UNIDAD 1 – INTRODUCCIÓN A LA ING. DE SW

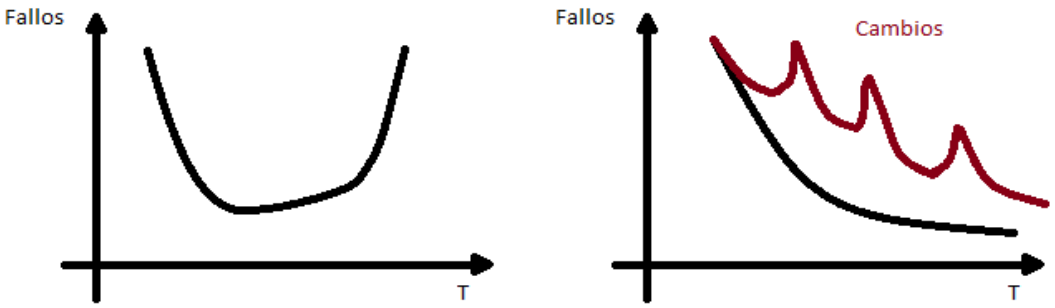
SWEBOK, **Software Engineering Body of Knowledge**, es un documento creado por la Software Engineering Coordinating Committee, promovido por la IEEE, que se define como una guía al conocimiento presente en el área de la Ingeniería del Software.

Diferencia entre Ingeniería y Ciencia

- El ingeniero trabaja con restricciones que un científico no tiene en cuenta, como por ejemplo 'look & feel', costo de comercialización, tiempos, etc.
- El ingeniero debe amoldarse a las restricciones sin perder la calidad ni la profesionalidad.
- Un científico construye para aprender. En cambio, un ingeniero aprende para construir y poder resolver problemas en el mundo real.
- Un ingeniero es alguien que hace por 1 centavo lo que cualquiera hace por un peso.
- El ingeniero se asocia con lo práctico, mientras que el científico se relaciona con la teoría.

Características del Software que lo diferencias de otros productos (según Brooks)

- Engañosamente fácil de cambiar (causas y efectos). Significa que es fácilmente modificable porque se puede ir actualizando, sin embargo, no siempre es fácil determinar dónde está el error, ni que implicancias puede tener una modificación. A veces, el punto donde se rompe está muy lejos de donde se produjo el error.
- Es un producto mental, no físico: no se rige por las leyes de la física. Es intangible, por eso el costo depende del desarrollo y no de su clonación.
- Se desarrolla, no se fabrica. No hay un proceso de fabricación.
- No envejece. No se deteriora.



Curva de envejecimiento de un producto físico vs. Curva de un producto software.

Problemas habituales en el desarrollo y mantenimiento de software:

- Los proyectos terminan con **mucho atraso** o se **cancelan**
- **Exceden el costo** estimado.
- El producto final no cumple las **expectativas del cliente**
- El producto final no cumple los requerimientos de **calidad mínimos**.

Para explicar cuáles son las causas habituales de estos problemas, complementamos con el paper de Classical Mistakes:

Lectura - Classical Mistakes

Es un listado de errores que ocurrieron tantas veces que los tengo que tener en cuenta para mi proyecto. Son 36 originales más 6 nuevos (en total son 42). “Los más importantes son los primeros diez” (palabras de Schivo). Se subclasifican en 4 categorías: Personas, Proceso, Producto y Tecnología:

Personas

1. Poca motivación
2. Personal débil (no funcionan como equipo)
3. Empleados sin supervisión
4. Heroicos (toman mucho riesgo)
5. Añadir gente a un proyecto atrasado
6. Oficinas ruidosas
7. Fricción entre desarrolladores y clientes (mala comunicación)
8. Expectativas irrealistas
9. Falta de participación de los involucrados
10. Optimismo

Proceso

1. Falta de gestión de Riesgos
2. Falta de planeamiento
3. Diseño inadecuado

Producto

1. Requerimientos que cambian siempre
2. Developer Gold-Plating: Agregar features que no son necesarios alargan la agenda.

Tecnología

1. “La bala de plata”: No hay soluciones absolutas!
2. Cambiar herramientas a mitad de proyecto

… Estos errores se resuelven con Gestión de Proyecto, Seguimiento, Gestión de Configuración.

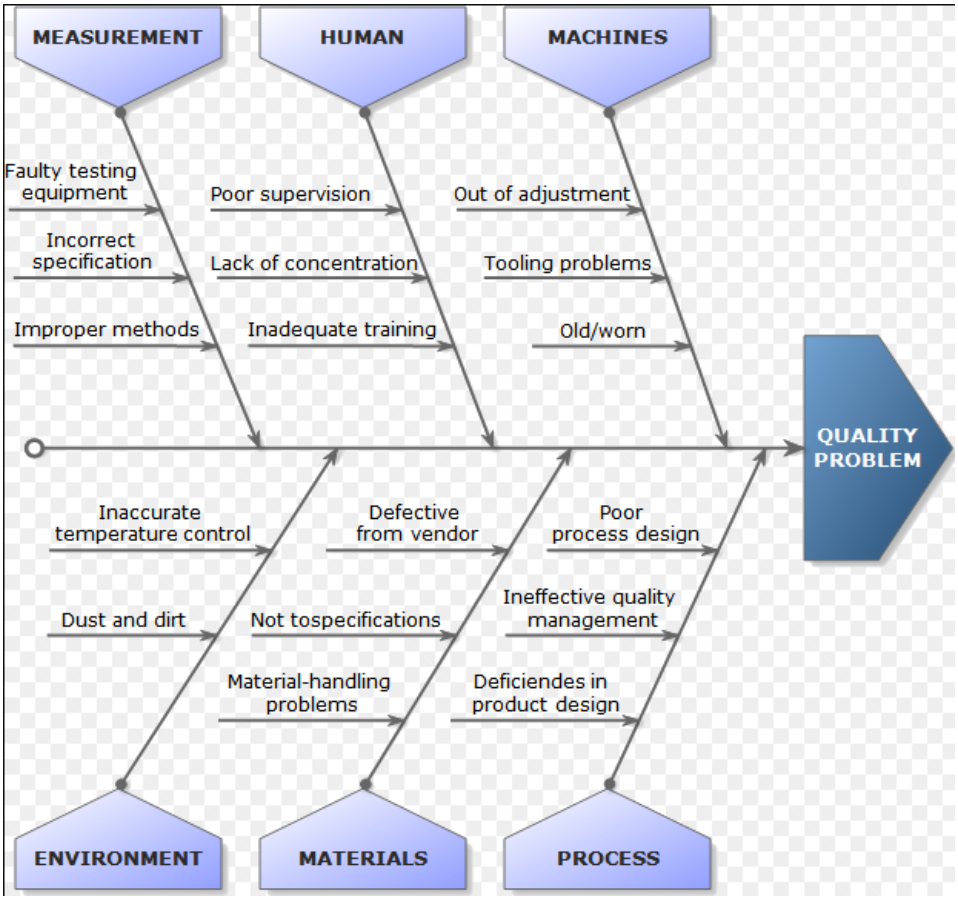
¿Qué podemos hacer para hacer frente a los problemas?

- Primero, reconocer que tenemos un problema. E inmediatamente después buscar un sponsor (persona que banca el proyecto, es decir, que interactúa con los interesados y es el motor de empuje para que se lleve adelante) y armar un equipo interdisciplinario que comparta la problemática y participe con su visión de identificación de causas.

- También podemos utilizar una técnica que es conocida como el diagrama de causa-efecto:

Diagrama de Causa-Efecto (Espina de Pescado / Ishikawa)

Permite identificar en detalle todas las posibles causas relacionadas a un problema. Concentra la atención en las **causas** y no en los síntomas. Gráfico ejemplo:

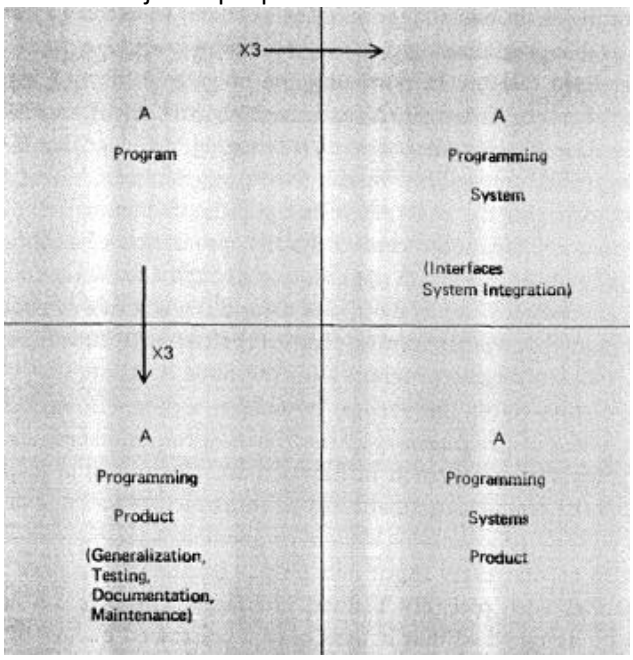


Lectura - The Mythical Man-Month

1. The Tar Pit

The Programming Systems Product

Es importante tomar conciencia de ***que*** es lo que se desea producir. En el siguiente esquema se visualiza distintos objetos que pueden ser creados:



Un **programa (Program)** es el objeto el que programador utiliza para estimar la productividad.

Un **producto de programación (Programming Product)**, es un programa que puede ser ejecutado, testeado, reparado y extendido. Para esto se requiere de un algoritmo generalizado, así como también de que el programa sea fuertemente testeado y documentado. Utilizando "la regla del pulgar" se estima que cuesta al menos tres veces lo que cuesta un programa debugueado con la misma funcionalidad.

Un **sistema programado (Programming system)** es una colección de programas que interactúan entre sí, coordinados en funcionamiento y con formato específico, tal que, este conjunto forma una entidad para realizar grandes tareas. Para que un programa se convierta en un sistema programado debe ser escrito de tal manera que todos los input y outputs de las interfaces queden bien definidos. Asimismo, no solo debe testearse el programa como unidad, sino que debe probarse con otros componentes del sistema en todas sus combinaciones posibles. Un sistema programado cuesta 3 veces más que un programa stand-alone con la misma funcionalidad.

Finalmente tenemos un **producto sistema programado (programming system product)**, que es la combinación de las dos anteriores y cuesta nueve veces más que el programa en sí. Este es el objeto más útil y el cual se desea obtener luego de realizar un esfuerzo que implique desarrollar un sistema.

The Joys of the Craft

En este apartado se menciona los aspectos "positivos" que tiene la programación para el hombre. En síntesis: es divertido crear cosas nuevas, que son útiles para otros, que son pequeñas piezas de un rompecabezas, que nos permite aprender y mejorar constantemente nuestra técnica y que es un proceso creativo.

The Woes of the creaft

En este apartado se mencionan los aspectos "negativos" (dolores de cabeza) que nos da la programación. En síntesis: Ajustar el requerimiento para que funcione perfecto (el humano es imperfecto, pero los programas deben serlo) es la tarea más difícil de programar; depender de programas creados por otras personas (que pueden estar mal diseñados, implementados, documentados, etc.); buscar y encontrar errores (bugs); darse cuenta que el programa será obsoleto cuando esté terminado (o incluso antes).

2. The Mythical Man-Month

The Mythical Man-Month

La mayoría de los proyectos que involucran software sufrieron más desvíos por falta de tiempo que por todas las demás causas combinadas. ¿Pero porque esta causa de desastres es tan común?

En primer lugar, porque las técnicas que utilizamos para estimar están desarrolladas pobremente (a dedo). Segundo, estas medidas, a su vez, suelen confundir esfuerzo con progreso. Tercero, el cronograma del progreso no es monitoreado correctamente. Cuarto, cuando se reconoce un desvío en el cronograma, la respuesta natura es agregar más gente.

Optimismo

La primera suposición falsa que impacta en el cronograma establecido para el desarrollo de sistemas es que "*todo irá bien".*

The'Man-Month

La segunda falacia se expresa en el modo de estimar el esfuerzo y armar el cronograma: el famoso mes-hombre. El costo varía en relación al producto entre el número de

hombres y el número de meses. El progreso no. Por lo tanto, el mes-hombre como unidad para medir la longitud de un trabajo es un mito peligroso y engañoso. Implica que los hombres y los meses son intercambiables.

En el único escenario en el que meses y hombres son intercambiables es cuando se necesitan realizar tareas que pueden ser repartidas entre los distintos trabajadores y no se necesita que haya ningún tipo de comunicación entre ellos. Es decir, para levantar algodón, este argumento es verdadero; sin embargo, no se aproxima en nada a la verdad a la hora de desarrollar sistemas.

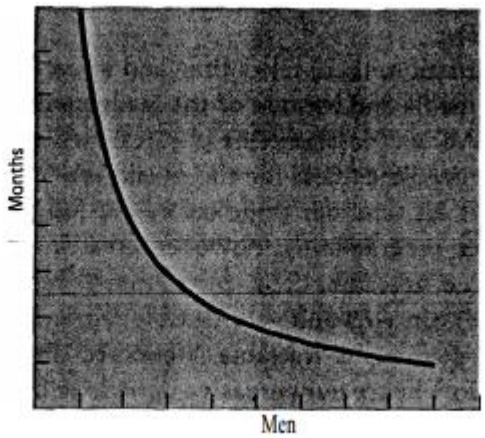


Fig. 2.1 Time versus number of workers—perfectly partitionable task

Cuando una tarea no puede ser particionada por sus respectivas restricciones, el intentar realizar un esfuerzo mayor no tiene impacto en el cronograma. Parir a un niño, conlleva un total de nueve meses, sin importar cuantas mujeres se asignen a dicha tarea. Muchas tareas relacionadas con el software tienen esta característica debido a su debuggeo secuencial.

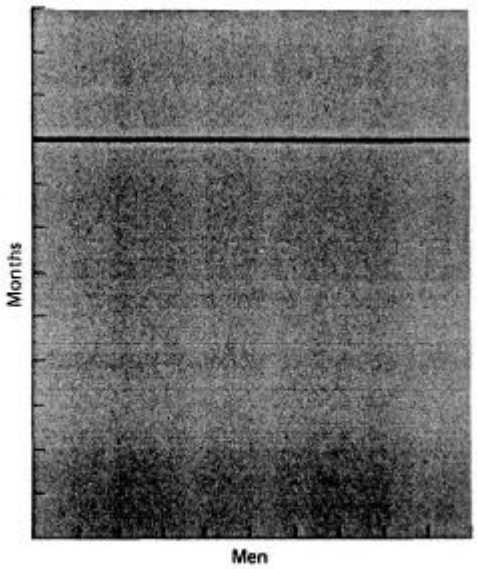


Fig. 2.2 Time versus number of workers—unpartitionable task

En las tareas que pueden ser particionadas pero que requieren comunicación entre las diferentes subtareas, dicho esfuerzo involucrado en dicha comunicación debe ser agregado al tiempo de trabajo.

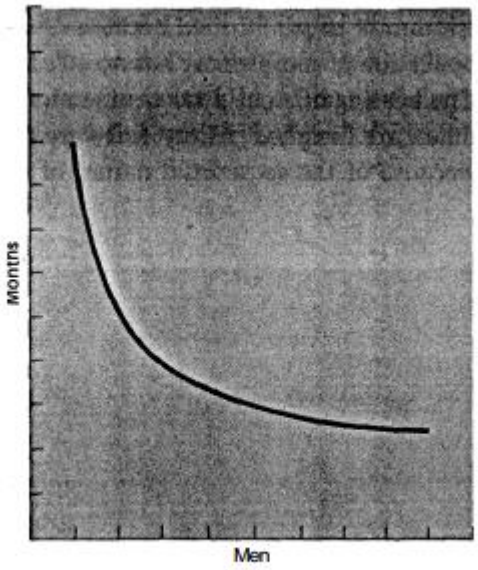


Fig. 2.3 Time versus number of workers—partitionable task requiring communication

Este agregado compuesto por el tiempo de comunicación, se conforma por dos partes, entrenamiento e intercomunicación. Cada empleado debe ser entrenado en la tecnología, objetivos, estrategia y plan de trabajo. Este entrenamiento no puede ser particionado, asi que esta parte del esfuerzo agregado varía linealmente según el número de empleados.

La intercomunicación es peor. Si cada parte de una tarea debe ser coordinada en forma separada con cada una de las otras partes, el esfuerzo incremente a razón de *n(n-1)/2*. Es decir, tres trabajadores requieren tres veces más intercomunicación que dos; cuatro requieren 6 veces más que dos. En definitiva, el esfuerzo que implica este tipo de comunicación puede directamente ser contraproducente al esfuerzo original de dividir la tarea original y presentar la situación que se muestra en la siguiente imagen:

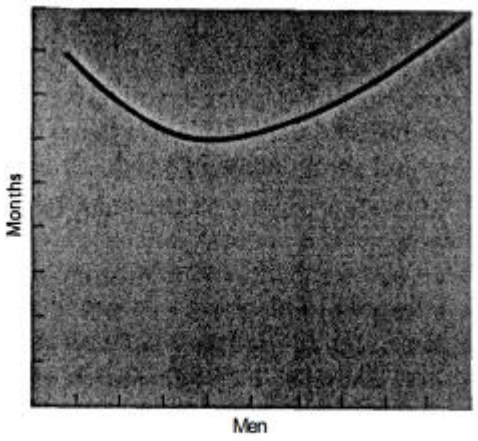


Fig. 2.4 Time versus number of workers—task with complex interrelationships

En síntesis, agregar más hombres, termina extendiendo el cronograma y no reduciéndolo.

Systems Test

Ocupa una gran porción en el cronograma de trabajo, debido a las restricciones que tienen los componentes para ser debugueados y testados. Asimismo, el tiempo requerido depende del número de errores encontrados.

Durante algunos años estuve utilizando con éxito la siguiente regla a dedo para realizar cronogramas que impliquen tareas de software:

- 1/3 Planificación
- 1/6 Codificación
- 1/4 Testeo de algunos componentes y testeo temprano de sistema
- 1/4 Testeo completo del sistema, y de todos los componentes.

1. El tiempo de planificación si bien es más extenso que el común permite producir una especificación lo suficientemente sólida en detalle, pero no tanto como para no dejar lugar a la exploración de nuevas técnicas.
2. La mitad del cronograma está destinado al debugging del código completo, lo cual es mucho más tiempo del normal
3. La parte sencilla de estimar es la codificación, lo restante.

No permitirse esta cantidad de tiempo para testear un sistema, es un error. Como los atrasos ocurren llegando al último tramo del cronograma, nadie se percata de que hay desvíos en el mismo hasta que es casi la fecha de entrega.

Además, un atraso en este punto puede traer repercusiones financieras y psicológicas.

Regenerative Schedule Disaster

Agregar más gente a un proyecto atrasado lo atrasa aún más.

Una opción es re planificar el cronograma, teniendo cuidado y dejando tiempo suficiente.

Otra opción es recortar formalmente la tarea (es decir, acortar su alcance) y re planificar, en vez de quedarse mirando como la tarea se recorta sola por un pobre diseño y un testeo incompleto.

En síntesis, el número de meses que demora un proyecto depende de sus limitaciones secuenciales. El máximo número de hombres depende del número de tareas independientes. De estas dos cantidades uno puede diseñar un cronograma utilizando menos hombres y más meses (El único riesgo aquí es que el sistema quede obsoleto). Uno sin embargo no puede, rediseñar un cronograma utilizando más hombres y menos meses.

UNIDAD 2 – CALIDAD (Risk Management)

- Los modelos de calidad permiten convertir la calidad del SW en algo tangible y medible.
- ¿Varía con el tiempo el concepto de Calidad? => Sí.
- ¿La calidad es un concepto universal? => No, es subjetiva, ¿depende de la percepción personal.

Visiones de la Calidad

1) **Visión Filosofal/Trascendental**: Se basa en la percepción de la gente (lo que la sociedad dice). “Veo la estrella de Mercedes Benz y reconozco que tiene calidad”. Esta visión está ampliamente relacionada con el marketing. El SW no se sitúa bien con esta visión, ya que la visión trascendental se adecua mejor con productos que apelan a los

sentidos. Es la menos objetiva de todas las visiones.

2) **Visión de Usuario**: La calidad es la satisfacción de lo que el usuario quiere o necesita ("¿Para qué la voy a usar"?). Es muy subjetiva ya que cada usuario tiene gustos y necesidades distintas.

3) **Visión de la Manufactura (Conformance to requirements)**:
   Esta visión compara el proceso. Si el producto cumple con todos los requerimientos => Hay "calidad" (no importan las necesidades del usuario ya que se asume que estas están expresadas en los requerimientos). Busco optimizar mi proceso, por ejemplo, minimizar el desperdicio que se produce en las fábricas. En SW intento que cada paso sea consistente con los demás.

   Mayor calidad => Menor costo … "Quality is Free". Toda la calidad que yo inyecto se retribuye al final.

   Ej.: ¿Un BMW tiene mayor calidad de manufactura que un Fiat Uno? => No hay respuesta. Si ambos productos siguen todos sus requerimientos de fabricación, ambos poseen "calidad", pero no es comparable.

4) **Visión del Producto**: Compara productos. Pienso en los features del producto y sus atributos. La calidad es en función de la cantidad de estos últimos.

   Más características => más calidad => más costo

   Ej.: Calidad de una Cámara = Resolución + Sensibilidad a la luz + Capacidad Almacenamiento.

5) **Visión basada en el Valor/costo**: Se un busca un precio acorde al juego de la oferta y la demanda. "Algo tiene buena calidad si alguien me compra algo a cierto precio". Mientras mayor es el costo, su calidad percibida baja (si no representa un buen valor para el cliente, su calidad baja).

Las visiones de calidad existen porque la calidad es subjetiva.

La calidad no es algo fijo. Es algo que se negocia. De hecho, en cada proyecto se negocian y determinan distintos patrones de calidad. Es parte del relevamiento determinar cuáles son las directivas de calidad. De hecho, es posible combinar distintos modelos de calidad para obtener la calidad deseada (Por ejemplo: usar la visión del usuario para tomar los requerimientos, la del producto para hallar los atributos de producto que satisfacen esos requerimientos, y la de la manufactura para construir el producto).

Costo de calidad

Todos apuntamos a conseguir una mayor calidad, pero la realidad es que el gasto en iniciativas para mejorar la calidad es económicamente viable solo si el ahorro que produce tener ese plus de calidad es mayor que la inversión inicial que se hizo para alcanzar esa calidad. El costo de calidad es un modelo métrico y económico para entender el costo-beneficio de la calidad vs no calidad. Puede utilizarse al comienzo de un proyecto para

justificar una inversión en la mejora de la calidad o al concluir un proyecto para medir los resultados de dichos gastos.

La calidad es una de las 4 variables que un tiene proyecto y que debe ser balanceada junto con otras 3 que son: costo, cronograma y features (funcionalidad, alcance). La siguiente formula expresa la relación entre estas 4 variables:

$$\text{Cost} * \text{Schedule} = \text{Features} * \text{Quality}$$

La fórmula anterior sugiere que mejorar la calidad requiere una combinación de mayor costo, un cronograma de proyecto más largo, o menos funcionalidad.

En la práctica la relación entre costo y calidad no es tan sencilla. En algunos casos es posible incrementar la calidad sin aumentar los costos. Esto se refleja en la siguiente fórmula:

$$\begin{aligned}\text{Cost of Quality} = &\ (\text{cost of defect prevention and early reviews}) \\ &+ (\text{cost of late stage testing and rework})\end{aligned}$$

Es decir, incrementando la primera categoría (prevención), se tiende a reducir la segunda categoría (re-trabajo). Cuando la cantidad de dinero ahorrado evitando defectos en el sistema aumenta más rápidamente que la cantidad invertida para evitar estos defectos, se dice "quality es free".

A continuación se muestra un gráfico del modelo tradicional de costo de calidad:

Figure 18.a. Traditional Cost of Quality Model

Los costes de calidad se dividen en costes de prevención y de evaluación, mientras que los de no calidad se dividen en internos y externos.

- **Calidad Interna**: Es propio del software desarrollado. Comprende: Estructura del programa, reusabilidad, mantenibilidad, legibilidad, complejidad, flexibilidad, testeabilidad, etc.
- **Calidad Externa**: Es lo que se ve externo al software desarrollado. Comprende: Si funciona apropiadamente, costo de mantenimiento, eficiencia, fiabilidad.

ISO9126: Calidad de Producto

ISO 9126 es un estándar internacional para la evaluación de la calidad del software. Tiene atributos + sub-atributos (27 en total).

1) **Funcionalidad**: Es la capacidad del software de cumplir y proveer las funciones para satisfacer las necesidades explícitas e implícitas cuando es utilizado en condiciones específicas:
    a. Adecuación: Capacidad para proporcionar un conjunto apropiado de funciones que cumplan las tareas y objetivos especificados por el usuario.
    b. Exactitud: Capacidad para hacer procesos y entregar los resultados solicitados con precisión o de forma esperada.
    c. Interoperabilidad: Capacidad del SW para interactuar con otros sistemas.
    d. Seguridad: Cap. Del SW para proteger la información (ej.: denegar acceso a personas no autorizadas).
    e. Cumplimiento funcional: La capacidad del software de cumplir los estándares referentes a la funcionalidad.

2) **Confiabilidad**: Es la capacidad del software para asegurar un nivel de funcionamiento adecuado cuando es utilizando en condiciones específicas.
    a. Madurez: capacidad para evitar fallar.
    b. Tolerancia a fallos: Mantener un nivel de prestaciones en caso de fallos.
    c. Recuperabilidad: La capacidad que tiene el software para restablecer su funcionamiento adecuado y recuperar los datos afectados en el caso de una falla.
    d. Cumplimiento de la confiabilidad: La capacidad del software de cumplir a los estándares o normas relacionadas a la fiabilidad.
3) **Usabilidad**: Es la capacidad del software de ser entendido, aprendido, y usado en forma fácil y atractiva
    a. Entendimiento: capacidad para ser entendido (cómo puede ser usado)
    b. Aprendizaje: La forma como el software permite al usuario aprender su uso.
    c. Operabilidad: La manera como el software permite al usuario operarlo y controlarlo.
    d. Atractividad: La presentación del software debe ser atractiva al usuario. Esto se refiere a las cualidades del software para hacer más agradable al usuario, ejemplo, el diseño gráfico.
    e. Cumplimiento de la Usabilidad: La capacidad del software de cumplir los estándares o normas relacionadas a su usabilidad.

4) **Eficiencia**: La eficiencia del software es la forma del desempeño adecuado, de acuerdo a al número recursos utilizados según las condiciones planteadas. Se debe tener en cuenta otros aspectos como la configuración de hardware, el sistema operativo, entre otros.
    a. Comportamiento de tiempos: Los tiempos adecuados de respuesta y procesamiento, el rendimiento cuando realiza su función en condiciones específicas.
    b. Utilización de recursos: La capacidad del software para utilizar cantidades y tipos adecuados de recursos cuando este funciona bajo requerimientos o condiciones establecidas.
    c. Cumplimiento de la Eficiencia: La capacidad que tiene el software para cumplir con los estándares o convenciones relacionados a la eficiencia.

5) **Mantenibilidad**: La capacidad de mantenimiento es la cualidad que tiene el software para ser modificado. Incluyendo correcciones o mejoras del software, a cambios en el entorno, y especificaciones de requerimientos funcionales.
    a. Analizable: La forma como el software permite diagnósticos de deficiencias o causas de fallas, o la identificación de partes modificadas.
    b. Cambiable: La capacidad del software para que la implementación de una modificación se pueda realizar, incluye también codificación, diseño y documentación de cambios.
    c. Estabilidad: Capacidad para evitar efectos inesperados debido a modificaciones en el SW.
    d. Testeabilidad: La forma como el software permite realizar pruebas a las modificaciones sin poner el riesgo los datos.
    e. Cumplimiento de normas de mantenibilidad

6) **Portabilidad**: Facilidad de transferirlo de un ambiente a otro. Incluye:
    a. Adaptabilidad: capacidad para ser adaptado a diferentes entornos sin reacciones negativas respecto del cambio.
    b. Instalabilidad: La facilidad del software para ser instalado en un entorno específico o por el usuario final.
    c. Coexistencia: La capacidad que tiene el software para coexistir con otros, la forma de compartir recursos comunes con otro software o dispositivo
    d. Remplazabilidad: La capacidad que tiene el software para ser remplazado por otro software del mismo tipo, y para el mismo objetivo. Ejemplo, la remplazabilidad de una nueva versión es importante para el usuario, la propiedad de poder migrar los datos a otro software de diferente proveedor
    e. Cumplimiento de normas de portabilidad

Para poder implementar ISO-9126 el interesado debe definir métricas con el objetivo de cuantificar los atributos y sub-atributos (Hay una decisión gerencial que va a asignar estos pesos a mi producto). El peso que le pongo a cada atributo es subjetivo y depende del caso.

Ejemplo: Un juego puede tener…
30% funcionalidad + 30% usabilidad + 20% eficiencia + 15% fiabilidad + 5% portabilidad + 0% mantenibilidad = 100%

CMMI (Capability Maturity Model Integration): Calidad de Proceso

CMMI es un modelo que evalúa la calidad en el proceso de desarrollo de software. Nos dice QUÉ hacer, pero no CÓMO. Nos dice que está mal, pero no te dice cómo implementarlo, eso lo decide cada organización.

Fundamentos CMMI – Capacidades

Las siglas CMMI, ya nos cuentan un poco de que trata este término, es decir, nos está indicando que es un modelo que integra capacidades con madurez esto. Estas capacidades son

de las que se nutre el modelo, por ejemplo: técnicas/desarrollo, PM, Testing, Control de cambios, deployement, análisis de requerimientos, etc.

Fundamentos CMMI – Madurez

Es el grado en que un proceso está definido, documentado, administrado, medido y controlado. Su riesgo es conocido y existe infraestructura para soportarlo. Si un proceso tiene un alto grado de madurez para una institución, quiere decir que dicho proceso no es complejo de utilizar en esa institución. No solo dice que el proceso se hace de una determinada manera, sino que lo hace así. Entonces la madurez evalúa que tan institucionalizado esta ese proceso.

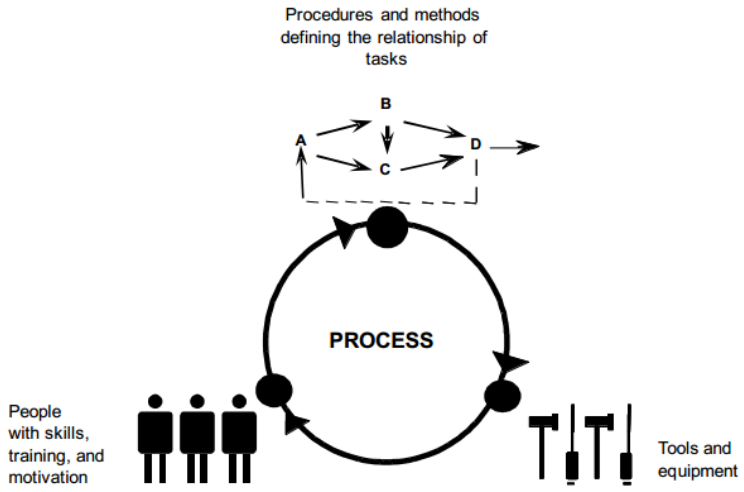
Proceso Maduro

- Soportado por la gerencia
- Definido, documentado, conocido, y practicado por todos
- Existe infraestructura adecuada para soportarlo
- Adecuadamente medido
- Adecuadamente controlado
- Presupuestos y plazos realistas
- Riesgo conocido y controlado
- Proactivo
- Es como respirar... institucionalizado

Proceso Inmaduro

- La gerencia dice soportarlo...
- Improvisado sobre la marcha
- Aunque esté definido no se sigue rigurosamente
- No hay entrenamiento formal ni herramientas para sustentarlo
- Presupuestos y plazos son generalmente excedidos por estimaciones no realistas
- Es una organización "reactiva"

Fundamentos del CMMI – Proceso



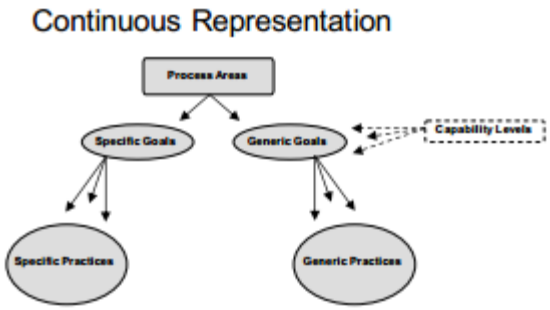
Representaciones de CMMI

Existen dos representaciones de CMMI. El contenido de ambas es el mismo pero su organización es diferente.

Hay que analizar cuál es la representación que más se adapta a los objetivos de la empresa.

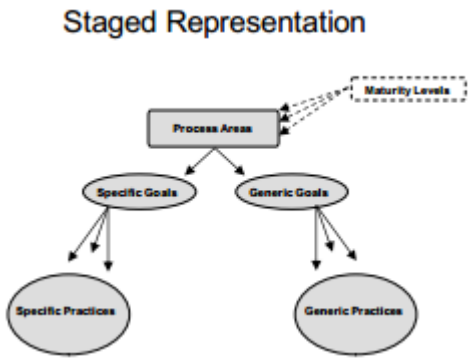
– Estructura o Continuas

Provee máxima flexibilidad ya que su objetivo es enfocarse en las "Process Area" específicas de acuerdo a los objetivos del negocio. Desaparece el concepto de nivel y me enfoco en las Process Areas que le incumben a mí organización, por ejemplo, todas las PAs relacionadas con PM o Soporte y mantenimiento, etc.



– Niveles

Marca un camino de Mejora. Estar en cierto nivel en particular me dice cómo trabaja esa empresa. Estructura conocida para aquellos que vayan desde SW-CMM.



NIVEL 1: Inicial / Caótico

No implementa CCMI. Los procesos son ad-hoc y caóticos readaptándose ante cada situación.

No hay un entorno estable que soporte procesos (no hay PAs) y el éxito de los proyectos está basado en el heroísmo de las personas.

Generalmente si estas organizaciones tienen éxito es porque excedieron presupuesto y plazo de entrega. También es común que estas organizaciones se sobren-comprometan, abandonen sus procesos ante las crisis, y no puedan repetir éxitos anteriores.

NIVEL 2: Administrado

Los procesos que siguen los proyectos se ejecutan de acuerdo a las políticas, y se monitorean, controlan, y revisan adecuadamente. La disciplina de los mismos hace que las prácticas se mantengan en tiempo de stress.

Los proyectos los llevan a cabo personas con habilidades y recursos adecuados para producir las salidas (entregables) de forma controlada. El estado de estos entregables es visible a la gerencia en puntos predefinidos

Se involucra a los participantes (stakeholders) relevantes.

Los trabajos y servicios satisfacen los estándares, procedimientos y procesos.

Process Areas:

- Admin. de Requerimientos (maneja el scope de los requerimientos)
- Planeamiento de Proyectos
- Monitoreo y Control de Proyecto
- Administración de Acuerdo con Proveedores
- Medición y Análisis
- Aseguramiento de Calidad
- Admin. de Configuración

NIVEL 3: Definido

Los procesos están bien descriptos y documentados. Este conjunto de procesos existe y se mejora en el tiempo. Asimismo, le otorga consistencia a toda la organización.

Los proyectos definen su proceso adaptando ese conjunto de procesos estándar, mientras que en un nivel 2 cada instancia específica de un proceso puede ser diferente, en nivel 3 tiene que ser consistente y las diferencias dadas por las reglas de adaptación.

Process Areas:

- Desarrollo de Requerimientos
- Gestión de Proyectos integrada
- Gestión de Riesgos
- Integración de Producto
- Validación (Cumple con las necesidades del usuario)
- Verificación (Satisface el requerimiento)
- Solución Técnica
- Enfoque del proceso organizacional
- Entrenamiento organizacional
- Resolución y Análisis de Decisiones
- Equipos Integrados
- Ambiente Organizacional para la Integración

La diferencia entre nivel 2 y nivel 3, es que en el último todos los procesos están bien definidos, documentados, no hay nada librado al azar.

A partir del nivel 3 existe el concepto de ***tailoring***. Esto quiere decir ajustar el proceso a mis necesidades. No siempre podemos cumplir con las PAs tal cual lo dice el manual, entonces lo moldeamos.

NIVEL 4: Cuantitativamente Administrado

Básicamente para entrar en este nivel, la organización tiene que realizar un gran esfuerzo en diseñar e implementar métricas que permiten medir sus procesos, y es en base a estas, que dicha organización podrá tomar decisiones.

Process Areas:

- Desempeño del proceso organizacional
- Gestión de Proyecto Cuantitativa.

NIVEL 5: Optimizado

En este nivel, la organización mejora continuamente sus procesos y tecnologías basada en un entendimiento cuantitativo de los objetivos del negocio y las necesidades de performance de la organización gracias a las métricas del nivel anterior. Aquí ya entra el

concepto de predicciones e hipótesis, con los cuales me adelanto a los problemas y a su solución.

Process Areas:

- Innovación y desarrollo organizacional.
- Resolución y análisis causal.

SCAMPI

Una evaluación SCAMPI le permite, a la empresa conocer en qué estado se encuentran sus procesos respecto de lo que requerido por el modelo de CMMI.

Esta evaluación solo sirve para implementar CMMI por niveles.

Existen 3 tipos de SCAMPI C, B, A en orden de jerarquía:

- **Scampi C:** Es una evaluación interna. Consisten en realizar entrevistas internas y recolectar evidencia. Puede hacer una o más personas.
- **Scampi B:** Es una evaluación más detallada que la anterior. Los evaluadores deben ser certificados y pertenecer a una entidad externa.
- **Scampi A:** Es la más rigurosa de todas. Otorga puntaje. Debe haber unanimidad para poder implementar CMMI.

Lectura – When good enough software is the best

Plantea un dilema para el desarrollo de software comparándolo con un cartel que dice su impresora: “Rápido, Barato, Bien hecho. Elige dos”. Los clientes sin embargo quieren los 3.

Entonces si los clientes quieren los 3 y no hay tiempo suficiente, hay que explorar el concepto “good enough software”, en las negociaciones, para poder tener éxito.

El autor expresa la idea de una de las cosas que sucede cuando el cliente pide software a medida y de calidad. Esa petición que realiza el cliente, obviamente tiene un costo un tiempo acorde al producto que esta espera. Sin embargo, en las negociaciones, nunca falta el simplista que dice “pero si eso lo hago con Word, Excel y Access”.

Plantea un ejemplo en el cual cambiar la tecnología parecía ser la mejor opción ya que su proceso actual tenía tareas molestas, no era de la mejor calidad, le faltaba automatización y era más caro. Sin embargo, la tecnología que parecía superar y corregir todas estas falencias por un amplio margen, requería cambiar de formato todos los archivos que procesaban y de sistema operativo. Se concluye entonces que el Project manager tomo una buena decisión que si hubiera hecho el “switch” probablemente el proyecto se atrasaba.

Expresa que el Project manager debe estar consciente de cada parámetro (costo, cronograma, personal, funcionalidad, calidad) es crucial. Sin embargo, es el cliente quien decide el balance apropiado de estos (y obviamente querrá todo al 100% o más). El Project manager es el encargado de bajarle a tierra al cliente de que con X cantidad de gente, en Y unidades de tiempo a Z dólares cada una, se puede entregar P unidades de funcionalidades con Q bugs por funcionalidad.

El autor plantea su preocupación por la incertidumbre o el desconocimiento que hay acerca de cómo interactúan estas variables X, Y, Z, P y Q. Tomando esto como premisa, en los

proyectos largos (más de un año) es inevitable la renegociación. Lo difícil es entonces justamente, sentarse frente al cliente para llevar a cabo esta renegociación. Se requiere mucho valor para decirle al cliente: “Puedo entregarte un software que hace lo que querés, pero sin que funcione”

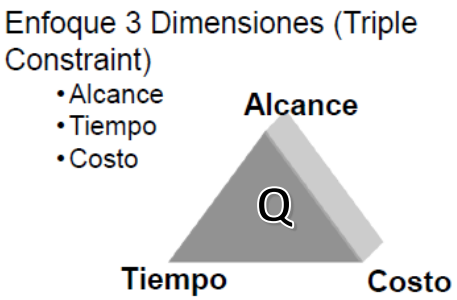
UNIDAD 3.0 – PREPARACIÓN DE UN PLAN DE PROYECTOS. RIESGOS

Conceptos Introductorios

Proyecto: Es un esfuerzo temporal emprendido para crear un producto o servicio único para lograr un objetivo.

Project Management: La administración de proyectos es una disciplina que consiste en planificar, organizar, obtener y controlar recursos, utilizando herramientas y técnicas para lograr que el proyecto logre sus objetivos en tiempo y forma.

Dimensiones de un proyecto: Al ser limitados en recursos, la relación entre los elementos que impulsan al proyecto puede verse de la siguiente forma:



No pueden cumplirse todos a la vez. Siempre voy a estar inclinándome para un lado. Dicha inclinación determina la calidad.

Características de las dimensiones:

- **Driver:** Es el atributo que me empuja a llevar adelante el proyecto.
- **Restricción**: Es como el *driver,* pero impuesto externamente.
- **Grado de libertad:** Son las variables con las que se puede jugar.

Plan de Proyectos

Presupuesto

Objetivos

Pueden ser de negocio, de proyecto y de producto. Un objetivo debe ser SMART (Specific Measurable Attainable Relevant Time-Related). El éxito del Proyecto deberá medirse en el grado de cumplimiento de dichos objetivos.

Alcance

Define los límites del sistema (lo que incluye, de ser necesario se aclara lo que “no” incluye).

Riesgos

Un riesgo es la posibilidad de que suceda un evento negativo que impacte en los objetivos del proyecto.

Se caracterizan por tener los siguientes atributos:

- Probabilidad de ocurrencia (1-100%)
- Impacto ($; 1-10; Bajo/Medio/Alto)
- Exposición = Probabilidad de ocurrencia * Impacto

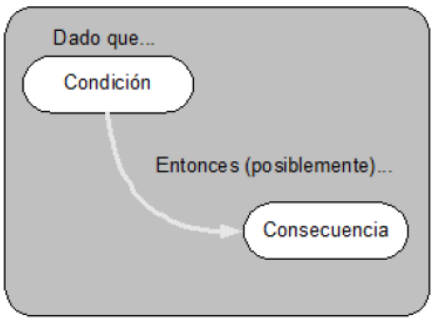
Dado que los riesgos son problemas que aún no llegaron, es necesario gestionar los mismos (Risk Managment).

La gestión del riesgo trata de identificar, resolver y comunicar los riesgos. Se basa en tomar decisiones bajo niveles de incertidumbre que tienen incidencia en el futuro. No es una actividad aislada, debe acompañar a todo el ciclo de vida de desarrollo de SW

Paradigma de Gestión del Riesgo (según SEI)

1. Identificación

Consiste en localizar los riesgos antes de que se conviertan en problemas y afecten al programa. Se utilizan varias técnicas como brainstorming, taxonomías, repaso de experiencia, etc. Deben documentarse. Para enunciarlos se utiliza la representación de Glutch:



2. Análisis

Consiste en transformar la información anteriormente capturada en información que permita tomar decisiones. Un buen análisis debe:

- Estimar probabilidad e impacto
- Estudiar causas y acciones correctivas
- Identificar causas comunes
- Identificar tiempos de ocurrencia

3. Planificación

Transformar la información acerca de los riesgos en decisiones y acciones. La prioridad se establece en función del grado de exposición y de la urgencia que demande la acción correctiva. Un plan de acción puede tener cualquiera de las siguientes formas:

- Evitar el riesgo.
- Reducir la probabilidad de ocurrencia con planes de mitigación.
- Atacar el impacto con planes de contingencia.
- Aceptar el riesgo sin tomar acciones, aceptando las consecuencias derivadas de su posible ocurrencia

Plan de Mitigación de Riesgos	**Plan de Contingencia de Riesgos**
Se toman acciones tempranas sobre el impacto y la probabilidad de ocurrencia de los riesgos, independiente de la ocurrencia de los mismos	Se planifican ciertas acciones, pero se monitorea señales de alerta (triggers). Se toman las acciones solo cuando se disparan dichas señales
Se gasta tiempo y dinero por adelantado debido a una condición de riesgo identificada. Proactivo	No se gasta tiempo y dinero por adelantado, pero se puede hacer una reserva para gastarla cuando sea necesaria

4. Seguimiento

Consiste en monitorear el “status” del resigo, y que las acciones que fueron definidas en el plan se ejecuten, informando tanto las posibles desviaciones respecto de los objetivos, como el impacto en el presupuesto, calendario y consideraciones técnicas. Para esto se definen métricas, y eventos que se disparan para asegurarse que las acciones estipuladas en la planificación funcionan correctamente

5. Control

Realizar las correcciones en los desvíos ocurridos en la planificación.

6. Comunicación

Es el centro del paradigma porque sin comunicación efectiva, ningún aprproach para gestionar el riesgo es viable. La comunicación es crítica porque es la que permite la interacción entre los elementos del paradigma. Otorga feedback acerca de las actividades que podrían causar algún riesgo, de los riesgos actuales, de los emergentes.

Composición de un Proyecto

Armado del plan

Para armar el plan de proyecto debemos descomponer el proyecto que estamos planificando en:

- Fases / Etapas
- Tareas / Subtareas / Actividades.

Una vez que tenemos esa división realizada, es necesario determinar las relaciones de precedencia en dicha descomposición y asignar los recursos del proyecto a cada tarea / subtarea / actividad.

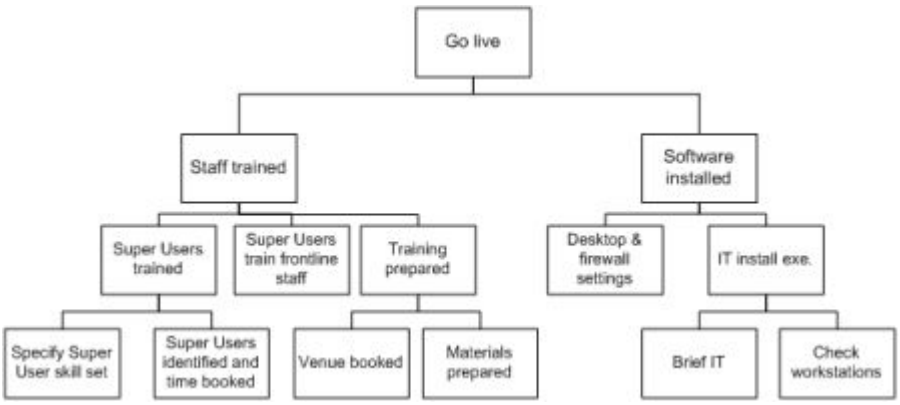
WBS (Work breakdown structure)

Es un método para representar en forma jerárquica los componentes de un proceso (en nuestro caso el plan de proyecto) o un producto. Este método no determina dependencias, solo jerarquía, y **contiene** todas las tareas o entregables para cumplir con el alcance (que se determinó a partir de los objetivos). La WBS permite dividir el problema en partes más chicas. No importa el orden de las tareas porque estas no tienen cronograma

Tipos de WBS

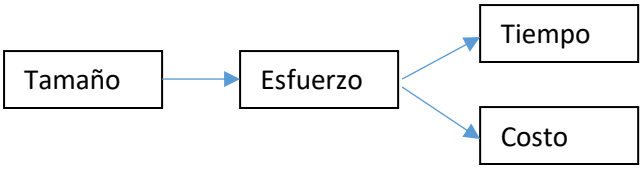
- ***A nivel conceptual***
  - *WBS por Proceso (Análisis/ Desarrollo/ Construcción/ Testing/ Implementación).*
  - *WBS por Producto (por funciones).*
  - *WBS Híbridas (ciclo de vida en el alto nivel con detalle de características de productos en el bajo nivel.*
- ***Por formato***
  - *Diagrama Jerárquico.*
  - *Tabla Indentada.*

El diagrama jerárquico (que posee estructura de árbol), es el que utilizaremos para explicar el método:



El primer paso para armar una WBS es definir el nodo raíz de la jerarquía (Por lo general es el nombre del proyecto). Acto seguido se divide en sub-partes hasta llegar al nivel de granularidad requerido. Las hojas del WBS son todas las tareas del proyecto que necesitan

realizarse para completarlo (suelen ser 5 ± 2 niveles), y sumando el peso de las mismas, puedo estimar el esfuerzo (por ejemplo, en horas/hombre).



Determinación de las dependencias

Ya que el WBS determina jerarquía, no dependencias, el siguiente paso es definir las dependencias de todas las tareas. No olvidarse las dependencias con terceras partes. Para esto hay que asegurar que cada tarea tenga un entregable preciso y a su vez que "deliverables" requieren las tareas sucesoras. Es importante incluir "milestones ("hitos, puntos de revisión) para saber el status del proyecto (donde estamos parados).

Una vez hecho todo esto, podemos identificar el camino crítico (secuencia de tareas cuyo atraso provoca atrasos en la fecha de fin del proyecto).

(DIAGRAMA DE GANTT)

Asignación de Recursos

En esta etapa debemos conformar el equipo de trabajo que se encargará de realizar las diferentes actividades planificadas. Para esto es necesario conocer todos los roles de los recursos con los que contamos (PM, Devs, QA, Tester, Analista, etc).

Por otra parte, no hay que olvidar la asignación de los roles cubiertos por el cliente (Usuarios, Sponsor, Stakeholder). Dentro de los usuarios tenemos a los directos/indirectos, claves, y campeones. El usuario campeón suele ser un recurso con alta demanda en su "día a día" y puede comprometer su participación en el proyecto.

Se recomienda evitar caer en la "dedicación completa" ya que por diversos motivos un recurso puede no estar disponible el 100% del proyecto. De todas formas, los recursos tienen que tener un backup asignado.

Lectura - Software Development Risk: Opportunity, Not Problem

Background

El autor agrega que los riesgos son parte de cualquier actividad y nunca pueden ser eliminados, ni que tampoco se pueden conocer todos los riesgos.

El riesgo en si no es algo malo; es esencial para el progreso y fallar suele ser clave en el aprendizaje. Sin embargo, hay que aprender a balancear los posibles efectos negativos de las consecuencias de un riesgo contra sus potenciales beneficios de su oportunidad asociada.

Se explica el concepto de riesgo técnico, es decir, la posibilidad de que la aplicación de teorías, principios, y técnicas de la ingeniería de software no consiga entregar el producto de software adecuado. El riesgo técnico está compuesto por factores tecnológicos subyacentes

que pueden hacer que el producto final sea demasiado caro, se entregue tarde, o sea inaceptable para el cliente.

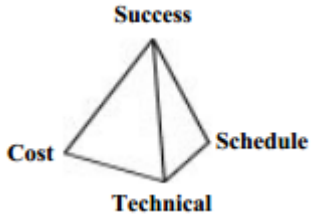
Hoy en día los riesgos técnicos son el corazón de muchos de los problemas que están causando las fallas del software. La esencia del "riesgo técnico" es fallar en construir el producto adecuado.

¿Cómo hacemos para controlar los riesgos técnicos? Gestión del riesgo:

1. **Identificar: I**dentificar los riesgos mientras haya tiempo de tomar acción.
2. **Comunicar:** Aceptar que los riesgos existen y comunicarlos a los responsables de resolverlos.
3. **Resolver:** Tomar decisiones sobre cómo actuar frente a los riegos, es decir, transformar un riesgo en una oportunidad para mejorar nuestras posibilidades de éxito.

La mayoría de los administradores del equipo de desarrollo de software entienden que deben gestionar riesgos. Sin embargo, por más que tengan la voluntad, muy pocas veces se les otorga la información que necesitan para realizar dicha actividad.

Para poder realizar una gestión del riesgo exitosa, el administrador debe tener en cuenta **todas** las variables, es decir, *costo, cronograma y **riesgos técnicos***.



El autor afirma que una de las principales dificultades para llevar esto adelante es la **comunicación**, es decir, hay un hecho cultural que inhibe a la gente a levantar la mano y decir "creo que esto podría sr un riesgo (o un problema).

El "SEI Risk Program" desarrollo una estrategia para ayudar a que el desarrollo de programas de software tenga éxito.

Risk Program Strategy

Si tuviéramos que sintetizar la estrategia, diríamos: "Hasta que no utilicemos una forma disciplinada y sistemática para identificar y enfrentar el riesgo técnico, nunca podremos controlar la calidad, el costo o el cronograma de nuestros productos de software".

Dicha estrategia se resume en los siguientes pasos:

1. Definir la gestión del riesgo de software mediante investigaciones y entrevistas.
2. Desarrollar un paradigma eficaz de gestión de riesgos de software mediante pruebas y prototipos con programas individuales.
3. Transferir la gestión del riesgo de software de los programas exitosos a las organizaciones y a la comunidad.

Los elementos del paradigma ya fueron enumerados y explicados anteriormente.

UNIDAD 3.1 – PREPARACIÓN DE UN PLAN DE PROYECTOS. ESTIMACIONES

Conceptos introductorios

¿Por qué fallan las estimaciones?

- Optimismo
- No hay registros de casos similares
- Mala definición de alcance
- Falta de experiencia
- No se estima el tamaño

Variables a estimar

1. Tamaño (podría ser LOC [“lines of code”], no hay un estándar para tamaño)
2. Esfuerzo
3. Costo
4. Tiempo

Tener presente

1) La estimación del desarrollo de SW vs la estimación de todo el proyecto. EL primero es un componente del segundo.

2) “Nivel de incertidumbre” que siempre está presente.

“Tips” para estimar:

- Asociar a las estimaciones un % de confiabilidad.
- Es recomendable presentar las estimaciones como rangos en lugar de un único valor
- Siempre presentar junto con la estimación, los supuestos que se tuvieron en cuenta para llegar a la misma.
- Tener presente la Ley de Parkinson: “Toda tarea se expande hasta ocupar el tiempo que tiene asignado”
- Considerar todas las actividades relacionadas al desarrollo de SW, no solamente codificación y testing (Análisis, diseño, actividades de SCM, etc.)
- No asumir que solo por el paso del tiempo y de las fases de un proyecto se avanza con menor incertidumbre en las estimaciones (Cono de la incertidumbre)
- Recolectar datos históricos para tener como referencia

El proceso de estimación

Es cíclico, y este compuesto por:

1. **Estimar:** Es decir, comenzar a estimar el tamaño para derivar el esfuerzo y el costo (la estimación se hace antes de que comience el proyecto).
2. **Medir:** Mientras evoluciona el proyecto, medir el tamaño, el esfuerzo y costo incurrido. Una vez terminado el proyecto, tengo los resultados de todas las mediciones.
3. **Registrar:** Dejar asentadas (registradas) las mediciones tomadas.
4. **Comparar:** Estimaciones vs. Mediciones

5. **Analizar:** Razones de desvíos, supuestos que variaron, temas no contemplados, es decir es la evaluación de por qué la estimación no coincide con la medición
6. **Calibrar:** Ajustar c/u de las variables y parámetros que intervienen en el proceso de estimación
7. **Volver a estimar:** 1) El mismo proyecto, pero ahora con más información que al comienzo del mismo y 2) nuevos proyectos con el proceso ajustado por la "calibración".

Métodos para estimar

Juicio experto (no paramétrico)

Depende de la persona que haga la estimación, por lo tanto, se basa en la experiencia personal. Por lo general se recurre a analogías.

Método PERT (no paramétrico)

Se basa en el juicio experto, incluye los factores optimistas y pesimistas en su estimación. Se pueda usar para estimar tamaño y esfuerzo.

$$Estimación = \frac{(Optimista + 4 * Medio + Pesimista)}{6}$$

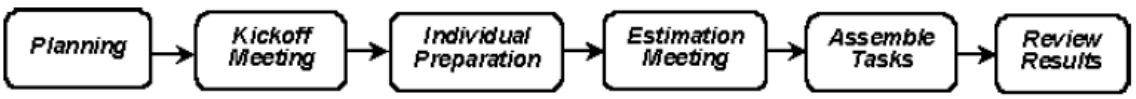
Wideband Delphi (no paramétrico) (complementado con paper "Stop Promising Miracles")

Consiste en pedirle a un pequeño equipo de expertos que, anónimamente, generen estimaciones individuales acerca de la descripción de un problema, para que luego logren alcanzar un consenso en la estimación, a través de iteraciones.

Entre sus ventajas se encuentran que es fácil de implementar, y otorga credibilidad. Se puede utilizar en etapas tempranas del proyecto, se recomienda utilizarlo en proyectos poco conocidos donde no se cuenta con historia

El punto de partida de una sesión Delphi podría ser directamente la especificación del problema a ser estimado o una lista inicial, a muy alto nivel, con las tareas a realizar, o el cronograma de trabajo del proyecto. Las salidas de aplicar este método pueden ser: una lista detallada con las actividades del proyecto; una lista con tareas en las cual se ve asociada su calidad, a que procesos pertenecen, etc.; estimaciones supuestas; un set de tareas con su respectivo estimado en el proyecto. Todas estas salidas nacen de los aportes de los participantes y permiten realizar un esfuerzo "bottom-up" para estimar el tamaño o cronograma del proyecto

A continuación, se muestra una imagen que ilustra el flujo de proceso de una sesión Wideband Delphi:



El problema a estimar, y quienes serán los participantes se define durante la etapa de Planeamiento (***Planning***). La reunión de arranque (***Kickoff Meeting***) se establece para que todos los estimadores se enfoquen en el problema. Cada participante prepara entonces una lista inicial con el análisis que realizó (puede ser cualquiera combinación de los outputs que

mencionábamos anteriormente) (***Individual Preparation***). Luego, llevan esos ítems a la reunión de estimación (**Estimation Meeting**), en donde se realizan varios ciclos de estimación, los cuales generan una lista de tareas y un set de estimados más concisos. Hecho esto, el Project manager consolida toda esta información (**Assamble Tasks**), y el equipo revisa los resultados de la estimación **(Review Results).** La sesión termina cuando todos los criterios fueron satisfechos. El resultado es una estimación más realista que cualquiera echa por un único estimador.

Function Points (paramétrico) (complementado con "Fundamentals of Function Point Analyisis")

Introducción

Es un método para desglosar el sistema en componentes más pequeños, con el objetivo de poder entenderlo y analizarlo mejor.

Los "Function Points" son una unidad de medida para el software, así como la hora lo es para el tiempo.

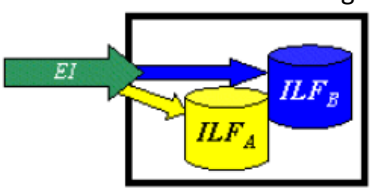
Para realizar un análisis con Puntos de Función, los sistemas se dividen en cinco clases. Las primeras tres clases o componentes son: External Inputs (Entradas externas), External Outputs (Salidas externas) y External Inquiries (Consultas externas). Cada uno de estos componentes operan con archivos, es por esto que son llamadas transacciones. Las otras dos clases son Internal Logical Files (Archivos lógicos internos) y External Interface Files (Archivos de interface externa). En estas es donde se almacenan datos que luego es combinada para generar información lógica.

Como los "Function Points" miden el sistema desde un punto de vista funcional, no dependen de la tecnología. Independientemente del lenguaje de programación, método de desarrollo, o plataforma de hardware utilizada, el número de function points en un sistema se mantiene constante. La única variable es la cantidad de esfuerzo que se necesita para entregar un conjunto de puntos de función. Gracias a que la cantidad de function points se mantiene constante, este método de análisis puede ser utilizado para determinar cuál es la herramienta, entorno o lenguaje más productivo en comparación con otros ya que cada function point se traduce en un esfuerzo según lo que se esté comparando.

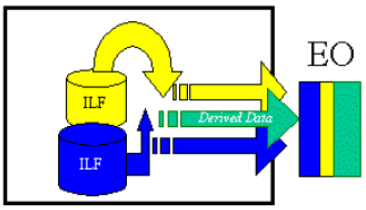
Los puntos de función pueden ser utilizados en todas las etapas del ciclo de vida de desarrollo de software.

Paso 1 – Identificar los elementos del sistema

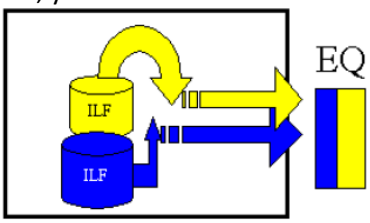
- **External Inputs (EI):** Es un proceso elemental en el cual la data cruza el limite desde el exterior al interior. Esta data puede provenir de una pantalla o de otra aplicación, y puede ser utilizada para actualizar uno o más archivos lógicos internos.



- **External Outputs (EO):** Es un proceso elemental en el cual la data cruza los límites desde el interior al exterior. Adicionalmente, una EO puede actualizar un ILF. Esta data es utilizada para crear reportes o archivos de salida que se envían a otra aplicación. Estos reportes o archivos se crean a partir de uno o más ILF's y EIF's.



- **External Inquiry (EQ):** Es un proceso elemental con componentes tanto de entrada como de salida que resulta de extraer data de uno o más ILF's y EIF's. El proceso de entrada no actualiza ningún ILF, y el de salida no contiene información derivada.



- **Internal Logical Files (ILF's):** Es un conjunto de datos relacionados lógicamente, que reside ***en*** las aplicaciones, que es mantenido a través de EI's.
- **External Interface Files (EIF's):** Es un conjunto de datos relacionados lógicamente que es utilizado solamente con el propósito de referenciar. Esta data reside ***afuera*** de la aplicación y es mantenida por otra aplicación. El EIF es un ILF utilizado para otra aplicación

Luego de identificar todos los componentes, esto se clasifican en **"High, Low o Average".** Para las transacciones, el ranking se basa en el número de archivos que actualizan o referencian (FTR's), y el número de tipos de elementos de datos (DET's). Para tanto los ILF's, como los EIF's, la clasificación se basa en los registros de tipos de elementos (RET's) y DET's.

Para los FTR's:

EI Table

FTR's	DATA ELEMENTS		
	1-4	5-15	> 15
0-1	Low	Low	Ave
2	Low	Ave	High
3 or more	Ave	High	High

Shared EO and EQ Table

FTR's	DATA ELEMENTS		
	1-5	6-19	> 19
0-1	Low	Low	Ave
2-3	Low	Ave	High
> 3	Ave	High	High

Values for transactions

Rating	VALUES		
	EO	EQ	EI
Low	4	3	3
Average	5	4	4
High	7	6	6

Para los RET's

RET's	DATA ELEMENTS		
	1-19	20 - 50	> 50
1	Low	Low	Ave
2-5	Low	Ave	High
> 5	Ave	High	High

Rating	Values	
	ILF	EIF
Low	7	5
Average	10	7
High	15	10

Paso 2 – Calcular FP sin ajustar

Unadjusted function points (UFP), se calculan de la siguiente manera:

Type of Component	Complexity of Components			
	Low	Average	High	Total
External Inputs	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	
External Outputs	___ x 4 = ___	___ x 5 = ___	___ x 7 = ___	
External Inquiries	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	
Internal Logical Files	___ x 7 = ___	___ x 10 = ___	___ x 15 = ___	
External Interface Files	___ x 5 = ___	___ x 7 = ___	___ x 10 = ___	
			Total Number of Unadjusted Function Points	___
			Multiplied Value Adjustment Factor	___
			Total Adjusted Function Points	___

Paso 3 – Factor de ajuste del valor (Value adjustment factor)

$$VAF = \frac{0.65 + (suma\ de\ factores)}{100}$$

Existen 14 factores de ajuste del valor. Cada uno se evalúa de acuerdo al grado de influencia (0-5):

1. *Data communications*
2. *Performance*
3. *Heavily used configuration*
4. *Transaction rate*
5. *Online data entry*
6. *End user efficiency*
7. *Online update*
8. *Complex processing*
9. *Reusability*
10. *Installation ease*
11. *Operations ease*
12. *Multiple sites*
13. *Facilitate change*
14. *Distributed functions*

Paso 4 – Calcular puntos de función (PFN)

Cálculo de puntos de función netos:

$$PFN = UFP * VAF$$

Conversión de FP a LOC: En base a los estándares del mercado, 1 FP equivale a:

	Mínimo	Media	Máximo
Java	40 LOC	55 LOC	80 LOC
C++	40 LOC	55 LOC	80 LOC
Cobol	65 LOC	107 LOC	150 LOC
SQL	7 LOC	13 LOC	15 LOC

Use Case Points (complementado con paper "Comparing Effort Estimates Base don Use Case Points with Expert Estimates"

Los modelos basados en casos de uso son utilizados en análisis orientados a objetos con el objetivo de describir los requerimientos funcionales de un sistema. Por lo tanto, sus atributos pueden servir como métricas del tamaño y complejidad de la funcionalidad de un sistema. Muchas organizaciones utilizan este modelo para el proceso de estimación.

Un modelo basado en casos de usos tiene dos partes, el diagrama de casos de usos y la descripción de los casos de uso. La primera ofrece un panorama de los actores y los casos de uso. La segunda detalla los requerimientos. Un actor representa un rol que el usuario puede interpretar relacionado con el sistema, y un caso de uso representa la interacción entre el actor y el sistema

El método del puntaje por casos de uso (*use case points method)* es aquel que sirve para estimar el esfuerzo en el desarrollo de software basado en caso de usos.

Este método está basado en el de function points, y apunta a otorgar un método simple de estimación adaptado a proyectos orientados a objetos.

El número de Use Case Points depende de la cantidad y complejidad de los casos de uso y de los actores intervinientes, así como también de factores técnicos y ambientales.

Este método establece que debería ser posible contar la cantidad de transacciones en cada caso de uso. Se denomina transacción al evento que ocurre entre un actor y un sistema. Los cuatro pasos en los que se divide el método basado en casos de usos son los siguientes:

Paso 1 – Clasificar Actores

- **Actores Simples:** Son sistemas externos, predecibles, y cuya interfaz de aplicación (API) está bien definida.
- **Actores Promedio:** Son sistemas que interactúan a través de un protocolo como puede ser el TCP/IP.
- **Actores Complejos:** Son personas que pueden estar interactuando con una GUI o una página web.

En este paso se cuenta la cantidad de actores en cada categoría y se multiplica ese número por el correspondiente peso. Así se obtiene el UAW (Unadjusted Actor Weight)

Tipo de Actor	Peso
Simple	1
Medio	2
Complejo	3

Paso 2 – Clasificar Casos de uso

Los casos de uso también se clasifican en simple (3 o menos transacciones), medio (4 a 7 transacciones) y complejo (más de 7 transacciones).

Se cuenta el número de casos de uso en cada categoría, multiplicando ese número por el correspondiente peso y se obtiene el UUCW (Unadjusted Use Case Weigth).

Tipo de Caso de uso	Peso
Simple	5
Medio	10
Complejo	15

Luego se obtiene el Unadjusted use case points:

$$\text{UUPC} = \text{UAW} + \text{UUCW}$$

Paso 3 – Factores de ajuste

Los casos de uso se ajustan basado en la cantidad de factores técnicos o del entorno que pueden influir en el esfuerzo necesario para desarrollar el software: A Cada factor se le asigna un valor entre 0 y 5 dependiendo de la influencia que tiene sobre él.

Factores técnicos

Factor	Descripción	Peso
T1	Sistema distribuido	2
T2	Respuesta Rendimiento Performance	2
T3	Eficiencia del usuario final	1
T4	Procesamiento interno complejo	1
T5	Código reusable	1
T6	Fácil de instalar	0.5
T7	Fácil de usar	0.5
T8	Portable	2
T9	Fácil de cambiar	1
T10	Concurrente	1
T11	Incluye características de seguridad	1
T12	Provee acceso a terceras partes	1
T13	Se requerien entrenamiento especial	1

Factores del entorno

Factor	Descripción	Peso
T1	Familiar con RUP	1,5
T2	Experiencia en la aplicación	0,5
T3	Experiencia en objetos	1
T4	Buena capacidad de análisis	0,5
T5	Motivación	1
T6	Requerimientos estables	2
T7	Trabajadores part - time	-1
T8	Lenguaje de programación	-1

*El **TechnicalComplexity Factor (TCF)*** es calculado en base a multiplicar cada factor de la tabla 1 por su peso y luego sumar todos estos valores para obtener el llamado ***TFactor***. Finalmente se aplica la siguiente fórmula:

$$\text{TCF} = 0.6 + (.01 * \text{TFactor}).$$

*El **Environmental Factor (EF)*** es calculado en base a multiplicar cada factor de la tabla 2 por su peso y luego sumar todos estos valores para obtener el llamado ***Efactor***. Finalmente se aplica la siguiente fórmula:

$$\text{EF} = 1.4 + (-0.03 * \text{EFactor})$$

El ***adjusted use case points (UCP)*** se calcula por la siguiente formula:

$$\text{UCP} = \text{UUCP} * \text{TCF} * \text{EF}$$

Paso 4 – Cálculo del proyecto

Karner propone un valor de 20 horas hombre por UCP para cada proyecto:

$$\text{UCP} * 20\ \text{HH} = \text{Costo en HH del proyecto}$$

Enfoque de Kirstein Rubi (2001): 15 a 30 hs x UCP

Object Points (complementado con lectura)

Son similares a los Function Points ya que proveen métricas para estimar que son aplicables durante todas las fases de desarrollo de software. Es importante destacar que la métrica de tamaño (size metric) puede ser aplicada en las etapas tempranas de desarrollo, como por ejemplo la toma de requerimientos para definir pantallas y reportes.

Una de las desventajas de los Function Points es que poseen múltiples implementaciones e interpretaciones de la métrica. Se han llevado a cabo estudios que muestran variancias de hasta un 30% entre distintas implementaciones.

Los Object Points evitan la subjetividad de los Function Points, ya implementan un factor de ajuste claro, sin ambigüedades.

Este método le asigna a cada componente un peso, de acuerdo a la clasificación por su complejidad y considera como factor de ajuste el porcentaje de reutilización de código.

Paso 1 – Identificar elementos del sistema

El método original solo contempla como elementos del sistema a las ***Pantallas, Reportes y 3 GL (módulos de software escritos en lenguajes de tercera generación como COBOL, C, C++, .NET, JAVA, etc).***

Paso 2 – Calcular Complejidad

Pantallas				**Reportes**			
N° de vistas contenidas	**# y fuentes de tablas de datos**			**Número de secciones contenidas**	**# y fuentes de tablas de datos**		
	Total <4 (<2 srvr < 3 clnt)	Total <8 (<2/3 srvr 3-5 clnt)	Total 8+ (>3 srvr >5 clnt)		Total <4 (<2 srvr < 3 clnt)	Total <8 (<2/3 srvr 3-5 clnt)	Total 8+ (>3 srvr >5 clnt)
<3	simple	simple	medio	0 o 1	simple	simple	medio
7-Mar	simple	medio	complejo	2 o 3	simple	medio	complejo
>8	medio	complejo	complejo	4+	medio	complejo	complejo

- ***srvr*** es el número de tablas de base de datos del tipo servidor (mainframe o equivalente) que se utilizan en la *pantalla o reportes.*
- ***clnt*** es el número de tablas de base de datos del tipo cliente que se utilizan en la *pantalla o reportes.*

Luego se asigna un peso de acuerdo a la siguiente tabla. Los pesos reflejan el esfuerzo relativo requerido para implementar una instancia de ese nivel de complejidad:

Object Type	Peso - Complejidad		
	Simple	Medio	Complejo
Screen	1	2	3
Report	2	5	8
3GL Component			10

Paso 3 – Calcular Objects Points

Ahora se suma el peso de todas las instancias de los objetos identificados, según la tabla anterior.

Finalmente, se pueden calcular los siguientes estimadores:

$$Esfuerzo = NOP/PROD$$

$$NOP(new\ object\ points) = OP * (100 - \%reusabilidad)/100$$

El valor de NOP depende del porcentaje de reusabilidad de código adquirido, y el valor de PROD (productividad) puede decidirse a través de la siguiente tabla:

	Muy Baja	Baja	Media	Alta	Muy Alta
PROD	4	7	13	25	50

Lectura Timebox Development

Introducción

Es una práctica que se lleva a cabo en el lapso de tiempo en el que se está desarrollando software, y cuyo objetivo es transmitirle al equipo de trabajo un sentido de urgencia y ayudarlo a mantener el enfoque del proyecto en sus características más importantes.

Esta práctica permite ahorrar tiempo gracias a que es posible redefinir el producto para que encaje en el cronograma en vez de redefinir el cronograma para que encaje en el proyecto.

El éxito de esta práctica radica en utilizarlo en los tipos de proyecto adecuados. Entre los requisitos para poder implementar este método podemos mencionar:

- Contar con un equipo que posea cierto seniority, en el cual los miembros del mismo ya hayan trabajado juntos (permite ahorrar tiempo de comunicación).
- Usuarios finales involucrados (ya que es inaceptable que el desarrollo de software se detenga por la no participación del usuario).
- Alcance definido y priorizado.
- Tecnología probada (no hay tiempo para probar nuevas tecnologías).
- La estimación debe ser realizada por el equipo de trabajo (esta es una estimación más confiable, no impuesta por un tercero, que nos permite “asegurar” que podemos cumplir el trabajo en el tiempo estimado).

- Equipo motivado.
- Contar con la voluntad de la administración de proyecto y usuarios finales de recortar características (alcance) en lugar de extender el programa.

Entre sus mayores riesgos, se encuentran el de utilizar esta práctica en productos no aptos y en sacrificar calidad en lugar de features.

Características

- **Su énfasis radica en darle prioridad el cronograma**, que es completamente moldeable**. El límite de tiempo es tan importante que es la prioridad número uno entre todas las otras consideraciones (nunca pasar el deadline del cronograma).**
- Gracias a esta característica, **evita el problema conocido como" 90-90"**. Muchos proyectos llegan al punto en el que está 90% completados y luego se quedan en ese punto por meses o incluso años. Es más eficiente hacer una primera versión más básica y que funcione, aprender de la experiencia, y luego desarrollar una segunda versión mejorada, que, en el mismo tiempo, realizar una versión llena de características.
- **Esclarece cuáles son las prioridades del proyecto**, es decir evita cuestiones existenciales como "¿el botón de impresión debe tener uno o dos pixeles de ancho?".
- **Limita el desarrollo conocido como "gold-plating".** Muchas veces, los desarrolladores, en busca de la iluminación, eligen implementaciones según su propio estándar de calidad, el cual suele demorar mucho más que la versión que requiere el proyecto para encajar en el calendario.
- **Controla el "feature creep".** Es decir, acortando los tiempos de desarrollo, se reducen los cambios que el mercado o el ambiente operacional pueden requerir y que ocasionan un ajuste en el software.
- **Ayuda a motivar a los desarrollares y usuarios finales.** A la gente le gusta sentir que su trabajo es importante, y el sentido de urgencia contribuye a ese sentimiento.

Uso

La construcción de software utilizando Timebox Development consiste en desarrollar un prototipo e ir evolucionándolo hasta conseguir el sistema final.

Generalmente, los Timeboxes duran entre 60 y 120 días ya que períodos más cortos no suelen ser suficientes para desarrollar sistemas significativos, y períodos más largos no logran crear esa pulsión de urgencia propia de la práctica. Aquellos proyectos demasiado extensos para ser desarrollados en 120 días, pueden ser divididos en múltiples proyectos timebox. No es recomendable utilizar el mismo equipo de trabajo para reiterados Timeboxes ya que implica un esfuerzo muy grande para estos.

Como se puede apreciar en la siguiente imagen, un Timebox consta de tres pasos, el Plan (Definición del sistema), la Ejecución (Desarrollo del sistema) y el Cierre. En este último se realiza una evaluación del sistema.

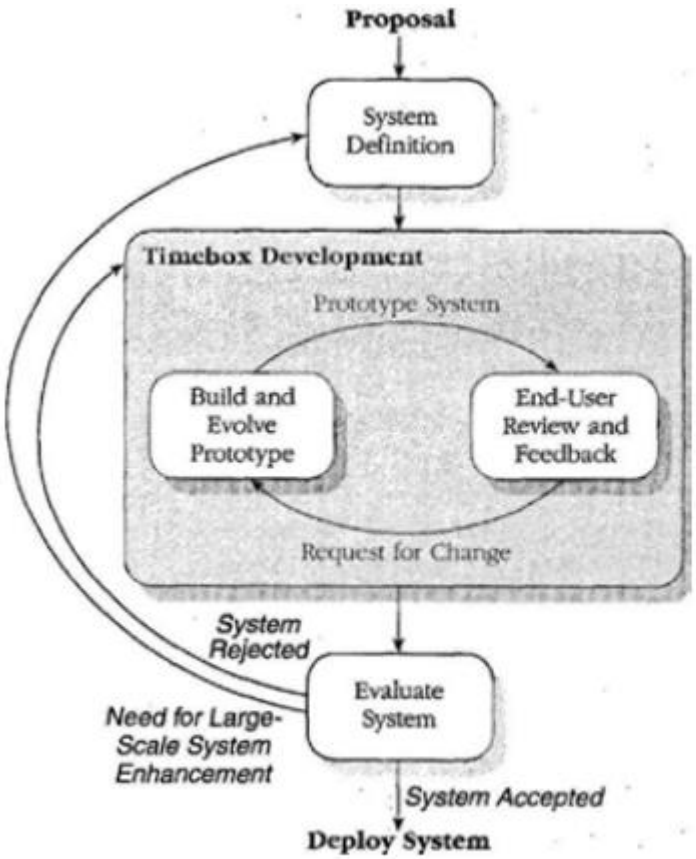


Figure 39-1. *Timebox Development cycle. Timebox Development consists of constructing and evolving an Evolutionary Prototype with frequent end-user interaction,*

Luego de la evaluación del sistema se debe elegir una de las siguientes tres posibilidades:

- Aceptar el sistema y ponerlo en el ambiente de producción.

- Rechazar el sistema porque falló la construcción ya sea por calidad insuficiente, o porque el equipo de desarrollo no fue capaz de implementar la funcionalidad necesaria para el core del sistema. Si esto sucede, la organización puede lanzar un nuevo esfuerzo Timebox development.

- Se rechaza el sistema porque no cumple con las necesidades de la organización que lo construyo. Lo usuarios finales concluyen que el sistema no es lo que querían. En este caso comienza una nueva definición del sistema y arranca nuevamente el ciclo de la imagen anterior.

UNIDAD 4 – SCM (SOFTWARE CONFIGURATION MANAGMENT)

Introducción

Algunos de los problemas comunes en la gestión del desarrollo incluyen:

1. **Problemas de versiones** (¿Cuál es la última versión? ¿Qué cambios estamos incorporando? ¿Quiénes autorizaron dichos cambios?).
2. **Pérdida de modificaciones y desarrollos previos.**
3. **Problemas en la instalación del Software.**
4. **Nadie conoce cuáles componentes forman un release.**
5. **Se modifican componentes sin autorización.**

La gestión de la configuración (CM) es una disciplina orientada a administrar la evolución de: **productos, procesos y ambientes.** Su objetivo es establecer y mantener la **integridad** de los productos del proyecto de software a lo largo del ciclo de vida del mismo.

SCM está muy relacionado con SQA ya que las actividades que se realizan en el primero contribuyen a cumplir con los objetivos planteados en el segundo. En algunos proyectos, sucede que ciertos requerimientos de SQA requieren obligatoriamente el cumplimiento de actividades de SCM específicas.

Definiciones básicas

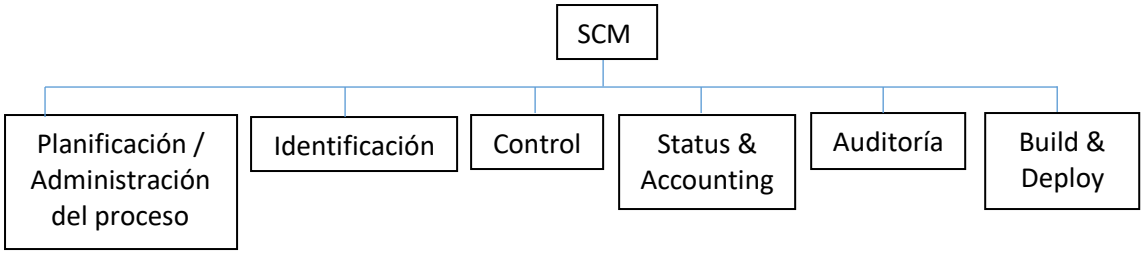
Ítem de configuración: Es cualquier artefacto que vamos a querer administrar en nuestro proyecto (puede ser de producto, de proceso). Para cada ítem se conoce información sobre su configuración (nombre, versión, autor, fecha de modificación, fecha de creación, historial, descripción, ruta, etc.). A continuación, se muestran algunos ejemplos de ítems de configuración:

- Código fuente (de producto)
- Scripts DB / Usuarios y Roles (de producto)
- Tests (de producto)
- Manual de usuario (de producto)
- Logs (de producto)
- Planes (Implementación/Proyecto/SCM/Riesgos) (de proceso)
- Resultados de testing (de proceso)
- Estimaciones (de proceso)
- Equipo (de proceso)
- Resultados de testing (de proceso)
- Alarmas (de proceso)

Configuración: Es el conjunto de ítems de configuración del proyecto en un momento dado.

Baseline (Línea Base): Es una configuración integra, totalmente validada en el ciclo de desarrollo que puede tomarse como punto de referencia para una siguiente etapa del ciclo.

Actividades en SCM



Administración del Proceso

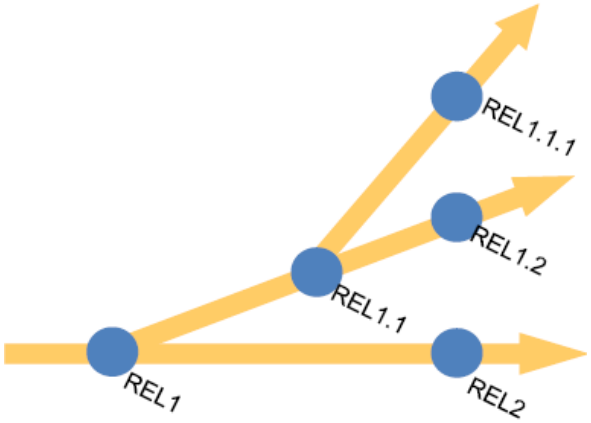
Para planificar el proceso de SCM de un proyecto primero se necesitan llevar a cabo una serie de tareas (entre otras):

1. Comprender el contexto organizacional y las relaciones entre los elementos ya que SCM interactuará con muchos de estos elementos.
2. Luego hay que identificar cuáles son las restricciones y las mejores prácticas para el proceso de SCM.
3. Para evitar confusiones acerca sobre quien ejecutará determinadas actividades o tareas, las organizaciones involucradas en el proceso de SCM deben especificar que entidades serán responsables de cuáles tareas.
4. En esta etapa de planeamiento de SCM, se debe identificar que recursos se utilizarán para llevar a cabo las tareas de SCM, así como también como es que afectan estas tareas al cronograma del proyecto.
5. Se hace la selección de las herramientas que darán soporte a este proceso. Entre ellas se encuentran: the SCM Library, the software change request (SCR) and approval procedures, code and change management tasks, reporting software configuration status and collection SCM measurements, software configuration auditing, managing and tracking software documentation, performing software builds, managing and tracking software releases and their delivery.

El resultado del planeamiento del proceso de SCM para un determinado proyecto se registra en un Software Configuration Management Plan (SCMP). Este es el documento formal que sirve como referencia para el proceso de SCM. El mismo es mantenido durante todo el ciclo de vida del software. En un SCMP se encuentra todo lo contenido en las tareas mencionadas anteriormente, y específicamente (entre otras):

- Cuáles son los “ítems” de configuración y cuando ingresan a formar parte del sistema de CM.
- Los procedimientos para crear “builds” y “releases”.
- Reglas de uso de la herramienta de CM.
- Define estándares de nombres, jerarquías de directorios, estándares de versionamiento.
- El rol del administrador de la configuración.
- El contenido de los reportes de auditoría y los momentos en que estas se ejecutan.

El proceso de SCM de un proyecto debe definir patrones posibles de ramificaciones con el objetivo de controlar los releases.



Identificación de la configuración

Consiste en definir cuáles ítems de configuración serán controlados por la gestión de la configuración. A su vez, se debe definir qué información es relevante para cada uno (nombre, versión, autor, etc)

En esta actividad, también se contempla el identificar líneas bases, ya que los futuros cambios que quieran realizarse sobre estas deberán pasar por un proceso formal.

Finalmente, debe establecerse las bibliotecas en las que se guardarán los IC.

Control de la Configuración

Los sistemas de SW son sujetos a constantes pedidos de cambio ya sea por parte de los usuarios, clientes, o mismo el equipo de desarrollo.

El propósito del control de cambios es asegurar que los ítems de configuración mantienen su integridad ante los cambios a través de un procedimiento que consta de:

1. La identificación del propósito del cambio (Análisis).
2. La evaluación del impacto y aprobación del cambio (Impacto/Factibilidad).
3. La planificación de la incorporación del cambio (Planificación).
4. El control de la implementación del cambio y su verificación (Ejecución).
5. El archivo de información sobre el cambio (el cual muestra la integridad).

Solicitud de cambio (Change request)

La definición de un formulario de solicitud de cambio es parte de la definición del proceso de CM. Esta solicitud tiene por objetivo registrar: el cambio propuesto, quien lo propuso, la razón de porque el cambio fue solicitado y la urgencia del mismo (establecida por el solicitante). También se utiliza para registrar la evaluación del cambio, el análisis del impacto y las recomendaciones de los equipos.

Ejemplo de Proceso genérico de administración de cambios

– Se solicita el cambio

- Puede ser solicitado tanto por un usuario como un desarrollador. En general, solo se aceptan cambios de personas autorizadas a pedirlo.
- Si la solicitud no está completa, es decir, no cumple con las normas definidas en la definición del proceso de CM, se desestima, caso contrario pasa a la siguiente fase.

– El cambio es evaluado y comparado contra los objetivos del proyecto (Análisis)

- Si luego del análisis se concluye que el cambio se rechaza, se debe indicar el motivo.

– Un grupo de asesores evalúa el impacto y la factibilidad del cambio, y lo aprueba o rechaza.

- Si se rechaza, se puede dejar como parte de una “Segunda Fase” del proyecto, o incorporarlo en una segunda iteración de desarrollo.

– Si es aceptado, se replanifican las tareas a realizar y se le asigna recursos para

Resolverlo (Planificación/Ejecución)

- Si luego de la ejecución, el cambio no está completo, se itera en este paso hasta que quede completo.

– El cambio implementado deberá ser revisado en auditorias.

Status & Accounting de la Configuración

El objetivo de esta actividad es registrar y reportar la información necesaria para administrar la configuración de manera efectiva. Entre las principales tareas se encuentran las de:

- Listas los ICs aprobados.
- Mostrar el estado de los cambios que fueron aprobados.
- Reportar la trazabilidad de todos los cambios efectuados al baseline

Auditoría de la configuración

El propósito de la auditoria es realizar una verificación del estado de la configuración a fin de determinar si se están cumpliendo los requerimientos especificados.

Existen tres tipos de auditoría:

- **Funcional**, que verifica el cumplimiento de los requerimientos. En software puede ser efectuada a través de pruebas funcionales o técnicas (testing).
- **Física**, que verifica el ítem para ver si es consistente con la documentación de su configuración.
- **De Proceso,** que verifica que se haya cumplido el proceso de SCM.

Software Release Management and Delivery (Build & Deploy)

La gestión de “Releases” hace referencia a dos temas:

- **Software Building:** Es la actividad de combinar versiones correctas de ítems de configuración, utilizando la configuración adecuada, en un programa ejecutable.
- **Release Managment:** Comprende la administración, identificación y distribución de los elementos de un producto, por ejemplo, un programa ejecutable, documentación, notas de reléase, y datos de configuración.

Lectura – But I Only Changed One Line of Code!

Introducción

Para darnos una introducción a SCM, el autor usa la analogía del proceso que realizó un mecánico para reparar la turbina de un avión. Este pudo realizar la reparación ya que contaba con una lista de todas las partes que conforman el motor, el estado de las mismas antes del despegue, y sabía cómo tenía que ser su estado al momento de despegue. Es decir, hubo que identificar la configuración (Trace), controlar (Action) y validar la integridad (Test).

Identification

Menciona que la actividad de identificación es la más importante ya que no se puede administrar elementos que previamente no hayan sido identificados. También nos dice que se debería identificar todos aquellos artefactos que se utilicen en el proceso de desarrollo de software (dogmático). Esto incluye sistemas operativos, compiladores, liberarías, herramientas de desarrollo, ambientes, manuales y otra documentación.

Control

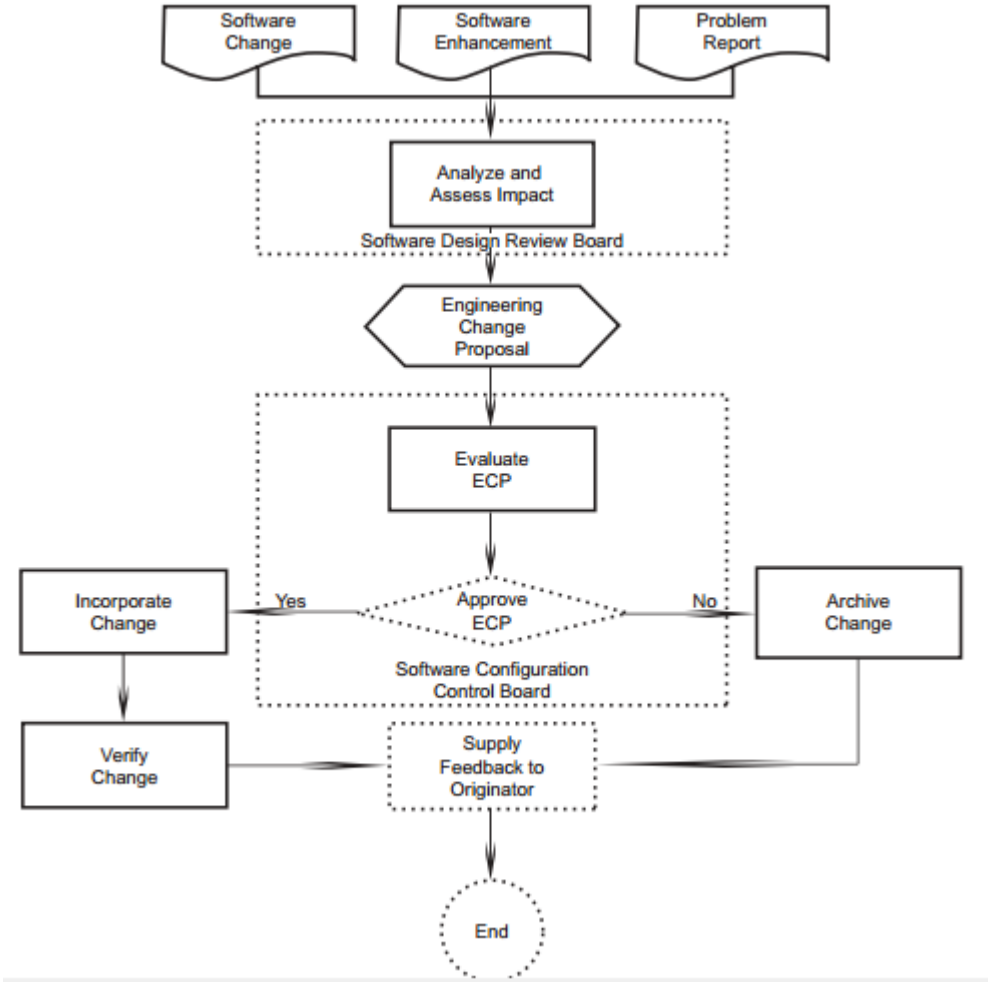
Theron R. Leishman nos narra la vivencia de un programador inexperimentado, que se encontraba realizando mantenimiento de software, el cual fue influenciando por un stakeholder de la aplicación a la que daba soporte para realizar un pequeño ajuste en una sección del programa. Ante la presión del interesado, y por la ansiedad de sumar unos puntos extras en su compañía, el desarrollador realiza el cambio, una modificación de tan solo una línea de código.

A las 2am de la mañana siguiente, en plena corrida de un proceso en producción, se determinó que ese cambio de una línea de código había hecho que la aplicación cesara su funcionamiento.

Este problema fue que ni la organización ni los desarrolladores estaban controlando los elementos del software. Cualquiera podía hacer cambios al software a voluntad. Y como si eso fuera poco, se podían migrar esos cambios al ambiente productivo sin, o con muy poco testing.

El control de la configuración cosiste en asegurar que los cambios que se realizan a las partes del software, solo ocurren una vez que estos han sido analizados, evaluados, revisionados y aprobados por un grupo autorizado.

Este grupo de gente, conocido como change control borad (CCB), tiene el objetivo de asegurar que cada “change request” acerca de un SCI es apropiadamente considerado y coordinado antes de ser incorporado. Para tomar una decisión, ellos evalúan los impactos de cada propuesta de cambio y deciden si vale la pena llevaras a cabo. A continuación, se puede observar una imagen de este proceso:



Status & Accounting

Esta actividad es la responsable de trackear y dar mantenimiento a la data correspondiente a cada SCI. Incluye el tracking de los cambios a los SCI's, y provee la habilidad de determinar el status de cada ítem en cualquiera de las etapas del desarrollo o mantenimiento de software. Definir reportes que muestren la configuración del software es una tarea crítica en este proceso.

Auditing

La auditoría ayuda a verificar que las políticas de desarrollo de software, procesos y procedimientos se están llevando a cabo. Verifica que el software se está construyendo de acuerdo a los requerimientos, estándares, o acuerdos de contrato

UNIDAD 5.0 – SQA (SOFTWARE QUALITY ASSURANCE)

Lectura + PPT– Process & Product Quality Assurance +

Objetivo

El objetivo del aseguramiento de la calidad de procesos y productos (PPQA) es el de proporcionar al personal y a la administración una visión objetiva de los procesos y productos de trabajo asociados.

Esta visión clara y objetiva permitirá a los responsables realizar una toma de decisiones apropiada para reducir el costo de no calidad.

Figure 18.a. Traditional Cost of Quality Model

Notas Introductorias

En las proccess áreas de PPQA se involucran las siguientes actividades (es decir, la función de PPQA involucra):

- Evaluación objetiva de los procesos ejecutados, los productos construidos, y los servicios brindados en función de su descripción de los procesos involucrados, estándares, y procedimientos.
- Identificación y documentación de las no-conformidades detectadas.
- Proporcionar 'feedback' al personal y gerentes del proyecto sobre los resultados de las actividades de aseguramiento de la calidad.
- Garantizar que las no-conformidades son tratadas.

Los proccess áreas de PPQA tienen como objetivo la entrega de productos de alta calidad, proporcionando al personal del proyecto ya los gerentes de todos los niveles una visibilidad y retroalimentación apropiadas sobre los procesos y productos de trabajo asociados a lo largo de la vida del proyecto.

Las prácticas en las 'proccess áreas' 'PPQA' aseguran que los procesos planificados se implementan, mientras que las prácticas en las 'proccess áreas' de 'Verificación' garantizan que se cumplan los requisitos especificados. Estas dos áreas de proceso en ocasiones pueden abordar el mismo producto de trabajo, pero desde diferentes perspectivas. Los proyectos deben aprovechar la superposición para minimizar la duplicación de esfuerzos, teniendo cuidado de mantener perspectivas separadas.

Equipo de trabajo en SQA

En las evaluaciones de PPQA, la objetividad es crítica para el éxito del proyecto. Es por esto, que tradicionalmente, los equipos de trabajo que se conforman para realizar quality assurance son independientes al proyecto (es decir, lo realiza personal ajeno a mi organización). Algunas reglas para conforman un equipo de SQA:

- En organizaciones con cultura "quality oriented" la tarea puede ser ejecutada por pares.
- El que "produce/ejecuta" no realiza QA (se pierde objetividad).
- Todos los participantes de actividades de SQA deben ser debidamente entrenados.
- Los resultados de las tareas de SQA deben tener un canal independiente de reporte a la dirección.

SQA en el SDLC (Software Development Life Cycle)

El aseguramiento de la calidad (quality assurance) debería comenzar en las fases más tempranas de un proyecto para poder, establecer planes, procesos, estándares y procedimientos que le darán valor al proyecto y permitirán satisfacer los requerimientos del proyecto y la organización. Estas actividades tienen como objetivo detectar problemas y no-conformidades lo antes posible.

Cuando se identifican problemas de no-conformidades (incumplimiento), primero se abordan en el proyecto y se resuelven allí si es posible. Los problemas de incumplimiento que no pueden resolverse en el proyecto se escalan hasta un nivel de gestión adecuado para su resolución.

Los resultados de realizar la actividad de software quality assurance son:

- Reporte de evaluación “compliance”.
- “Lecciones aprendidas” para mejora de procesos.
- Seguimiento de las no-conformidades.
- Mantenimiento de “Quality Trends”

Lectura + PPT– Tipos de Revisiones de SQA (When two eyes are not enough)

El objetivo de las **peer reviews** es remover los defectos de los productos de trabajo realizados en el desarrollo de SW, de manera temprana y eficiente. Estas involucran revisiónes metódicas de estos productos, que permiten identificar los defectos y áreas donde los cambios son necesarios.

Existen distintos tipos de peer reviews. Cada una tiene su grado de formalidad, y una estructura que se adapta a las distintas situaciones:

Inspección

Durante una inspección los siguientes roles son utilizados:

- **Moderador:** Este es el líder de la inspección. El moderador tiene previsto la inspección y coordina la misma.
- **'Autor/es:'** La persona que creó el producto que se inspecciona.
- **Inspector:** La persona que examina el producto para identificar posibles defectos.
- **Lector:** La persona que, de manera uniforme, lee los documentos.
- **Escriba/Registrador:** Quien documenta los defectos (findings) que se encuentren durante la inspección.

Las etapas en el proceso de las inspecciones son:

- **Planificación:** La inspección se planea por el moderador.
- **Reunión general:** El autor describe los antecedentes del producto.
- **Preparación**: Cada inspector examina el producto para identificar posibles defectos.
- **Reunión de Inspección:** Durante esta reunión, mientras que el lector lee (uniformemente) el producto con el objetivo de que todos los participantes tengan la misma interpretación acerca de cada sección del mismo; los inspectores remarcan los defectos que encuentran en cada parte. Cada vez

que haya un 'finding', el escriba lo registra. Al finalizar la reunión, el equipo llega a un acuerdo (validación) de los cambios que deberá realizar el autor

- **Repetición del trabajo:** El autor realiza cambios en el producto de acuerdo a los planes de acción de la reunión de la inspección.
- **Seguimiento:** Los cambios del autor son revisados para asegurarse de que todo está correcto.
- El proceso es finalizado por el moderador cuando satisface algunos criterios de salida predefinidos.

Las inspecciones son el método que más findings encuentran. Aproximadamente, se encuentran entre 16 y 20 defectos por cada mil líneas de código, en comparación con otras técnicas de revisión más informales en las que solo se encuentran hasta 3 por cada mil líneas de código.

Las inspecciones son el método más costoso, tanto por el tiempo, como por la cantidad de recursos que requiere. Generalmente se hacen inspecciones sobre código (code reviews), pero podrían realizarse sobre elementos que no fueran código.

El feedback de una inspección es constructivo, se critica el componente y no la persona. Entre sus beneficios se encuentran:

- Encontrar los defectos en forma temprana.
- Training para equipo de trabajo (cuyos miembros podrían participar como 'oyentes' de la inspección).
- Encontrar aquellos defectos que el testing no podría encontrar.
- Da cobertura de código completa.
- Va directamente a la causa raíz del problema, no al síntoma (falla) o manifestación del defecto

Team Review

Son "inspecciones light". No requieren de la participación roles tales como líder (el mismo autor podría liderar la team review), lector, ni tampoco de una validación al cierre de la sesión.

Los participantes reciben el material a revisar varios días antes de la reunión formal para poder interpretarlo. Durante la sesión de trabajo, el equipo notifica los defectos encontrados, y el escriba los registran a media que van surgiendo, utilizando las formas de que la organización haya adoptado.

Walkthrough

Es como una Team Review, pero liderada por el autor. El interés por realizar la revisión parte de este, es quien describe el 'work product' a un grupo de pares y solicita sus comentarios. Las modificaciones al componente son también responsabilidad del autor. Mientras que en una inspección se busca alcanzar los objetivos de calidad de un equipo, una walkthrough trata de servir a las necesidades del autor.

En este método generalmente no se define ningún procedimiento, ni requiere de una gestión especial, métricas o reportes. Puede ser de gran utilidad en la etapa de mantenimiento ya que el autor puede requerir de la atención de los revisores sólo en las porciones del entregable que han sufrido cambios. Sin embargo, se corre el riesgo de pasar por alto

secciones que hayan sido modificadas, pero que el autor desestima, ya que no presenta interés equitativo.

Pair Programming

Es la práctica que consiste en que dos desarrolladores trabajen en el mismo producto, al mismo tiempo. Esto facilita la comunicación, y lleva a un trabajo superior al realizado por una persona ya que "dos cabezas piensan mejor que una".

Se puede decir que las 'revisiones' son tiempo real. Sin embargo, esta técnica no es específicamente una técnica de revisión sino más bien una estrategia de desarrollo.

Peer Deskcheck

En esta práctica, el work product sólo es examinado por una persona (además del autor). El autor se desentiende completamente de la revisión, y desconoce el criterio con el cual es revisionado su trabajo.

Este método depende complemente del conocimiento del revisador. El autor no está presente al momento de la revisión, por lo que no puede responder preguntas o participar de una discusión que lo ayude a encontrar defectos adicionales.

Passaround

Es como el peer deskcheck, pero colectivo. Hoy en día, en esta práctica es muy, común contar con un espacio de trabajo común en donde los revisadores van colaborando (Ejemplo: un 'gdoc' en el que se ve plasmado un trabajo y varios revisadores van dejando sus 'annotations', para que luego el autor se nutra de las mimas).

Ad Hoc

"Che, no tenes 15 y me ayudas a revisar esta función".

Lectura – Seven Deadly Sins of SW Reviews

Se describen siete problemas comunes que reducen la efectividad de las revisiones de software.

Los participantes no entienden el proceso de revisión

En ocasiones sucede que los participantes no saber cómo liderar o contribuir a las revisiones de software. También ocurre que estos cuenten con distinto grado entendimiento de cuáles son sus roles, responsabilidades, o actividades a realizar durante la revisión. Los miembros del equipo podrían no saber cuál de sus 'software work products' debería ser revisionado, ni qué tipo de 'approach' es el más apropiado según el caso.

La solución consiste en capacitar a los miembros del equipo para que todos tengan el mismo grado de entendimiento del proceso de revisión.

La revisión crítica al autor en vez de al producto.

Se da cuando la revisión conlleva tratar temas personales como la habilidad o estilo del autor. Esto genera una confrontación que pasa por alto la raíz del problema, y repercute en los

sentimientos y motivación del autor, quién más tarde podría ser el revisor de uno de sus revisores y querer tomar venganza.

La solución consiste en darle el enfoque correcto a la revisión. Esta debería enfocarse en utilizar la sabiduría colectiva, y la experiencia del grupo encargado de la revisión para mejorar la calidad del producto, en vez de iniciar una batalla de egos e inteligencia entre el autor y sus revisores.

Las revisiones no son planeadas

Sucede que, muchas veces, las revisiones no aparecen en la WBS o el cronograma de trabajo. Y si llegaran a aparecer, lo hacen como milestones en vez de como tareas. Como, por definición, los milestones consumen 0-tiempo de proyecto (y las revisiones no), esto conlleva a que el proyecto se atrase.

La solución consiste en realizar el cronograma de trabajo adecuadamente, estimando el tiempo para las posibles revisiones. No haber planeado revisiones, no significa que más tarde el proyecto no las necesite.

La reunión de revisión termina en de "Problem-Solving"

Nuevamente, es un problema de enfoque. Las revisiones deberían priorizar encontrar defectos, pero no en cómo deben ser arreglados.

La solución consiste en darle el enfoque correcto a la revisión. Si el problema no puede ser resuelto en 1 minuto, entonces no debería gastarse más esfuerzo en pensar su solución, y se pasar al siguiente defecto en vez de seguir perdiendo el tiempo destinado para la revisión.

La revisión no es preparada

Ocurre cuando no se destina el tiempo necesario para preparar el entorno utilizado para la revisión. Como toda actividad, tiene una etapa de preparación en la cual deben establecerse las bases para que la actividad pueda ser ejecutada. En este caso podría ser entender el contexto, preparar la sala, tener un papaer con los posibles lugares en los que hay errores, etc.

Solución: Como el 75% de los defectos encontrados durante inspecciones, son hallados durante la preparación individual, esta revisión (por ejemplo) tiene mucho overhead sino se realiza una adecuada preparación previa a la reunión. Es por esto que el moderador comienza la reunión recolectando los tiempos de preparación incurridos por todos los participantes. Si este considera que el tiempo de preparación es inadecuado, podría desestimar esta reunión y solicitar agendarla para otro momento.

Hay que destinar siempre un tiempo para la preparación, teniendo en cuenta que el tiempo incurrido para esta actividad puede ser capitalizado por el hecho de no tener gastar tiempo en explicar a los colegas de trabajo ciertas cuestiones triviales.

Participación de gente inadecuada

Se da cuando los participantes no tienen las habilidades ni conocimientos necesarios para encontrar defectos, ya que sus aportes a la revisión son mínimos. Aquellos participantes que solo están para aprender podrán beneficiarse, pero probablemente no mejoren la calidad del producto (objetivo principal de la revisión).

También se da cuando participan muchos revisores, ya que es muy difícil que se llegue a un consenso.

Solución: Armar un equipo de 3 a 7 participantes. Entre estos participantes deberían estar todos los responsables de llevar a cabo el 'work product' (por ejemplo: la persona que capto los requerimientos, el analista funcional y el desarrollador). El administrador de la reunión tiene que ser activo, no puede simplemente estar ahí para "ver como se está llevando a cabo la revisión".

Revisores se enfocan en el estilo y no en la sustancia

Lo mismo que pasaba con criticar al autor, se pierde el objetivo principal de la revisión, que es encontrar defectos (de lógica), por el hecho de armar un debate acerca del estilo del 'work product'.

UNIDAD 5.1 TESTING

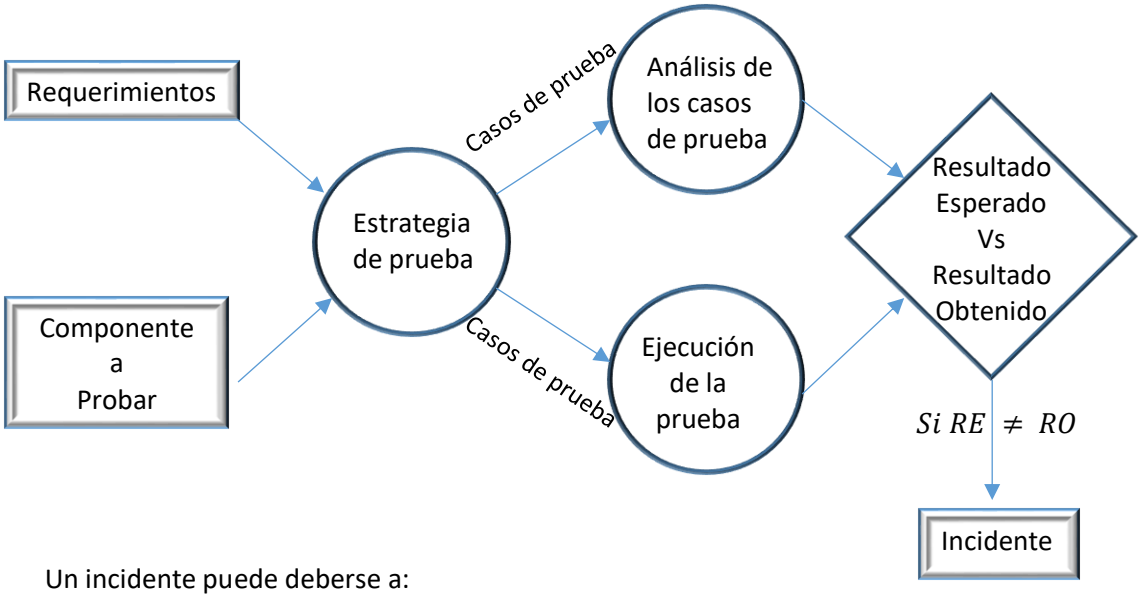
Introducción

Según el IEEE, el "testing" es una actividad en la cual un sistema o componentes es ejecutado bajo condiciones específicas, los resultados de dicha ejecución son observados o registrados, y a partir de los mismos, se realiza una evaluación de algún aspecto de sistema o componente. Se deduce que en proceso dinámico.

Informalmente, el objetivo principal del testing es encontrar fallas en el software. Algunas características que se buscan en el testing:

- Hacerlo lo más eficiente posible.
- Lo más barato posible.
- Lo más rápido posible.
- Reproducible.
- Lo más completo posible.
- Encontrar las fallas más importantes.
- "Sin falsos positivos" (No detectar fallas que en realidad no son).

Proceso de testing



Un incidente puede deberse a:

- Un problema del estado de prueba.
- Un problema en un caso de prueba
- Entorno y configuración del ambiente.
- Un error en la prueba

Si nada de esto ocurre, entonces el incidente lo produjo una ***falla.***

Recordar: El testing no puede encontrar problemas en los requerimientos.

Conceptos:

- ***Equivocación:*** Acción humana que produce un defecto.
- ***Defecto:*** Algo que funciona mal, o la ausencia de cierta característica.
- ***Falla:*** Resultado de ejecución incorrecto. Es un error producido en el software, es decir, este tiene un defecto

- Una equivocación lleva a uno o más defectos que están presentes en el código.
- Un defecto lleva a cero, una o más fallas.
- La falla es la manifestación del defecto.
- Una falla tiene que ver con uno o más defectos.

- **Casos de prueba:** Son los lotes de datos necesarios para que se dé una determinada condición de prueba.

- **Condición de prueba:** Son descripciones de situaciones a las que el sistema debe responder. Un caso de prueba es una instancia de una condición de prueba.

Depurar es eliminar un defecto que posee el software. En la depuración debemos:

1. **DETECTAR:** dada la falla debemos encontrar el defecto.
2. **DEPURAR:** encontrado el defecto debemos eliminarlo.

3. **VOLVER A PROBAR:** Asegurar tanto que sacamos el defecto como que no hemos introducido otros (regresión).
4. **APRENDER PARA EL FUTURO**

Economía del Testing

Ya que nunca voy a poder demostrar que un programa es correcto, continuar probando es una decisión económica. Es necesario entonces contar con una estrategia para establecer "cuando parar de probar". Algunas de ellas podrían ser:

- Pasa exitosamente el conjunto de pruebas diseñado.
- "Good Enough": cierta cantidad de fallas no críticas es aceptable.
- Cantidad de fallas detectadas es similar a la cantidad de fallas estimadas.

Enfoques de Prueba

Prueba de la Caja Negra

Es una prueba funcional, la cual tiene como punto de partida los requerimientos o especificación. En este proceso de prueba, nos desentendemos completamente del comportamiento o estructura interna del componente, solo nos concentramos en la "entrada/salida". Prueba lo que el software ***debería hacer***.

Como la prueba de caja negra exhaustiva es imposible de alcanzar, tenemos distintos métodos de prueba que nos ayudan a cubrir un extenso conjunto de casos de prueba posibles:

Clases de equivalencias

Podemos suponer que la prueba de un valor representativo de cada clase de equivalencia es equivalente a la prueba de cualquier otro valor. Algunas de estas clases pueden ser:

- **SI/NO:** 2 posibilidades.
- **Rango de valores.** Ej: 100 < Nro. Factura < 200.
- **"Contenido en".** Ej: [1, 3, 5, 11].
- **"Debe ser".** Ej "Primera letra = 'A'".

Condiciones de Borde

La experiencia muestra que los casos de prueba que exploran las condiciones de borde producen mejor resultado que aquellas que no lo hacen.

Clases inválidas

(Numéricos en vez de alfabéticos y viceversa, fechas erróneas, etc).

Integridad del modelo de datos

Pensar el caso de prueba, tomando como punto de partida el DER. Ej: ¿Puedo dar de alta dos clientes con el mismo mail?

Conjetura de Errores

Esta conjetura se hace cuando el tester "sospecha" que puede producirse un error ante una condición determinada. Este método requiere, principalmente, que el tester tenga mucha experiencia.

Variación de eventos

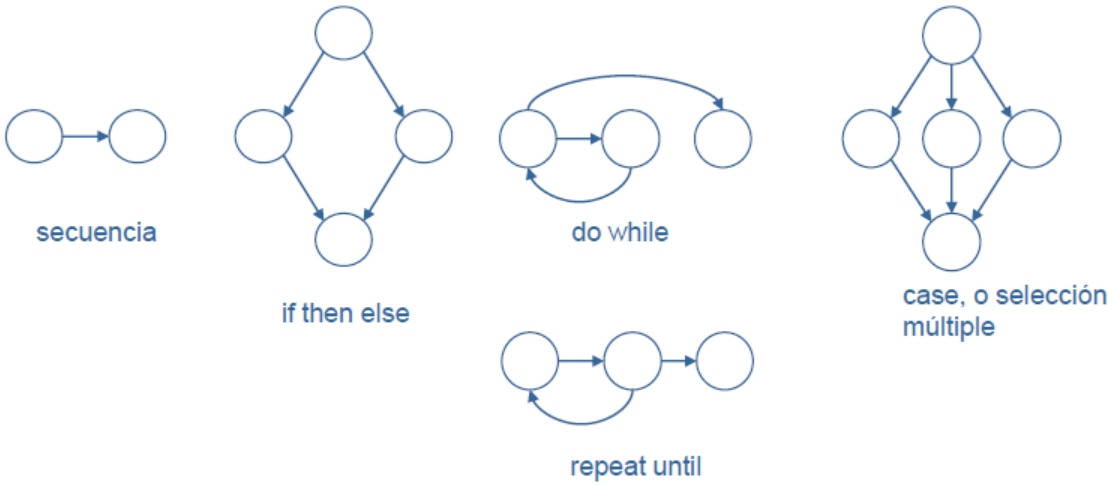
Prueba de Caja Blanca

Es una prueba estructural, la cual tiene como punto de partida el componente a probar, y se basa en cómo está estructurado dicho componente internamente. Es usada para incrementar el grado de cobertura de la lógica interna del componente. Se Prueba lo que el software ***hace***.

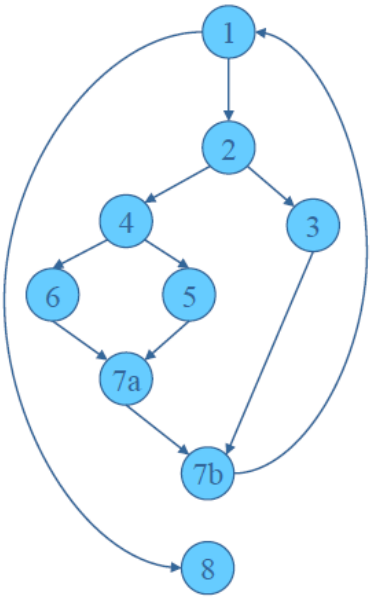
¿Cómo lo pruebo todo? Existen distintos grados de cobertura:

- **Cobertura de sentencias**: Se enfoca en que la prueba ejecute todas las sentencias del código.
- **Cobertura de decisión:** Es más abarcativo que el anterior. La prueba pasa por todos los caminos posibles del programa.
- **Cobertura de condiciones**: Es más abarcativo aún. El test recorre todos los caminos y las variables de decisión ("IF" o "WHILE") toman todos los valores posibles
- **Prueba del Camino Básico:** Prueba todos los caminos independientes. Comprende todas las coberturas anteriores. Esta prueba se basa en la **complejidad ciclomática,** que es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Esta métrica, nos otorga información acerca de la cantidad de caminos independientes $(aritas - nodos + 2)$ que este posee.

Camino Básico: Se representa el flujo de control de una pieza de código a través de un grafo de flujo.



Ejemplo:




```
   procedimiento ordenar
1: do while no queden registros
     leer registro;
2:   if campo1 del registro =0
3:   then procesar registro
        guardar en buffer
        incrementar contador
4:   elsif campo2 del registro =0
5:      then reinicializar contador
6:      else procesar registro
        guardar en archivo
7a:     endif
     endif
7b:enddo
8: end
```

Prueba de la caja gris

No es caja negra porque se conoce parte de la implementación o estructura interna y se aprovecha ese conocimiento para generar condiciones y casos que no se generarían naturalmente en una prueba de caja negra. Este conocimiento es “parcial”, no “total” (lo que sería caja blanca)

Las pruebas de caja gris prueban el SW como si fuera caja negra, pero suman condiciones y casos adicionales derivados del conocimiento de la operación e interacción de ciertos componentes de SW que componen la solución.

V-Model

Este modelo representa un proceso de desarrollo de software que puede ser considerado una extensión del “cascada”. En vez simplemente moverse hacia abajo de manera lineal, luego de la fase de codificación, el proceso escala hacia arriba (formando una ‘V’).

El “V-Model” muestra la relación entre cada fase del ciclo de vida de desarrollo de software con su fase de testing asociada. Estas fases de prueba se detallan a continuación:

Pruebas unitarias: Son ejecutadas para eliminar bugs en unidades (módulos) de código claramente definidas. Estas pruebas verifican que la entidad más pequeña funciona correctamente cuando es aislada del resto de las unidades.

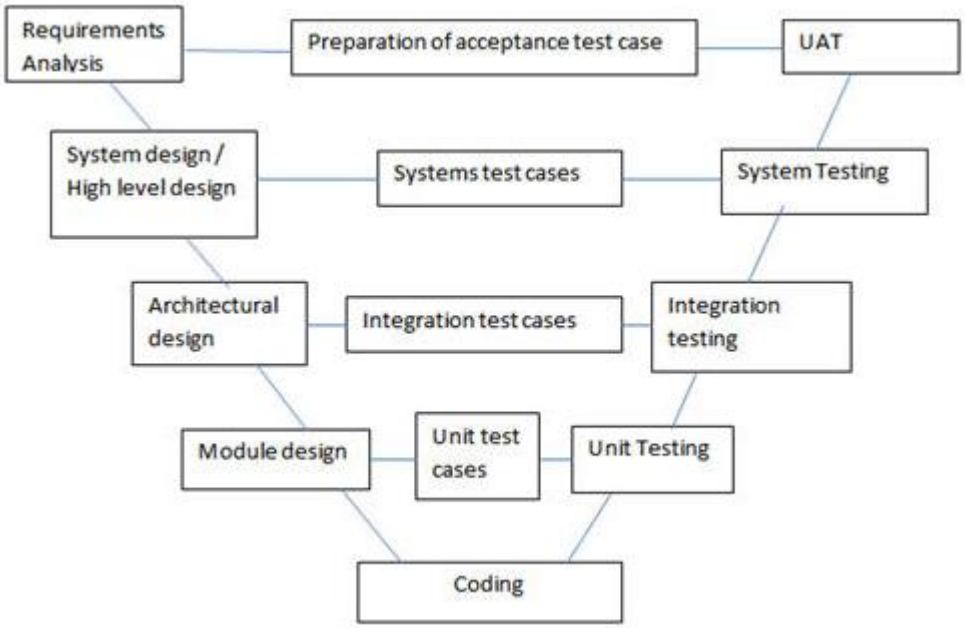
Pruebas integrales: Verifican que las unidades creadas y testeadas de forma independiente pueden coexistir y comunicarse entre ellas. Los resultados de estas pruebas se comparten con el equipo del cliente.

Pruebas de sistema: Su objetivo de asegurar que las expectativas de la aplicación desarrollada se cumplen. En estas pruebas se testea todo el sistema, y se verifica que tanto los requerimientos funcionales como no funcionales se cumplen. En este testing se incluyen las siguientes pruebas:

- **Pruebas de volumen y performance.**
- **Prueba de Stress:** Orientada a someter al sistema a que exceda los límites de su capacidad de procesamiento y almacenamiento para que este en algún punto falle.
- **Pruebas de regresión:** Verifican que luego de introducido un cambio en el código la funcionalidad original no ha sido alterada ("Funciona lo viejo + lo nuevo").
- **Prueba Alfa y Beta: S**e hace entrega una primera versión al usuario que se considera lista para ser probada por ellos. Normalmente tiene muchos defectos, pero es una forma económica de identificarlos (ya que el trabajo lo hace otro). La versión Alfa la hace el usuario en mis instalaciones, y la Beta la hace el usuario en sus instalaciones.
- **Pruebas de seguridad.**

Prueba de aceptación de Usuario: Esta prueba es realizada por los usuarios para verificar que el sistema se ajusta a sus requerimientos (es una prueba de caja negra, las condiciones de prueba están basadas en el documento de requerimientos).

A continuación, el gráfico de un V-Model:



UNIDAD 6 – MÉTRICAS

Conceptos introductorios

- Una **Medida** proporciona una indicación cuantitativa (extensión, cantidad, dimensiones, capacidad o tamaño) de algunos atributos de un proceso o producto. Ej. un programa tiene 10.000 LDC (líneas de código).

- Una **Métrica** es una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Ej. la productividad de este proyecto fue de 500 (LDC/persona-mes).

- Un **Indicador** es una métrica o combinación de métricas que proporcionan una visión más profunda de un proceso, un proyecto de software o de un producto en sí.

Cómo especificar una métrica:

- Nombre
- Unidad de la métrica
- Objetivo
- Interpretación (Ej: 1000 LOC/SPRINT es mucho; <100 es poco).
- Método de medición:
  - Fuente/s de datos.
  - Fórmula (Ej: bugs/sprint).
  - Visualización gráfica.
- Método de recolección (aquí se incluye la frecuencia de cálculo).
- Valores esperados.

Implementar un programa de métricas

Pasos a seguir:

1. Identificar los objetivos (que mi organización quiere alcanzar).
2. Definir las métricas
3. Recolectar datos (históricos, para conocer la situación actual).
4. Automatizar el proceso (si es posible).
5. Utilizar las métricas en la toma de decisiones

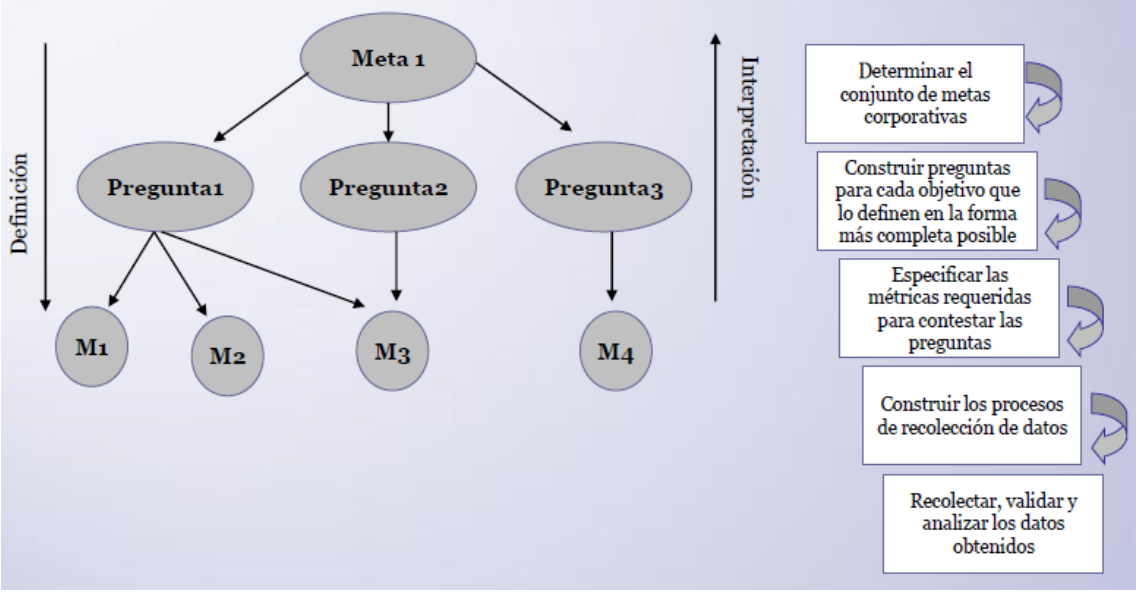
Recomendaciones:

- Comenzar midiendo pocos indicadores, simples.
- Detectar aquellos más relevantes
- No buscar resultados en el corto plazo
- Sostener el esfuerzo de recolección en el tiempo
- En lo posible automatizar la captura de información para la obtención de las métricas
- Utilizar las métricas en la toma de decisiones

GQM – Goal Question Metric

Su objetivo es definir que métricas implementar y como utilizarlas. Para ello, define un modelo de medición basado en una jerarquía de tres niveles:

- **Conceptual level (Goal):** Se define la o las metas (goals) que la organización intenta alcanzar.
- **Operational level (Question):** Se define un set de preguntas (questions) cuya respuesta permiten definir el cumplimiento de las metas.
- **Quantitative level (Metrics):** Se asocia un conjunto de métricas a las preguntas con el objetivo de ayudar a responder estas preguntas y confirmar si las mejoras del proceso cumplieron su objetivo.



Tipos de Métricas de SW

Clasificación:

- **Métricas del Proceso:** El análisis de cómo los procesos se realizan a partir de medidas tomadas en el desarrollo de SW es la base para su posterior mejora. Ej. Métricas de Gestión de Proyectos.

- **Métricas del Producto de SW:** Las métricas sobre el producto están orientadas a medir las características inherentes al mismo. Ej. Tamaño, Calidad del Producto

El **objetivo** de establecer métricas e indicadores se puede resumir en:

- Detectar áreas problemáticas antes de que se conviertan en críticas.
- Brindar información para la toma de Decisiones.

Métricas orientadas al proceso

Algunos ejemplos:

- Duración promedio de proyectos.
- Cantidad de proyectos segregados por tipo, tamaño, etc.
- Esfuerzo promedio segregado por tipo, duración.
- Defectos introducidos en una fase del ciclo de vida.
- Defectos detectados en una fase del ciclo de vida.
- % de tiempo/esfuerzo/costo dedicado a una fase del ciclo de vida.
- % promedio de desvío en proyectos (En costo o duración).
- **Earned Value.**

El “Earned Value Analysis” (o análisis del valor ganado), es una métrica del proceso de gestión de proyectos cuyo objetivo es medir la performance de un proyecto en términos de alcance, tiempo y costo. Este análisis me permite tomar una “foto” del proyecto en un determinado momento para conocer su estado. Características:

- ¿Cuánto trabajo se PLANIFICO completar? (**Planned Value**).
- ¿Cuánto trabajo ACTUALMENTE se completó? (**Earned Value**).
- ¿Cuánto se "GASTÓ"" en completar el trabajo actual? (**Actual Cost**).

- **Schedule Variance (SV)** = EV – PV
- **Schedule Performance Index (SPI)** = EV/PV
    - Valor esperado: Mayor o igual a 1 (adelantado)
    - Valor no deseado: Menor a 1 (atrasado)
- **Cost Variance (CV)** = EV – AC
- **Cost Performance Index (CPI)** = EV/AC
    - Valor esperado: Mayor o igual a 1

Métricas orientadas al producto

Algunos ejemplos:

- Cantidad de líneas de código (LOC).
- Funcionalidad (Puntos por Función / Use Case Points).
- **Complejidad Ciclomática.**
- Cohesión.
- Acoplamiento.
- Calidad (ISO 9126).
- **Confiabilidad (MTBF, MTTR)**
    - **MTBF**: Mean Time Between Failures
    - **MTTR:** Mean Time To Recovery