobar. Tiene que ver con que es lo que voy a probar concretamente
- **Datos de entrada y de ambiente:** datos a introducir al sistema que se encuentra en condiciones preestablecidas. Condiciones previas o precondiciones que se deben contemplar.
- **Comportamiento esperado:** la salida o acción esperada en el sistema de acuerdo con los requerimientos del mismo.

¿Como encontrar defectos?

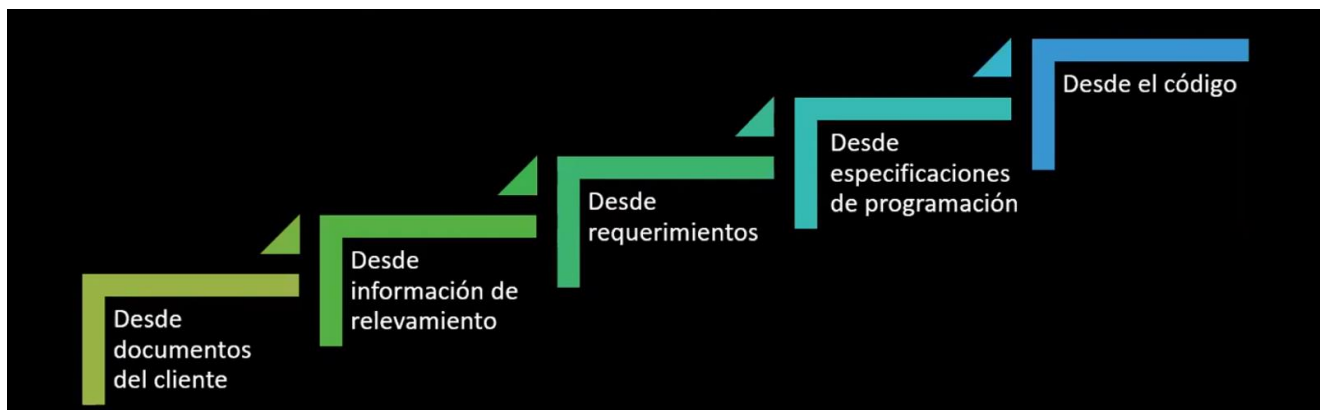
Hay que ejecutar la menor cantidad de casos de prueba pero que abarque el máximo nivel de cobertura, o que permita cubrir varios aspectos de la funcionalidad.

“Los bugs se esconden en las esquinas y se congregan en los limites...”

Boris Beizer

- OBJETIVO: descubrir errores
- CRITERIO: en forma completa
- RESTRICCIÓN: con el mínimo esfuerzo y tiempo

Derivación de Casos de Prueba:



Por ejemplo, derivar desde una *User Story* (los casos de prueba NO SON las pruebas de usuario de la *User*), desde Casos de Uso

Conclusiones sobre la Generación de Casos:

- Ninguna técnica es completa
- Las técnicas atacan distintos problemas
- Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una
- Depende del código a testear
- Sin requerimientos todo es mucho más difícil
- Tener en cuenta la conjetura de defectos
- Ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas.

Ciclo de Test:

Un **ciclo de pruebas** abarca la ejecución de la totalidad de los casos de prueba establecidos aplicados a una misma versión del sistema a probar.

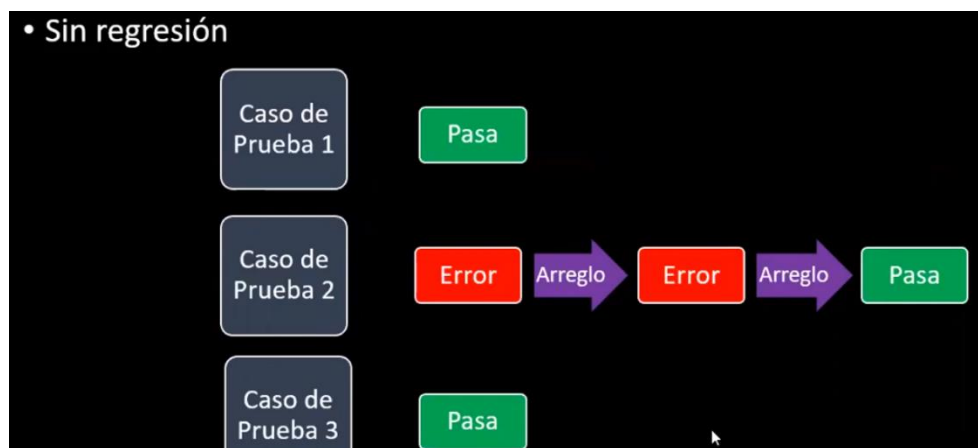
Cada vez que se ejecutan todos los casos de prueba sobre una misma versión de un sistema, es un ciclo de test. Cuando se corrigen los errores y se vuelve a ejecutar, es un nuevo ciclo de test.

Se ejecutan tantos como sean necesarios hasta alcanzar el criterio de aceptación.

Regresión:

Al concluir un ciclo de pruebas, y reemplazarse la versión del sistema sometido al mismo, debe realizarse una verificación total de la nueva versión, a fin de prevenir la introducción de nuevos defectos al intentar solucionar los detectados.

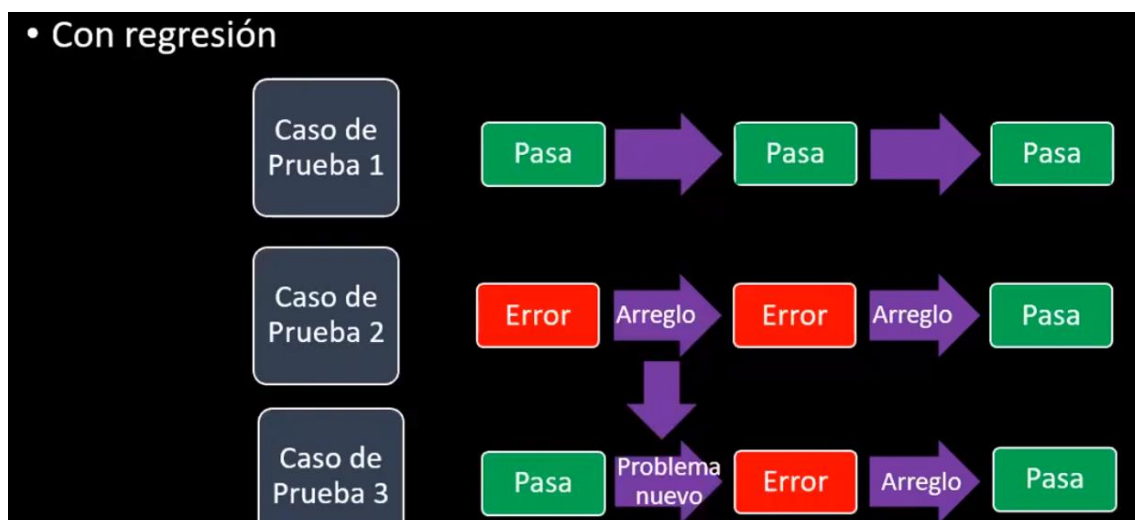
Lo que se hace es volver a verificar todos los casos de prueba de manera completa, debido que al intentar corregir un defecto, es bastante probable que se haya introducido otro defecto que apareció con la corrección.



Se asume que el caso de prueba 1 y 3 siguen funcionando...



Ahora bien con regresión....



Smoke Test:

Se hace una primera corrida del sistema en términos generales (no exhaustivos) para ver que no haya ninguna falla catastrófica en el sistema.

Tipos de Pruebas:

I. Testing Funcional:

Las pruebas se basan en funciones y características (descrita en los documentos o entendidas por los testers) y su interoperabilidad con sistemas específicos.

- Basado en ***Requerimientos***
- Basado en los ***Procesos de Negocio***

II. Testing No Funcional

Es la prueba de “como” funciona el sistema.

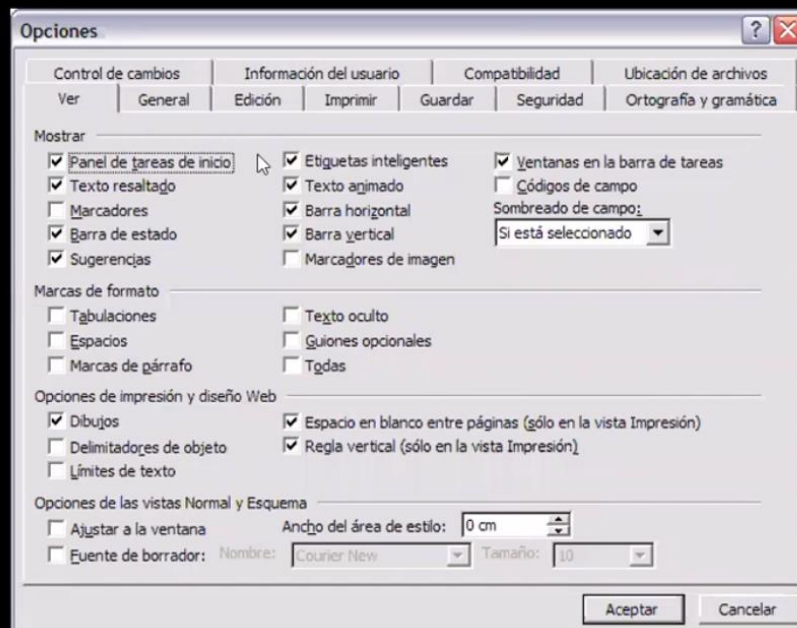
NO HAY QUE OLVIDARLAS. Los requerimientos no funcionales son tan importantes como los funcionales.

- Performance de ***Testing***
- Pruebas de ***Carga***
- Pruebas de ***Stress***
- Pruebas de ***Usabilidad***
- Pruebas de ***Mantenimiento***
- Pruebas de ***Fiabilidad***
- Pruebas de ***Portabilidad***

III. Pruebas de Interfaces de Usuarios

Usuario en control
+
Muchas combinaciones
=
Más pruebas

- Funciones de negocios
- **Interfaces de usuarios**
- Performance
- Carga
- Estrés
- Volumen
- Configuración
- Instalación



***Las GUIs,
son mucho
más
complejas
que las
interfaces
basadas en
caracteres***

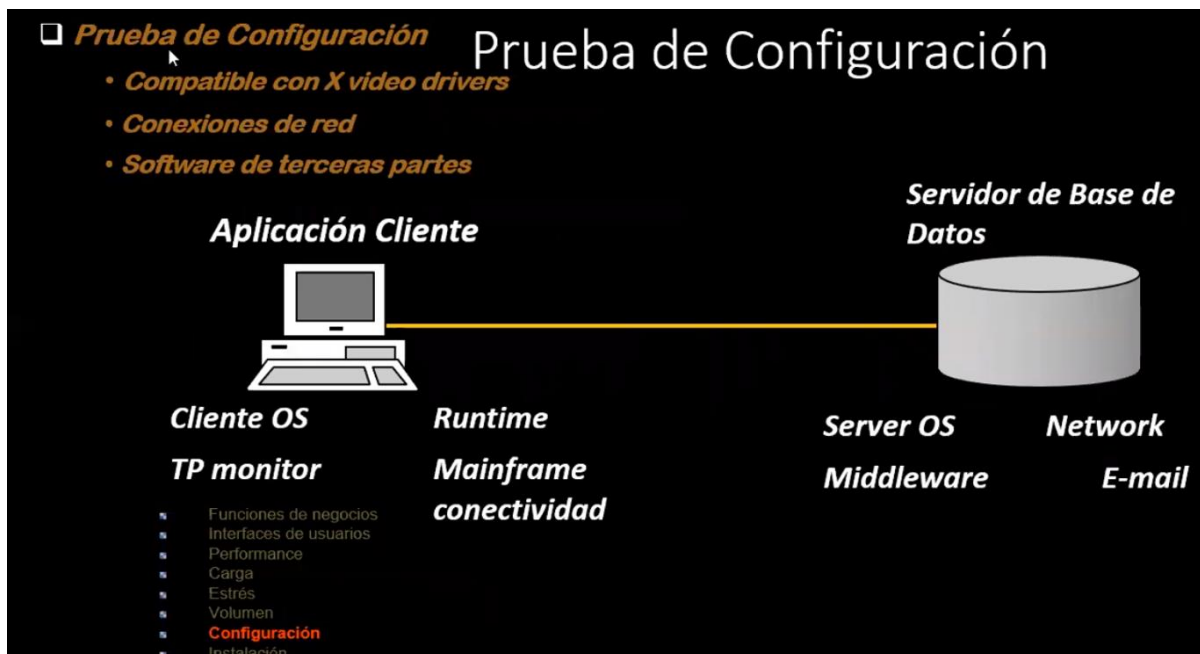
IV. Pruebas de Performance

Se analizan:

- Tiempos de Respuesta
- Concurrencia

V. Prueba de Configuración

Tiene que ver con hacer la prueba de mi entorno con una determinada configuración de cada componente (del cliente, del servidor e incluso con las redes)



ESTRATEGIA DE CAJA NEGRA:

No se conoce la estructura interna de la implementación, y solo se analiza en términos de entradas y salidas.

Se identifican cuales son las entradas que la implementación puede tener, se eligen los valores por los cuales se va a ejecutar esa funcionalidad y finalmente se comparan las salidas obtenidas con lo que se esperaba.

Tipos de Métodos:

I. Basado en especificaciones:

- a. Partición de equivalencias
- b. Análisis de valores limites

II. Basados en la experiencia

- a. Adivinanza de defectos
- b. Testing exploratorio

I. a) Partición de Equivalencias:

Lo que hace es analizar primero cuales son las diferentes condiciones externas (entradas y salidas) que van a estar involucradas en el desarrollo de una funcionalidad.

Es decir, me planto ante una funcionalidad e identifico cuales son todas las entradas y salidas posibles.

Ejemplos de entradas: campo de texto, combo de selección, coordenadas en las que me encuentro actualmente, sensor que mide temperatura, etc.

Ejemplos de Salidas: mensaje en pantalla, luz en un control remoto, emisión de un mensaje a través de FM, etc.

Consiste en dos pasos:

- 1) Identificar las clases de equivalencia (Validas y No Validas)
 - Rango de valores continuos
 - Valores Discretos
 - Selección Multiple
 - Selección Simple

2) Identificar los casos de prueba

Clase de equivalencia / Partición de equivalencia: son subconjuntos que cualquier valor que tome de ese subconjunto ante la ejecución de una funcionalidad, produce un resultado equivalente.

Es un subconjunto de valores que puede tomar una condición externa para el cual si yo tomo cualquier miembro de ese subconjunto el resultado en la ejecución de la funcionalidad es equivalente (no igual).



Componentes del Caso de Prueba:

Id del Caso de Prueba	Prioridad (Alta, Media, Baja)	Nombre del Caso de Prueba	Precondiciones	Pasos	Resultado esperado

Id del caso de prueba: nro que identifica unívocamente al caso de prueba

Prioridad del Caso de Prueba: se basa en que si tenemos que seleccionar casos de prueba para ejecutar antes que otros, se priorizan aquellos que permitan encontrar la mayor cantidad de defectos y que garanticen la salud de la funcionalidad

- **Alta:** se encuentran siempre los caminos felices.
- **Media:** todo lo que está al medio de ambos, combinaciones de valores que puedan ejecutarse bajo ciertas condiciones que producen alguna falla o alguna condición no feliz.
- **Baja:** están relacionados con aquellos casos de prueba que tienen que ver con validaciones, valores no ingresados, valores que no corresponden con formatos esperados, etc.

Nombre del Caso de Prueba: es un nombre que represente el escenario por el cual se va a estar ejecutando el caso de prueba

Precondiciones: es el conjunto de valores o características que tiene que tener el contexto para que se pueda llevar adelante el caso de prueba.

Ej.: si se requiere que un usuario este logueado o que tenga ciertos permisos para ejecutar una funcionalidad, se haría lo siguiente:

“El usuario “juan” esta logueado con permisos de administrador”

O si se depende del día en el que se ejecuta el caso de prueba, por un evento que ocurre en tal fecha, se escribiría lo siguiente:

“La fecha actual es: 15/05/2020”

Pasos: un conjunto de operaciones ordenadas y numeradas en donde el formato de redacción es:

1. **“El ROL ingresa “18” en el campo “edad””**

Resultado esperado: por ejemplo, “Se muestra el mensaje “bienvenido al sitio web””

I. b) Análisis de Valores Limites:

Es una variante de la partición de equivalencias, en vez de seleccionar cualquier elemento como representativo de una clase de equivalencia, se seleccionan los bordes de una clase.

Se basa en la premisa de que la mayor cantidad de defectos se encuentran en los límites de las clases de equivalencia.

ESTRATEGIA DE CAJA BLANCA:

Permiten diseñar casos de prueba, maximizando la cantidad de defectos encontrados con la menor cantidad de casos de prueba posibles, disponiendo de los detalles de la implementación.

Se basan en el análisis de la estructura interna del SW o un componente del SW.

Se puede garantizar el testing coverage

Coberturas:

- Cobertura de enunciados o caminos básicos
- Cobertura de sentencias
- Cobertura de decisión
- Cobertura de condición
- Cobertura de decisión/condición
- Cobertura múltiple
- etc.

Cobertura es la forma de recorrer distintos caminos que nuestro código nos provea para desarrollar una funcionalidad, cada ramificación que se presente va a representar diferentes caminos a recorrer.

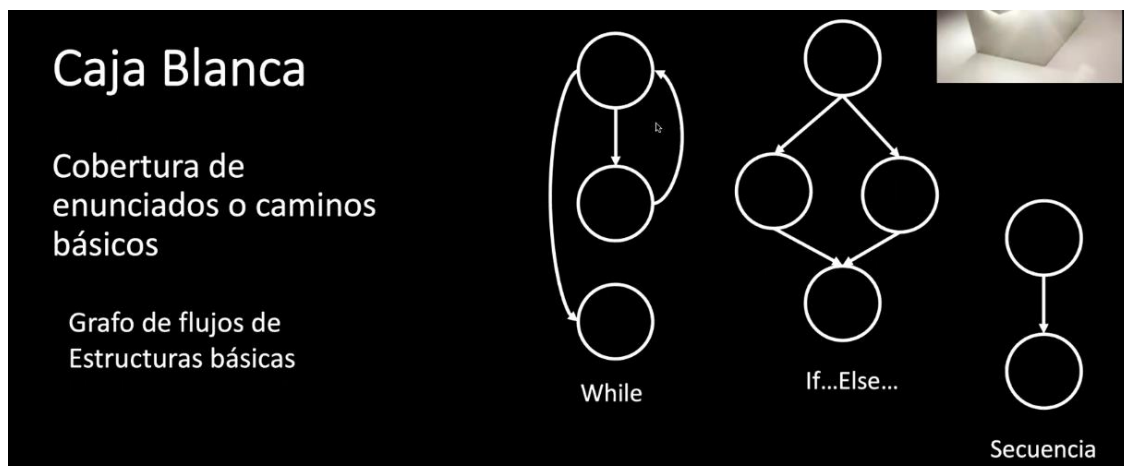
Cobertura de enunciados o caminos básicos:

Busca garantizar que a todos los caminos independientes que tiene la funcionalidad lo vamos a recorrer aunque sea una vez.

Permite obtener una medida de la complejidad de un diseño procedimental, y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución.

Para la prueba del camino básico:

- Se requiere poder representar la ejecución mediante grafos de flujo
- Se calcula la complejidad ciclomatica
- Dado un grafo de flujo se pueden generar los casos de prueba



Complejidad Ciclomática:

Coincide con la cantidad de ***caminos independientes***, es una métrica de SW que provee una medición cuantitativa de la complejidad lógica de un programa.

Usada en el contexto de testing, define el numero de caminos independientes en el conjunto básico y entrega un limite inferior para el numero de casos necesarios para ejecutar todas las instrucciones al menos una vez.

Formula:

$$\begin{aligned} M &= \text{Complejidad ciclomática.} \\ E &= \text{Número de aristas del grafo} \\ N &= \text{Número de nodos del grafo} \\ P &= \text{Número de componentes conexos,} \\ &\quad \text{nodos de salida} \\ M &= E - N + 2 * P \end{aligned}$$

O, viendo el grafo:

$$M = \text{Número de regiones} + 1$$

El numero "M" va a indicar la cantidad mínima de casos de prueba que se deberían hacer para alcanzar la cobertura de caminos básicos, que a su vez coincide con la cantidad de caminos independientes.

Heurística de la Complejidad Ciclomática:

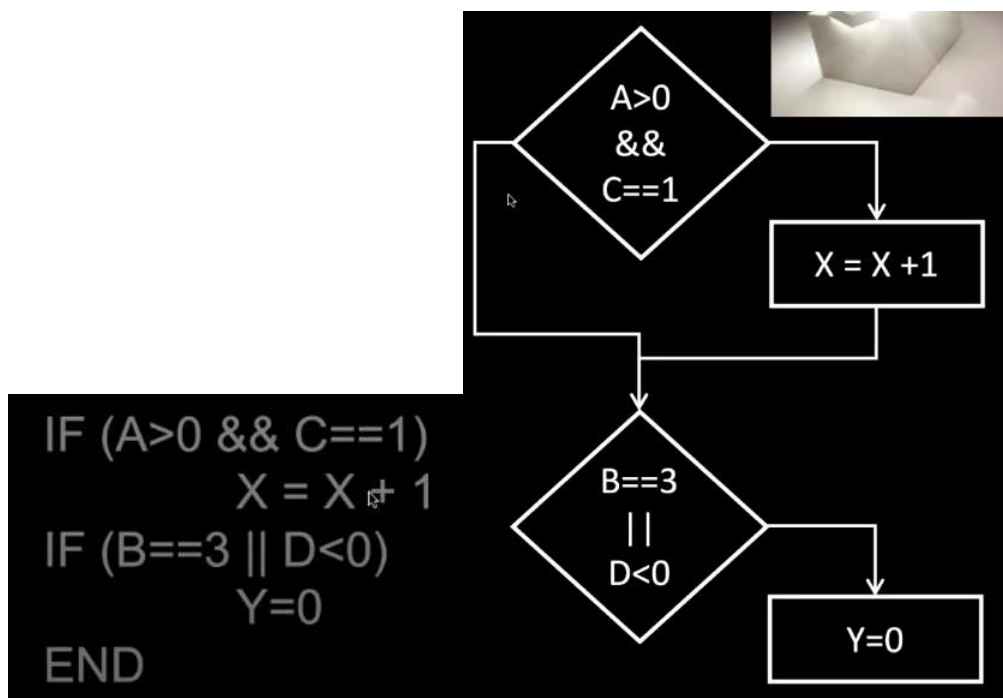
Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

Pasos del diseño de pruebas mediante el camino básico:

- Obtener el grafo de flujo
- Obtener la complejidad ciclomatica del grafo
- Definir el conjunto básicos de caminos independientes
- Determinar los casos de prueba que permitan la ejecución de cada uno de los caminos anteriores
- Ejecutar cada caso de prueba y comprobar que los resultados son los esperados.

Cobertura de Sentencias:

Este ejemplo se utiliza, junto con la representación de pseudocódigo en diagrama de flujo:



Una **sentencia** es cualquier instrucción que veamos, ya sea una asignación de variables o una invocación de métodos, mostrar un mensaje, lanzar una excepción, etc. (mientras que no sean estructuras de control)

También una sentencia puede representar el estado de un pedido, la selección de una ciudad, la existencia de una patente...

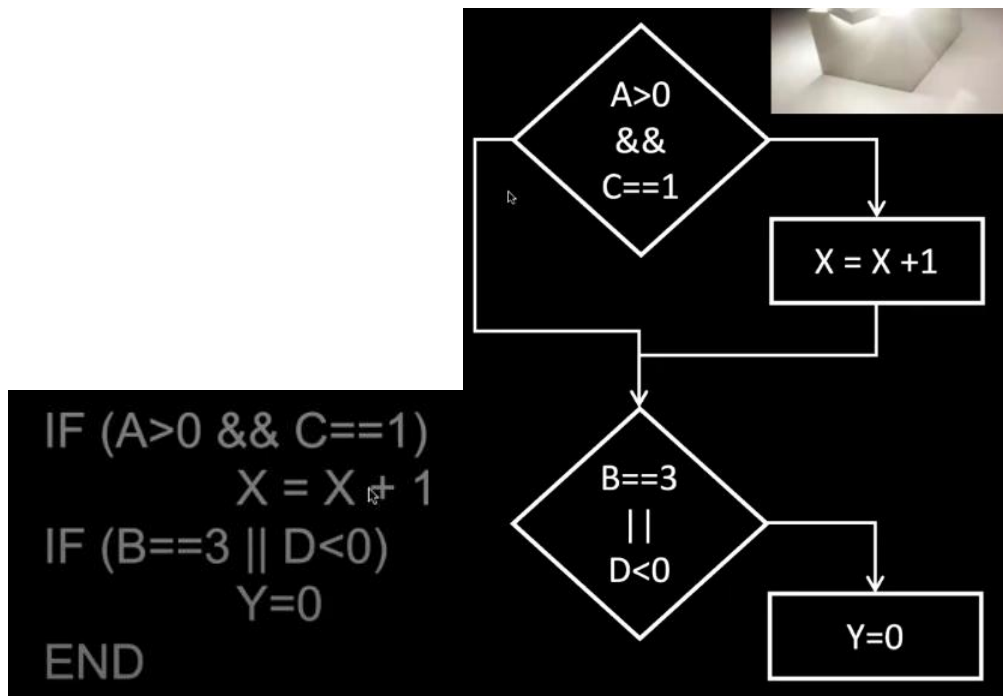
En este ejemplo, tenemos 2 sentencias ($Y = 0$; $X = X + 1$).

El **objetivo de la cobertura de sentencias** es encontrar la cantidad mínima de casos de prueba que permitan pasar o ejecutar o evaluar todas las sentencias.

En este caso, con un solo caso de prueba sería suficiente, de manera que recorra el camino que pase por ambas sentencias.

Cobertura de Decisión:

Se toma el mismo ejemplo que el anterior:



Una **decisión** es una estructura de control completa, cada una de las estructuras de control (los rombos) va a llevarnos por cada uno de los caminos dependiendo del resultado lógico de la estructura de control (en código serían los if), y puede haber estructuras de control anidadas (if anidados).

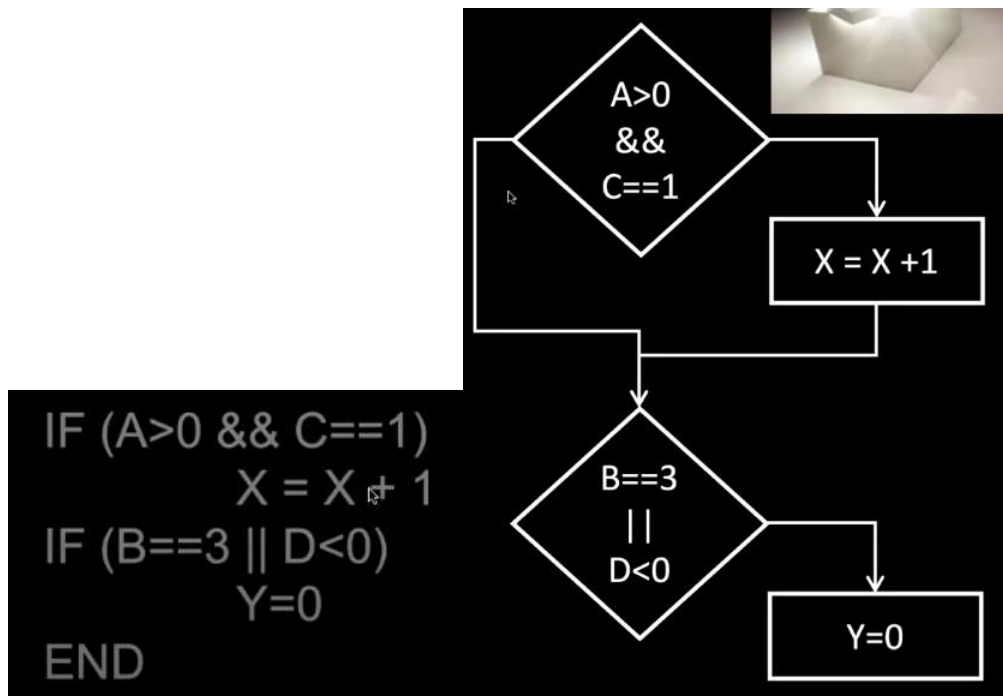
El **objetivo de la cobertura de decisión** es encontrar la cantidad mínima de casos de prueba que permitan probar todas las decisiones que tengo en el código, es decir, que recorra tanto el camino “verdadero” y el camino “falso”. Se trata de forzar la rama del true y la rama del false para comprobar que las decisiones funcionan bien.

En este ejemplo, con 2 casos de prueba serían suficientes de manera de probar todas las decisiones que se tienen en el código.

Funciona con 2 porque ambos rombos no están anidados, por lo que con solo 2 caminos se puede probar, por ejemplo, que pase por *true* en el primero y *true* en el segundo, y que pase por *false* en el primero y por *false* en el segundo, y allí se habrían analizado todas las decisiones.

Cobertura de Condición:

Utilizando el mismo ejemplo:



Una **condición** es cada una de las evaluaciones lógicas que están unidas por operadores lógicos que están dentro de una decisión.

Ej: `A>0 ; C==1 ; B==3 ; D<0...` **no se considera:** “&&” o “||”

El **objetivo de la cobertura de condición** busca encontrar la cantidad mínima de casos de prueba que permitan valuar cada una de las condiciones, tanto en su verdadero como en su valor falso, **independientemente** de por donde salga la decisión.

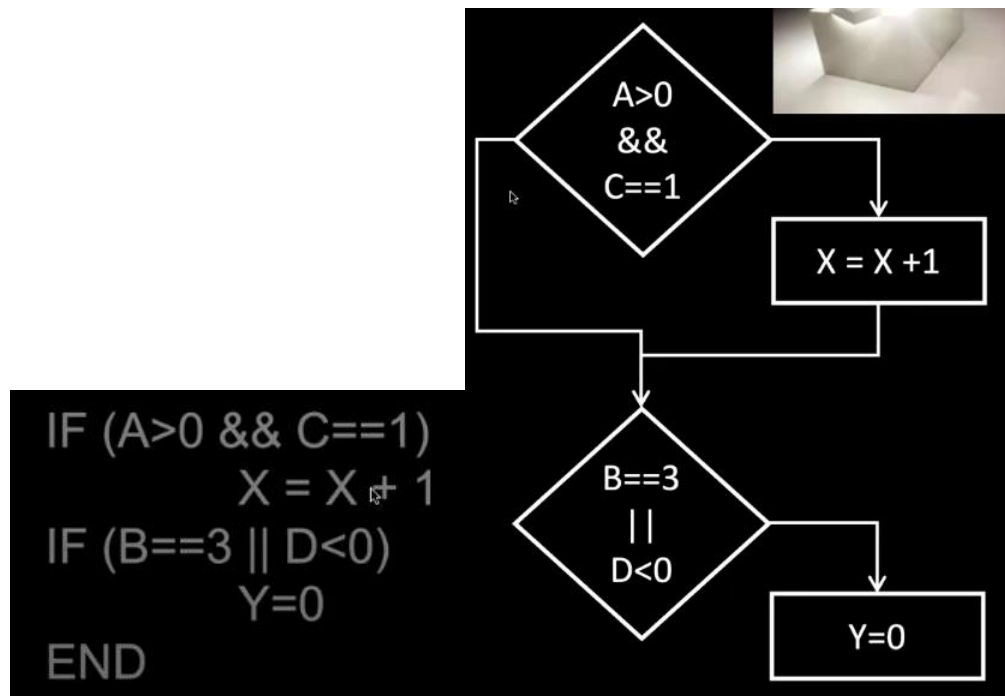
En este ejemplo, se necesitarían 2 casos de prueba para probar todas las condiciones...

Cobertura conjunta de decisión y condición:

Lo que busca es valuar no solamente las decisiones en su valor verdadero y valor falso, sino que también busca valuar las condiciones tanto en su valor verdadero como en su valor falso.

Cobertura múltiple:

Utilizando el mismo ejemplo:



El **objetivo de la cobertura múltiple** es encontrar la cantidad mínima de casos de prueba que permitan valuar el combinatorio de todas las **condiciones** en todos sus valores de verdad posibles.

En este caso, tenemos 4 condiciones, por lo que para la primera decisión tenemos:

Cuatro combinatorios para todos los valores de verdad de esas dos condiciones

Y para la segunda decisión:

También cuatro combinatorios

VV

VF

FV

FF

Por lo que, la cantidad mínima de casos de prueba necesarios para valuar todos los combinatorios son 4.

ASEGURAMIENTO DE CALIDAD DE PROCESO Y DE PRODUCTO

Cobertura del PPQA



Calidad:

Es un concepto abstracto, y en gran medida tiene demasiado que ver con la percepción o la mirada que tienen los distintos actores que miran o trabajan sobre esa calidad.

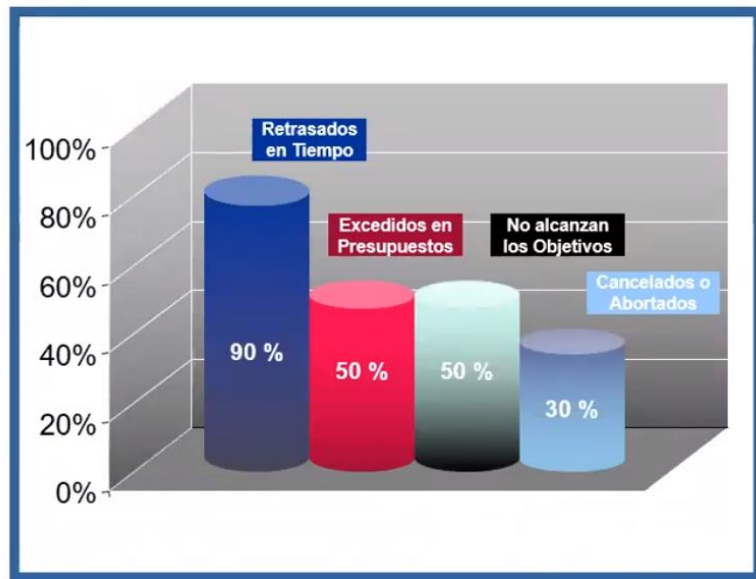
Tiene que ver con todas las características, features o con aspectos de un producto o servicio, que se relacionan con su habilidad de alcanzar las necesidades manifiestas o implícitas.

La calidad es **relativa** a las personas, para un mismo observador una calidad no es única en el tiempo (puede cambiar con el tiempo).

Cosas que ocurren frecuentemente en los proyectos

- Atrasos en las entregas
- Costos Excedidos
- Falta de cumplimiento en los compromisos
- No están claros los requerimientos
- El software no hace lo que tiene que hacer
- Trabajo fuera de hora
- Fenómeno del 90-90
- ¿Dónde está este componente?

Situación de Proyectos de Software



Un software es de calidad cuando satisface:

- Las expectativas del Cliente
- Las expectativas del Usuario
- Las necesidades de la Gerencia
- Las necesidades del equipo de desarrollo y mantenimiento
- Otros interesados...

No se debe satisfacer solo las expectativas del usuario, no es suficiente.

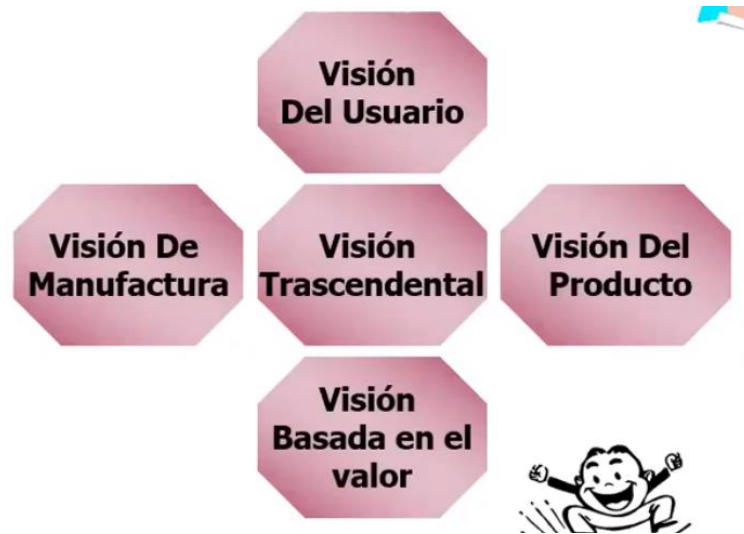
Principios de la Calidad:

- **La calidad no se “inyecta” ni se compra, debe estar embebida:** hacer más testing no agrega calidad al producto.
- **Es un esfuerzo de todos**
- **Las personas son la clave para lograrlo:**
 - **Capacitación:** es clave este aspecto para que las personas logren el aseguramiento de la calidad
- **Se necesita sponsor a nivel gerencial**
 - **Pero se puede empezar por uno**
- **Se debe liderar con el ejemplo**
- **No se puede controlar lo que no se mide**
- **Simplicidad, empezar por lo básico.**

- **El aseguramiento de la calidad debe planificarse.**
- **El aumento de las pruebas no aumenta la calidad**
- *Debe ser razonable para mi negocio.*

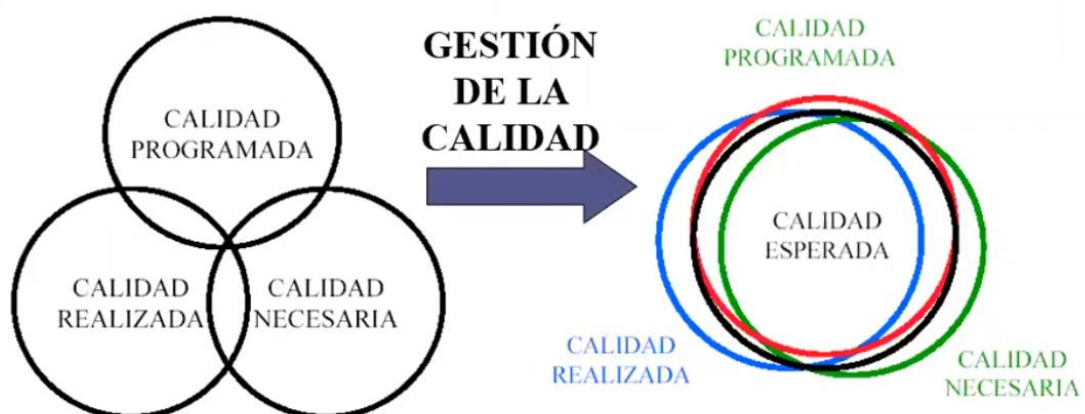
¿Calidad para quién?:

Se habla en el contexto de las *visiones*, cuando se habla de visión se analizan distintos aspectos.



También con respecto a las visiones se analizan diferentes aspectos más allá de quien sea el que posea esa visión.

Se plantea la calidad sobre tres ejes:

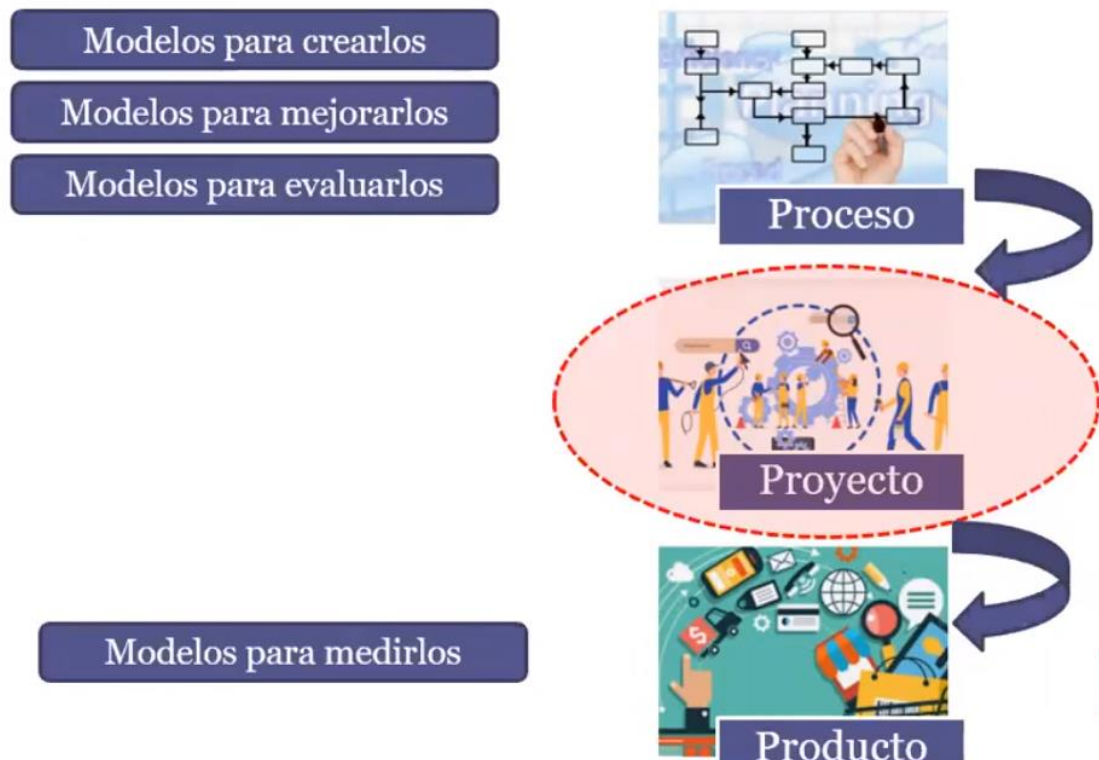


- **Calidad Programada:** es la que espero que tenga el producto (como equipo de desarrollo)
- **Calidad Necesaria:** calidad mínima que debería tener el producto de SW

- **Calidad Realizada:** es lo que realmente ocurrió

Lo que se busca con la gestión de calidad es la intersección entre esos tres ejes o conceptos, cuanto mas lejos se encuentren los tres ejes más dificultades se van a presentar

Calidad en el Software ¿Cómo hacemos para trabajar acerca de la calidad del Software?:



1. Se define un **proceso** o trabajamos con procesos empíricos
 - a. La **motivación** para definir el proceso, o la teoría sobre por qué definir un proceso, es que se supone que *“si se define un proceso, y el mismo esta bien definido, cuando se ejecuta un proyecto y se respetan los pasos del proceso correctamente, el producto a obtener **va a ser de calidad**”*
 - b. **En los procesos empíricos** eso cambia (siempre se trabaja con un proceso), debido a que se trabaja de manera diferente (el proceso va evolucionando, cambia según la experiencia de las personas). **Lo que hace que construya un producto de calidad** es la experiencia de las personas, o con las personas adecuadas

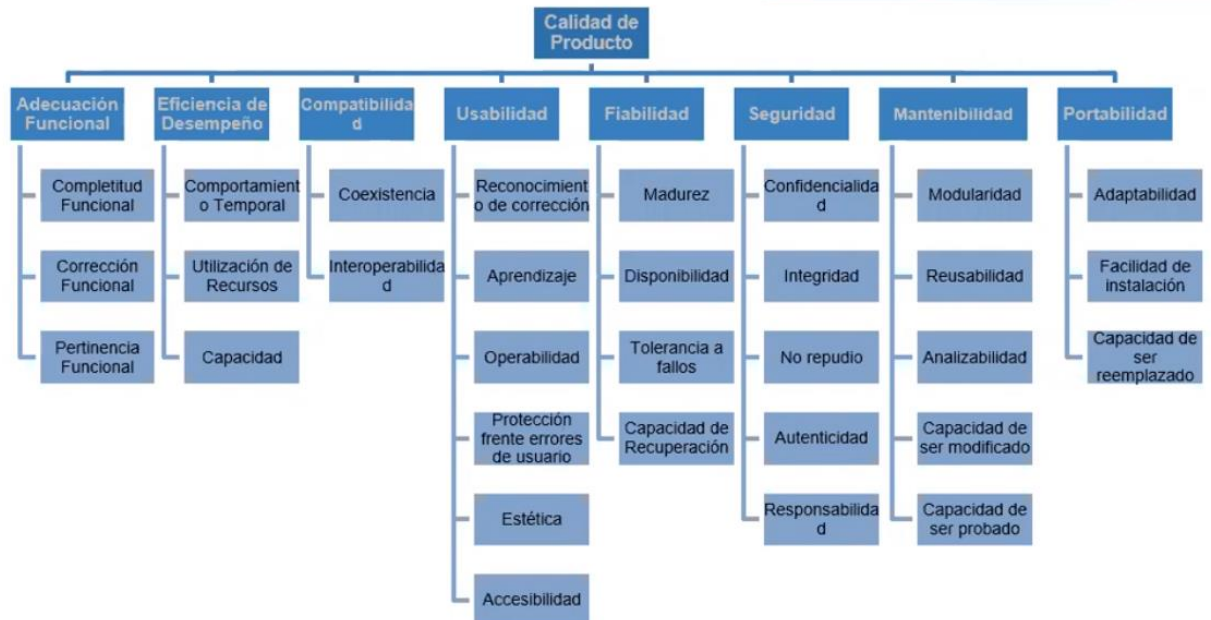
(en realidad no es algo solo de los procesos empíricos, también puede pasar para procesos definidos)

- c. Si o si para analizar que tan bueno es un proyecto o que tan bien esta hecho, hay que instanciarlo en un proyecto para analizar resultados y en todo caso modificarlo y mejorarlo.
2. A partir del proceso definido, se instancia en el **proyecto** (el proceso cobra vida a partir del proyecto)
- a. Cuando se ejecuta el proyecto, además de las actividades de Ingeniería, se van a tener que ejecutar otras actividades de aseguramiento de calidad (con técnicas y herramientas tales como “Revisiones Técnicas” y “Auditorias”)
3. A partir del proyecto se crea en términos concretos el **producto**.
- a. También se pueden ejecutar tareas de aseguramiento de calidad, utilizando técnicas y herramientas tales como “Revisiones Técnicas”, “Auditorias (de configuración)” y “Testing”. La diferencia con las técnicas mencionadas en el proceso es que esas técnicas están orientadas para chequear como se esta ejecutando el *proyecto* en términos del *proceso*. En cambio, estas técnicas verifican que el producto que se esta construyendo en ese proyecto cumpla con los estándares definidos.

Calidad de Producto

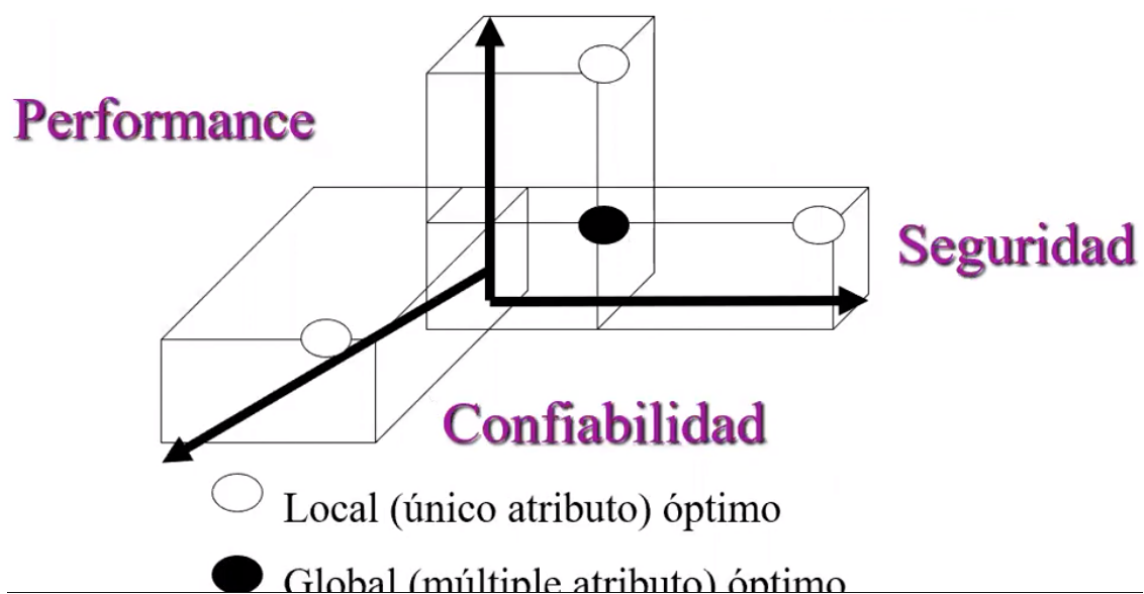
Modelos respecto a la calidad del producto:

- **Calidad en Uso (ISO 25010 : 2011)**

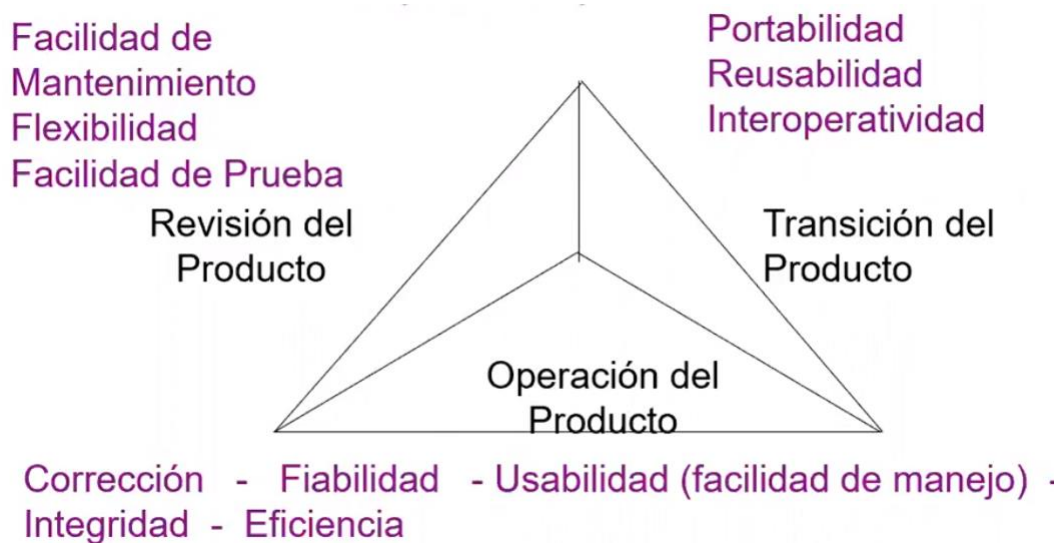


Modelo de Calidad de Producto (En uso) – ISO 25010

- Calidad del producto (ISO 25010: 2011)
- Calidad de Datos (ISO 25012: 2008)
- **Modelo de Barbacci / SEI**



- **Modelo de MCALL**



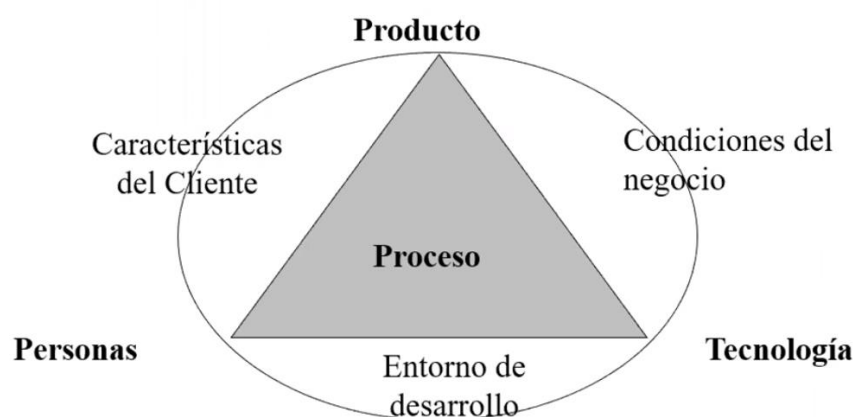
El punto es que, independientemente del modelo elegido para comparar y determinar la calidad del producto, hay un aspecto que no se puede materializar, y tiene que ver con que si el producto cumple con los requisitos del usuario o del cliente (si no cumple con los requisitos, evidentemente no va a ser de calidad)

Pero sirven de referencia.

Calidad en el Proceso de Desarrollo:

Es un aspecto fundamental, debido a que en lo único que podemos influir o controlar en el contexto de la organización, es sobre el proceso. Por ende se concentran en diferentes aspectos del aseguramiento de la calidad sobre el proceso

El proceso es el único factor <<controlable>> al mejorar la calidad del software y su rendimiento como organización

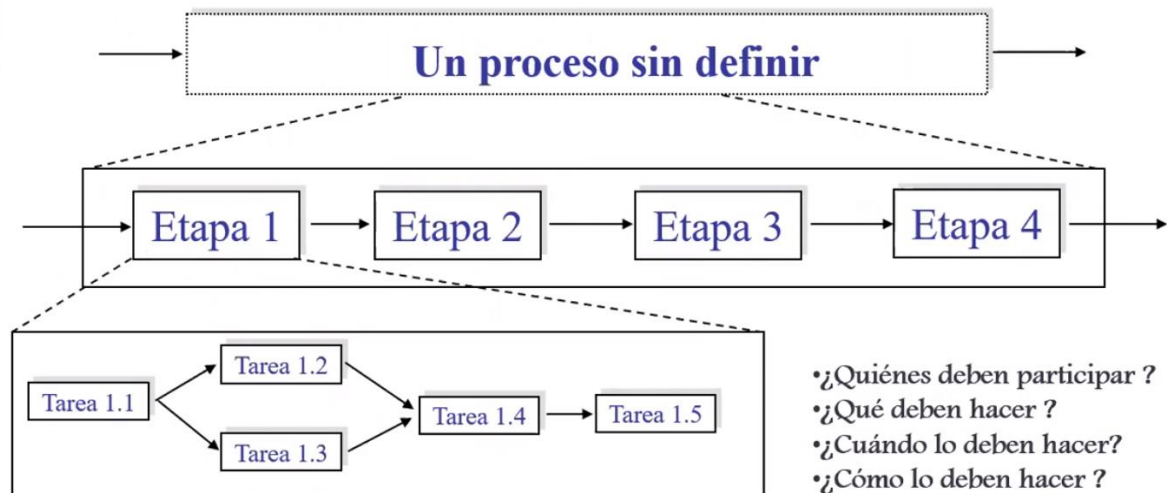


Proceso sin definir:

Lo primero que se tiene que hacer es definir un proceso si es que no lo tengo, porque no se puede asegurar calidad sin tener un proceso.

Definición de Proceso de SW:

Se deben definir, cuales son las etapas que se ejecutan y cuales son las tareas, quienes tienen que participar, en que momento, como tienen que participar, cuales son sus responsabilidades, y que herramientas deben utilizar.



Se obtendría algo como:



Las disciplinas que aparecen al final son las que nos permiten trabajar en el aseguramiento de la calidad. Por ende, tienen que estar definidas dentro del proceso.

Aseguramiento de la Calidad

Se busca alcanzar los requerimientos de la calidad del producto de SW. Para saber si se alcanzaron, se debe tener un estándar definido y comparar con el mismo para saber si se alcanzó o no.

- Concerniente con asegurar que se alcancen los niveles requeridos de calidad para el producto de software.
- Implica la definición de estándares y procesos de calidad apropiados y asegurar que los mismos sean respetados.
- Debería ayudar a desarrollar una “**cultura de calidad**” donde la calidad es vista como una **responsabilidad de todos y cada uno**.

Todos estos conceptos de aseguramiento de calidad tienen sentido en el contexto de **procesos definidos**.

En metodologías ágiles, en SCRUM por ejemplo, en la Retrospective se evalúa como se viene trabajando y a partir de ahí se hacen modificaciones para asegurar la calidad.

Reporte del Grupo de Aseguramiento de Calidad (GAC)

- No debería reportar al Gerente de Proyectos
- No debería haber más de una posición entre la Gerencia de Primer Nivel y el GAC
- Cuando sea posible, el GAC debería reportar alguien realmente interesado en la calidad del software

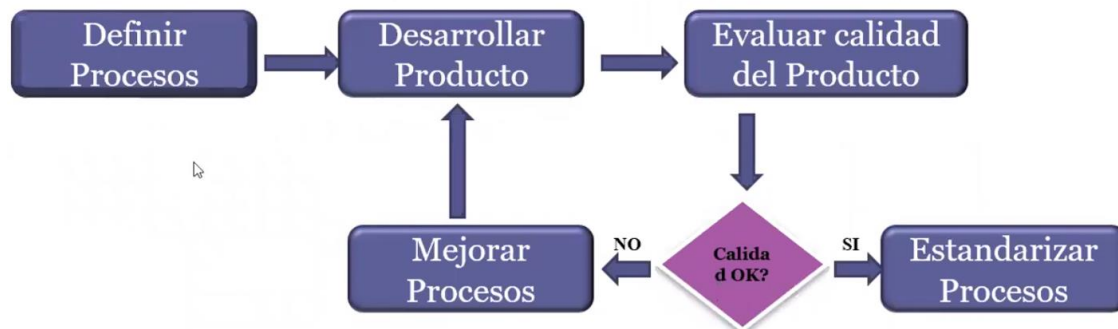
La administración de calidad debería estar separada de la Administración de Proyectos para asegurar independencia. Recordando que esto es solo para procesos definidos, en procesos empíricos esto funciona de manera diferente.

Actividades de la Administración de Calidad de Software:

- **Aseguramiento de Calidad**
 - Establecer estándares y procedimientos organizacionales de calidad.
- **Planificación de Calidad**
 - Selecciona los procedimientos y estándares aplicables para un proyecto en particular y los modifica si fuera necesario
- **Control de Calidad**
 - Asegura que los procedimientos y estándares sean respetados

Estas actividades de aseguramiento de calidad tienen que estar embebidas en el proceso de software (no se puede inyectar la calidad al final).

¿Como funcionaria eso?



En el agilismo, no serviría estandarizar un proceso.

En la Práctica, se materializa de la siguiente manera:

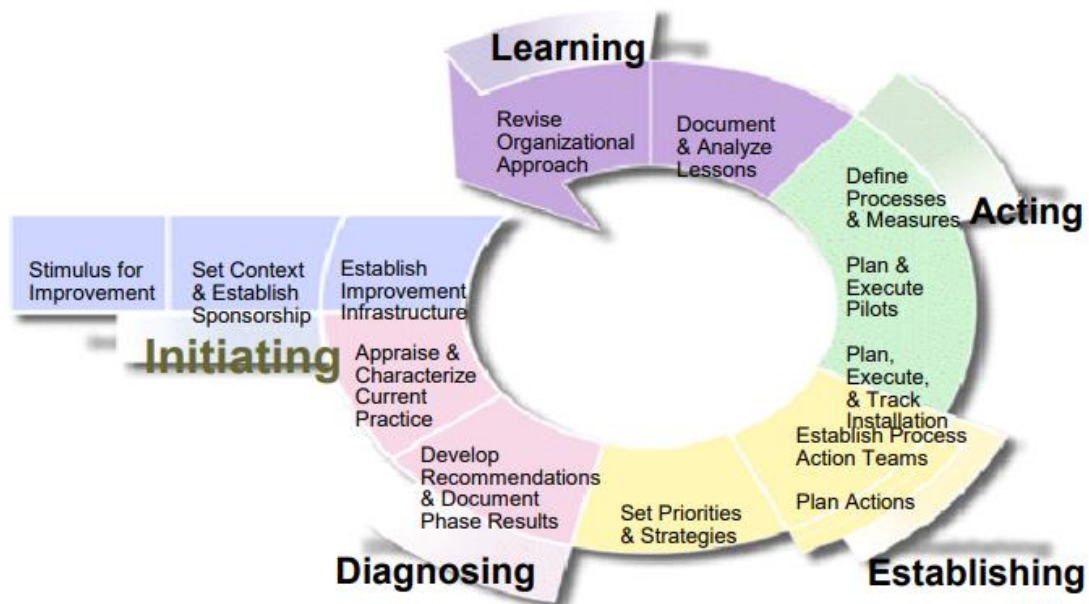
- **Definir procesos estándares tales como:**
 - Cómo deberían conducirse revisiones
 - Cómo debería realizarse la administración de configuración, etc.
- Monitorear el proceso de desarrollo para asegurar que los estándares sean respetados.
- Reportar en el proceso a la Administración de Proyectos y al responsable del software.
- No use prácticas inapropiadas simplemente porque se han establecido los estándares.

Y sobre todo, definir estándares (esto funciona comparando):

- ❖ Los estándares son la clave para la administración de calidad efectiva.
- ❖ Pueden ser estándares internacionales, nacionales, organizacionales o de proyecto.
- ❖ **Estándares de Producto** definen las características que todos componentes deberían exhibir, ej. estilos de programación común.
- ❖ **Estándares de Procesos** definen cómo deberían ser implementados los procesos de software.

Modelos de Mejora de Procesos:

- **SPICE:** Software Process Improvement Capability Evaluation.
- **IDEAL:** Initiating, Diagnosing, Establishing, Acting, Leveraging



Es un modelo que se utiliza cuando no hay un proceso definido o hay que definir un proceso.

Se hace con distintas actividades o etapas, primero **se busca tener sponsor** que indique que es necesario definir un nuevo proceso, luego **se hace un diagnóstico** si hubo un proceso definido previamente, **luego se define el plan de acción, se ejecuta, se mide** y por último en función de esas medidas **se adecua el proceso** o se deja como un proceso definido para la organización

Filosofía Lean:

En términos conceptuales, cuando hablamos de *lean* no hablamos de proyectos, sino de un trabajo continuo.

Principios:

La filosofía lean tiene 7 principios:

- I. Eliminar desperdicios
- II. Amplificar Aprendizaje
- III. Embeber la Integridad conceptual
- IV. Diferir compromisos
- V. Dar poder al equipo
- VI. Ver el todo
- VII. Entregar lo antes posible



Están fuertemente relacionados a los *principios agile*

I. Eliminar el desperdicio:

Con respecto a la producción, está relacionado con el *just in time*, y esta relacionado con tener la menor cantidad de stock (debido al costo de mantenimiento, infraestructura, etc.)

Con respecto al software, la característica fundamental tiene que ver con el software funcionando es lo que da valor y “*el arte de maximizar lo que no hacemos*”, es decir, optimizar las tareas que hacemos para evitar retrabajo o evitar reconstruir cosas.

En donde mas se ve es en la parte de **requerimientos** (para que me pongo a trabajar de antemano en los requerimientos si probablemente los mismos cambien en el futuro).

❖ **Eliminar Desperdicios:** reducir el tiempo removiendo lo que no agrega valor.

- Desperdicio es cualquier cosa que interfiere con darle al cliente lo que valora en tiempo y lugar donde le provea más valor.
- *En manufactura:* el inventario
- *En software:* es el trabajo parcialmente hecho y las características extra
- El 20% del software que entregamos contiene el 80% del valor

Se trata de lograr que al construir el software sea realmente lo que tenga sentido para el cliente.

Gastos en producción Lean (en términos de producción):



Los siete desperdicios de Lean (en software):

- Características extra
- Trabajo a medias
- Proceso extra
- Movimiento
- Defectos
- Esperas
- Cambio de Tareas

II. Amplificar el aprendizaje:

Se relaciona a la autoorganización del equipo, que el equipo tenga las habilidades para poder evolucionar, con respecto al aprendizaje de procesos y aprendizaje técnico. Se relaciona con que el mismo equipo es el que genera el mismo conocimiento espiralado para poder aprender y compartir con el equipo el nuevo conocimiento que va surgiendo.

- **Amplificar el aprendizaje:** crear y mantener una cultura de mejoramiento continuo y solución de problemas.
 - Un proceso focalizado en crear conocimiento esperará que el diseño evolucione durante la codificación y no perderá tiempo definiéndolo en forma completa, prematuramente.
 - Se debe generar nuevo conocimiento y codificarlo de manera tal que sea accesible a toda la organización.
 - Muchas veces los procesos “estándares” hacen difícil introducir en ellos mejoras.

III. Embeber la Integridad conceptual:

Esta relacionado con que la integración entre las personas hacen que el producto en sí mismo en términos arquitectónicos y en términos de producto emergente sea más consolidado. Esta pensado en términos de coherencia y de consistencia desde el principio y a lo largo de toda la construcción del producto.

- **Embeber la integridad conceptual:** Encastrar todas las partes del producto o servicio, que tenga coherencia y consistencia (tiene que ver con los Requerimientos No Funcionales). La integración entre las personas hace el producto más integro.

Se refiere en gran medida a la arquitectura, que sea robusta, sustentable, etc.

- Se necesita más disciplina no menos!
- **Integridad Percibida:** el producto total tiene un balance entre función, uso, confiabilidad y economía que le gusta a la gente.
- **Integridad Conceptual:** todos los componentes del sistema trabajan en forma coherente en conjunto.
- El objetivo es construir con calidad desde el principio, no probar después.
- **Dos clases de inspecciones:**
 - Inspecciones luego de que los defectos ocurren.
 - Inspecciones para prevenir defectos.
- Si se quiere calidad no inspeccione después de los hechos!
- Si no es posible, inspeccione luego de pasos pequeños.

IV. Diferir los compromisos:

Tiene que ver con decidir lo mas tarde posible pero de manera responsable.

Se intenta tomar las decisiones lo mas tarde posible dentro del contexto en el que estoy parado (dependiendo del caso probablemente no tengamos opción).

- **Diferir Compromisos.** El último momento responsable para tomar decisiones (en el cual todavía estamos a tiempo). Si nos anticipamos tenemos información parcial.

Se relaciona con el principio ágil: decidir lo más tarde posible pero responsablemente. No hacer trabajo que no va a ser utilizado. Enlaza con el principio anterior de aprendizaje continuo, mientras más tarde decidimos más conocimiento tenemos.

- Las decisiones deben tomarse en el último momento que sea posible.
 - No significa que todas las decisiones deben diferirse.
 - Se debe tratar de tomar **decisiones reversibles**, de forma tal que pueda ser fácilmente modificable.
 - Vencer la “parálisis del análisis” para obtener algo concreto terminado.
 - Las mejores estrategias de diseño de software están basadas en **dejar abiertas opciones** de forma tal que las decisiones irreversibles se tomen lo más tarde posible.

V. Dar poder al equipo:

El equipo tiene un peso importante, puede ser autoorganizado, no hay una persona que asigna tareas sino que cada persona toma sus decisiones, y trabaja en las responsabilidades de manera colaborativa.

- **Dar poder al equipo:** ejemplo, vamos a comer a un restaurante y no nos metemos en la cocina del restaurante. Nos fijamos en el precio, pedimos y esperamos. Hay mucho micro management, el dueño no decide cuánta sal poner a la comida.
- **Respetar a la gente**
 - Entrenar líderes
 - Fomentar buena ética laboral
 - Delegar decisiones y responsabilidades del producto en desarrollo al nivel más bajo posible
- **Ágil:** El propio equipo pueda estimar el trabajo.

VI. Ver el todo:

Ver el producto en su conjunto, esta relacionado con el concepto de la arquitectura, trata de poder tener la visión de hacia donde va nuestro producto (el valor agregado que el producto va a ofrecer a lo largo del tiempo)

- **Ver el todo:** tener una visión holística, de conjunto (el producto, el valor agregado que hay detrás, el servicio que tiene los productos como complemento).

VII. Entregar rápido:

Los incrementos pequeños y de valor crea una mejor relación con nuestro cliente. No entregar rápidamente va en contra del principio de “Eliminar desperdicios”.

- **Entregar rápido:** estabilizar ambientes de trabajo a su capacidad más eficiente y acotar los ciclos de desarrollo.
- **Entregar rápidamente** esto hace que se vayan transformando “n” veces en cada iteración. Incrementos pequeños de valor. Llegar al producto mínimo que sea valioso. Salir pronto al mercado.
 - **Relacionado con el principio Ágil de entrega frecuente.**

KANBAN:

En desarrollo de software el principal referente es Kanban.

Kanban es un **framework de mejora de proceso** que materializa los principios *Lean* y principios *agile* también.

- **kan-ban:** termino japones que significa “señal”. Significa que de alguna manera voy a tener visible la señal que necesito para saber cual es el estado o en que momento se encuentra mi proceso o que tengo que hacer en el proceso.

Kanban en pocas palabras – “Just in Time”

Surge con el concepto de solo tener lo que voy a necesitar, y de esta manera reducir los costos de manera importante.

Kanban plantea como **primer aspecto fundamental** del framework al “Just in Time”. Avanzo solamente con lo que necesito.

Si se extrapola al software, **esta relacionado con el concepto fundamental de “Eliminar el desperdicio”** (con respecto a *Lean*)



Administración de Colas:

Otro concepto fundamental que surge en Kanban tiene que ver con la administración de las colas, donde se administran las distintas colas de mi proceso, que hacen cosas diferentes, haciendo foco en que en ninguna de las colas haya cuello de botella y que ninguna de las colas tenga tiempo ocioso.

Ninguna cola debería implicar un atoramiento, pero tampoco debería tener tiempo ocioso.

Ejemplo:

- Los cajeros se focalizan en tomar órdenes.
- El Barista se focaliza en proveer café.
- Separarlos por la cola permite que se absorba la demanda variable.
- Los cajeros se mueven a ayudar al Barista cuando no hay clientes esperando para hacer su pedido.
- **Foco es en Flujo “fin a fin” FLOW = Centrado en el Cliente**

The image shows a Starbucks logo at the top. Below it is a Starbucks cup with a white lid. Overlaid on the cup is a Kanban board with a vertical list of tasks: Order, Start, Mix, Brew, Pour, Compostable, and others. The board has yellow and green markers indicating the status of each task.

Principios de Kanban:

- I. **Visualizar el Flujo:** hacer el trabajo visible. (por ejemplo, en una tabla)
- II. **Limitar el Trabajo en progreso (WIP):** tiene que ver con limitar cuanto trabajo tenemos en progreso al mismo tiempo.
- III. **Administrar el flujo:** ayudar a que el trabajo fluya
- IV. **Hacer explícitas las políticas**
- V. **Mejorar colaborativamente:** trabajar en términos del equipo.

A diferencia de SCRUM, Kanban plantea en que no te pongas a redefinir todo, no se tiene una definición del proceso que deberías seguir, uno dibuja su proceso y su cadena de valor y lo que se va haciendo con ese framework es ir “**mejorando**” el proceso que uno ya tiene definido.

En términos de adopción de un framework, lo hace más fácil.

Kanban en el Desarrollo de Software:

- El método fue formulado por David J. Anderson
- Es un **enfoque para gestión de cambio**.
- **No es un proceso de desarrollo de software o una metodología de administración de proyecto.**
- Kanban es un **método para introducir cambios** en un proceso de desarrollo de software o una metodología de administración de proyectos

Kanban aprovecha muchos de los conceptos probados de Lean:

- Definiendo el Valor desde la perspectiva del Cliente.
- Limitando el Trabajo en Progreso (WIP).
- Identificando y Eliminando el Desperdicio.
- Identificando y removiendo las barreras en el Flujo.
- Cultura de Mejora Continua.

Otros Conceptos de Kanban:

Kanban **fomenta la evolución gradual** de los procesos existentes.

Kanban no pide una revolución, sino que **fomenta el cambio gradual**. No se aplican todos los cambios de una sola vez.

Kanban está basado en una idea muy simple: **Limitar el trabajo en progreso (WIP)**. Ayuda a concentrarse mas bien a una sola tarea, y evitar el multi tasking. (por ende se relaciona con el principio de “Eliminar el desperdicio”).

El Kanban (o tarjeta de señal) implica que una señal visual se produce para indicar que el nuevo trabajo se puede tirar (“pull”) porque el trabajo actual no es igual al límite acordado.

¿Cómo aplicar Kanban?

Se va de a poco, no se puede resolver todo de un momento para el otro. Se tiene que entender el proceso actual y una vez entendido, se definen los limites (es decir, el límite de los WIP).

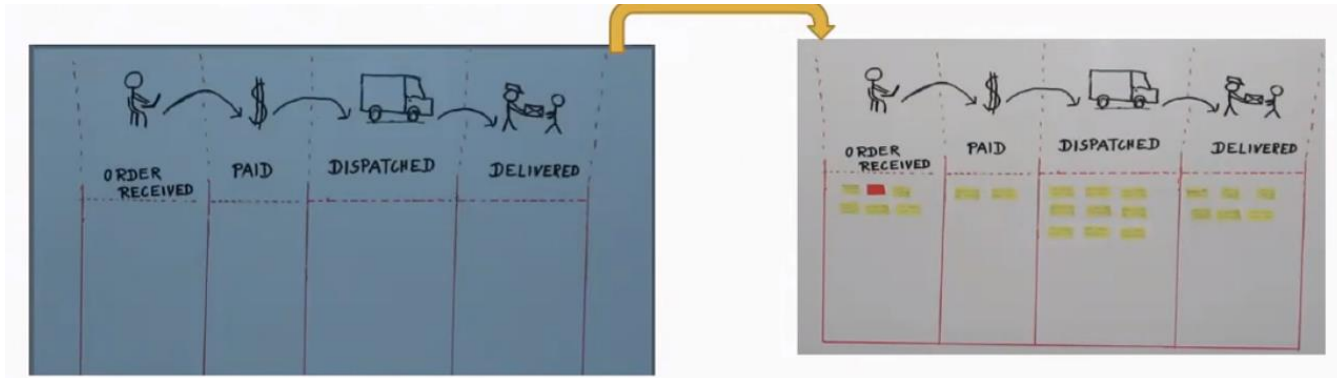
- Empezar con lo que se tiene ahora.
- Entender el proceso actual.
- Acordar los límites de WIP para cada etapa del proceso.
- A continuación, comienza a fluir el trabajo a través del sistema tirando de él, en presencia de señales Kanban.

Lo primero que se tiene que hacer es:

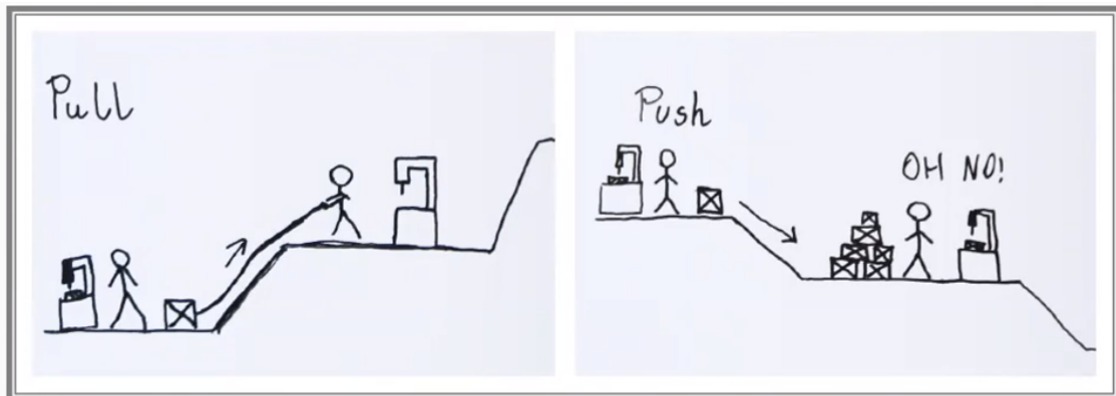
1. **Dividir el trabajo en piezas**, las user stories son buenas para eso.



2. **Se mapea el proceso:** colocando las columnas de izquierda a derecha, imaginando como fluye mi proceso. Por donde empieza y a donde termina.
- Utilizar nombres en las columnas para ilustrar donde esta cada ítem en el flujo de trabajo
 - Distribuir el trabajo en las columnas: el trabajo fluirá de izquierda a derecha en las columnas.

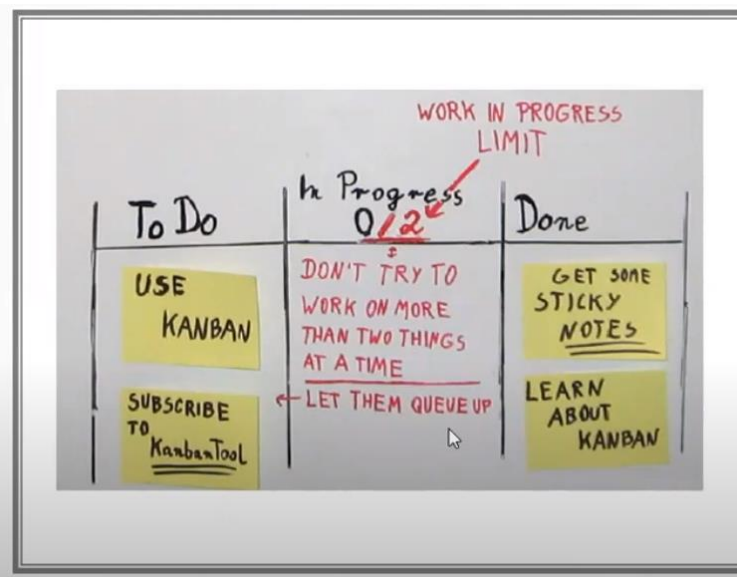


3. Para pasar de una pieza de trabajo de una columna a la otra, **se basa en el concepto de que cada uno de los integrantes del equipo toma la tarea en la que va a trabajar** (en lugar de que alguien asigna las tareas). Es una de las cosas mas complicadas de trabajar con Kanban.



Pull, no push !!!

Limitar el WIP:



¿Cómo aplicar Kanban?

- Limitar WIP – Asignar límites explícitos de cuántos ítems puede haber en progreso en cada estado del flujo de trabajo.

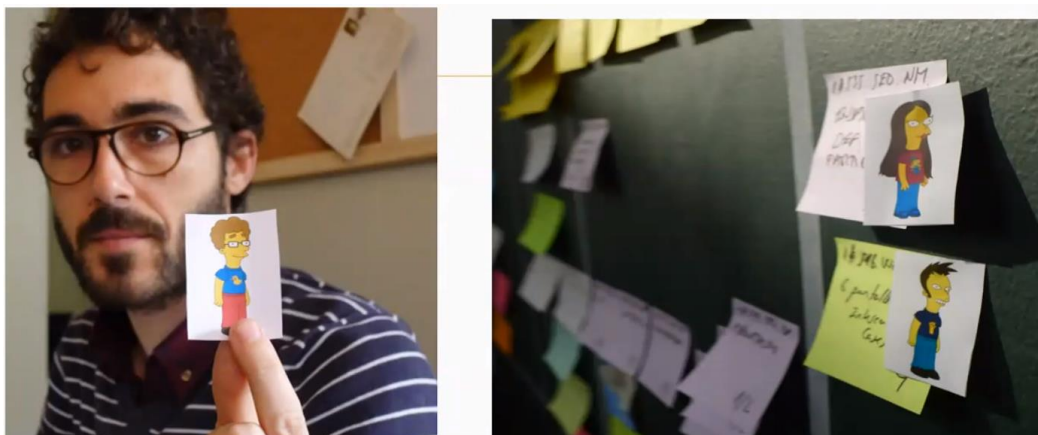
Ayudar a que el trabajo fluya:



Ayudar a que el trabajo fluya....

Al 100 % de capacidad se tiene un rendimiento mínimo...

La auto asignación de tareas se ve reflejada con un avatar personalizado:



1) Ejemplo de Definición del Proceso:

Cola de Producto	Análisis		Desarrollo		Listo para Build	En Testing		En Producción
	En progreso	Hecho	En progreso	Hecho		En Progreso	Listo para Despliegue	

A diferencia de SCRUM, en la que se habla de un único producto, en Kanban no hay una restricción de que el tablero sea para un único producto, se podrían tener mas de un producto en un mismo tablero.

2) Ejemplo de definición de Tipos de Trabajo:

Definir tipos de trabajo...

Asignando capacidad en función de la demanda

Requerimientos

- Caso de uso
- Historias de Usuario
- Porciones de Casos de Uso
- Características

Defectos

- Defectos en Producción
- Defectos

Desarrollo

- Mantenimiento
- Refactorización
- Actualización de Infraestructura

Solicitudes

- Solicitud de Cambio
- Sugerencias de Mejora

3) Ejemplo de definición de WIP:

	5	4	4	4	2	Total = 19		
Cola de Producto	Análisis		Desarrollo		Listo para Build	En Testing		En Producción
	En progreso	Hecho	En progreso	Hecho		En Progreso	Listo para Despliegue	

4) Ejemplo de Definición de Políticas Explícitas para cada clase de servicio:



Métricas Clave:

- **Lead Time = Vista del Cliente**

Representa el tiempo que transcurre desde que ingresa la pieza de trabajo hasta que esta terminada. Para el cliente la entrega de valor tiene que ver con el Lead Time.

- ❖ Es la métrica que registra el tiempo que sucede entre el inicio y el final del proceso, para un ítem de trabajo dado. Se suele medir en días de trabajo o esfuerzo.
- ❖ Medición más mecánica de la capacidad del proceso
- ❖ **Ritmo de Terminación**

- **Cycle Time = Vista Interna**

- ❖ Representa el tiempo que transcurre desde que empiezo a trabajar en la pieza de trabajo hasta que la termino.
- ❖ Es la métrica que registra el tiempo que sucede entre el momento en el cual se está pidiendo un ítem de trabajo y el momento de su entrega (el final del proceso). Se suele medir en días de trabajo.
- ❖ **Ritmo de entrega**

- **Touch Time**

Tiene que ver con el tiempo específico en el que estoy trabajando en una determinada pieza de trabajo.

- ❖ El tiempo en el cual un ítem de trabajo fue realmente trabajado (o "tocado") por el equipo.
- ❖ Cuántos días hábiles pasó este ítem en columnas de "trabajo en curso", en oposición con columnas de cola / buffer y estado bloqueado o sin trabajo del equipo sobre el mismo.

$$\text{Touch Time} \leq \text{Cycle Time} \leq \text{Lead Time}$$

Eficiencia del Ciclo del Proceso:

En realidad, cuanto tiempo estamos trabajando sobre el tiempo total.

$$\% \text{ Eficiencia ciclo proceso} = \text{Touch Time} / \text{Elapsed Time}$$

Kanban condensado:

