Capítulo 2

Software Requirements Specification

SRS (Software Requirements Specification):

- Se requiere de una persona que lo haga, no puede ser automatizado
- Se realiza para gente que no entiende de código, es para acordar con el cliente
- Establece las bases para el acuerdo entre el cliente/usuario y quien suministrará el software
- Hay 3 partes involucradas
    - Necesidades del cliente
    - Consideraciones del usuario
    - ?
- Generalmente hay una gran brecha comunicacional entre las partes, el cliente no entiende el proceso de desarrollo del software y el desarrollador no conoce el problema del cliente ni su área de aplicación, La SRS es el medio para reconciliar las diferencias y especificar las necesidades de cada uno.
- Es necesaria para que el usuario comprenda cuales son sus necesidades, ya que estos muchas veces no saben qué es lo que quieren/necesitan.
- Reduce los costos de desarrollo si esta bien hecho(aparte de reducir también errores y cambios), de lo contrario son más caros de corregir a medida que progresa el proyecto
- **ETAPAS DE LA SRS:**
    - **Análisis del problema o requerimientos**: El desarrollador tiene que entender el problema a resolver, en esta etapa se realizan entrevistas con el cliente y los usuarios, se leen manuales, se estudia el sistema actual y se ayuda al cliente/usuario a comprender nuevas posibilidades.
        - DFD (Diagrama de Flujo de Datos): Es una representación gráfica del flujo de datos a través del sistema.
          ¿Cómo realizar un DFD?:
            1. Identificar las entradas, salidas, fuentes y sumideros del sistema
            2. Trabajar consistentemente desde la entrada hacia la salida
            3. Si se complica => Cambiar el sentido (de la salida a la entrada)
            4. Avanzar identificando los transformadores de más alto nivel para capturar la transformación completa
            5. Cuando los transformadores de alto nivel están definidos, refinar cada uno con transformaciones más detalladas.
            6. No mostrar nunca lógica de control; si se comienza a pensar en término de loops/condiciones; parar y recomenzar.
            7. Etiquetar cada flecha y burbuja, identificar cuidadosamente las entradas y salidas de cada transformador

8. Hacer uso de + y *
9. Intentar dibujar grafos de flujo de datos alternativos antes de definirse por uno.

Errores comunes al realizar un DFD:

- Flujos de datos sin etiquetar
- Flujos de datos omitidos (necesarios para un proceso)
- Flujos de datos irrelevantes (dibujado pero no usado por el proceso)
- Consistencia no preservada por el refinamiento
- Procesos omitidos
- Inclusión de información de control

- Especificación de los requerimientos: Plasmar el entendimiento en la SRS
- Validación: Búsqueda de errores

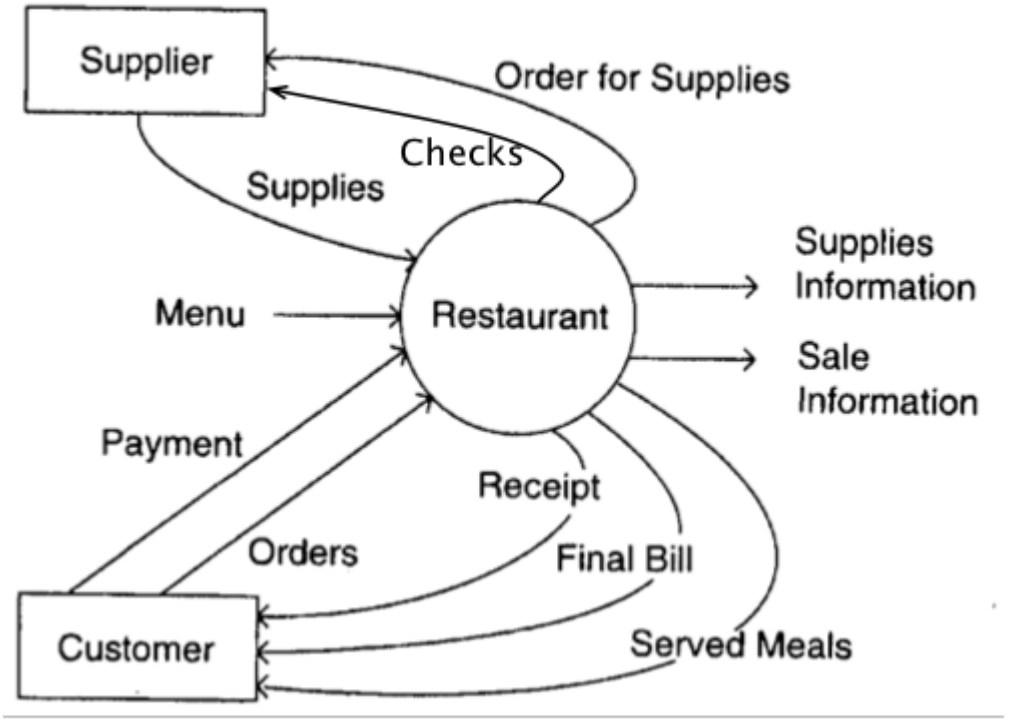
Ejemplo de un modelo de flujo de datos:
La dueña de un restaurante cree que automatizando algunas partes del negocio mejorará la eficiencia de su negocio. Ella también cree que un sistema automatizado puede hacer el negocio más atractivo para el cliente. Por lo tanto ella desea automatizar su restaurante lo más posible.

Aquí viene donde realizamos entrevistas y cuestionarios con el cliente y los usuarios y recolectamos la información en general.

Primero identificamos las partes involucradas:

1. Cliente: La dueña del restaurante
2. Usuarios potenciales: Mozos, operador de la caja registradora

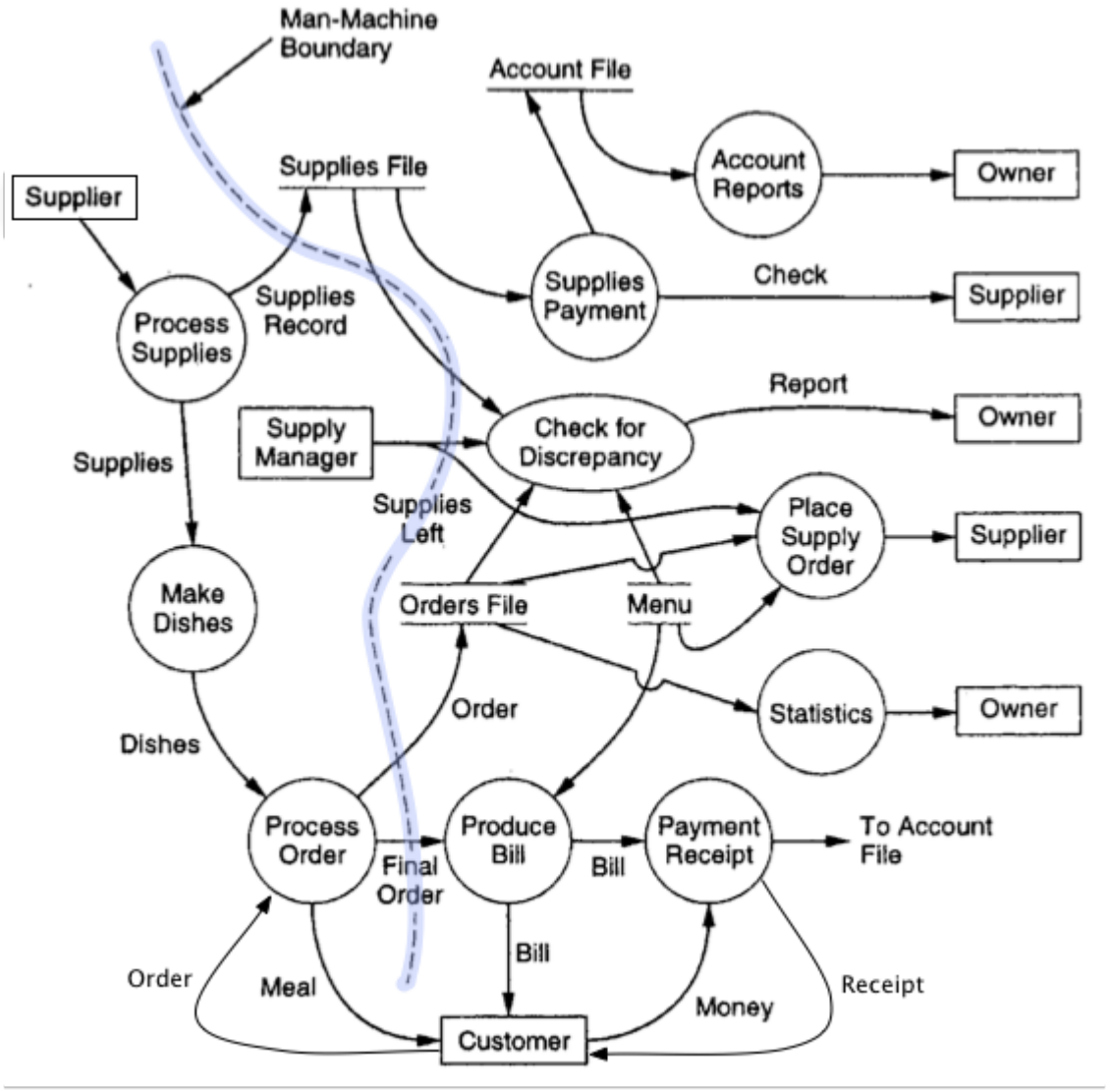
DFD del sistema existente:



Luego de la interacción con la cliente/usuarios se concluye:

- Automatizar la mayoría del proceso de recepción y procesado
- de la orden y cobrado.
- Automatizar la contabilidad y stock.
- Hacer más exacto el procesamiento de suministros de manera
- de minimizar los desperdicios al final del día y maximizar la disponibilidad de órdenes.
- La propietaria sospecha que el personal puede estar robando suministros. Ella desea que el nuevo sistema detecte y reduzca esto.
- La propietaria también desea estadísticas sobre las ventas.

Luego el DFD queda así:



Modelo orientado a objetos:

El sistema es visto como un conjunto de objetos interactuando entre sí o con el usuario.

Objetivo:

- Identificar los objetos (clases) en el dominio del problema
- Definir las clases identificando cual es la información del estado que encapsula
- Identificar relaciones entre los objetos

Pasos más significativos para realizar el análisis del modelo orientado a objetos:

- Identificar objetos y clases.
- Identificar estructuras.
- Identificar atributos.
- Identificar asociaciones.
- Definir servicios.

CARACTERÍSTICAS DE UNA SRS:

- Correcta
- Completa

- No ambigua
- Consistente
- Verificable
- Rastreable (Traceable)
- Modificable
- Ordenada en aspectos de importancia y estabilidad

- **Corrección**: Cada requerimiento representa precisamente alguna característica deseada por el cliente en el sistema final.

- **Completitud:** Todas las características deseadas están descritas
- **No ambigua:** Cada requerimiento tiene exactamente un significado, Si es ambigua los errores se colaran fácilmente
- **Consistente:** Ningún requerimiento contradice a otro (i.e conflictos lógicos, temporales, de dependencias)
- **Verificable:** Si cada requerimiento es verificable, i.e. si existe algún proceso efectivo que puede verificar que el software final satisface el requerimiento.

- **Rastreable:** Se debe poder determinar el origen de cada requerimiento y cómo Este se relaciona a los elementos del software.
    - Hacia adelante: dado un requerimiento se debe poder detectar en qué elementos de diseño o código tienen impacto.
    - Hacia atrás: dado un elemento de diseño o código se debe poder rastrear qué requerimientos está atendiendo.

- **Modificable:** Si la estructura y estilo de la SRS es tal que permite incorporar cambios fácilmente preservando completitud y consistencia.

- **Ordenada en aspectos de importancia y estabilidad:**
    - Los requerimientos pueden ser críticos, importantes pero no críticos, deseables pero no importantes.
    - Algunos requerimientos son esenciales y difícilmente cambien con el tiempo. Otros son propensos a cambiar.
    - => Se necesita definir un orden de prioridades en la construcción para reducir riesgos debido a cambios de requerimientos.

Una SRS debe especificar requerimientos sobre:
- Funcionalidad.
- Desempeño (performance).
- Restricciones de diseño.
- Interfaces externas.

Requerimientos de desempeño:
- Todas las restricciones en el desempeño del sistema de software.
- Requerimientos Dinámicos (especifican restricciones sobre la ejecución):

    - Tiempo de respuesta.
    - Tiempo esperado de terminación de una operación dada.
    - Tasa de transferencia o rendimiento.
    - Cantidad de operaciones realizadas por unidad de tiempo.
    - En general se especifican los rangos aceptables de los distintos parámetros
    - (en casos normales y extremos).

- **Requerimientos Estáticos o de capacidad** (no imponen restricción en la ejecución):
    - Cantidad de terminales admitidas.
    - Cantidad de usuarios admitidos simultáneamente.
    - Cantidad de archivos a procesar y sus tamaños.

Todos los requisitos se especifican en términos medibles => verificable.

Casos de uso:

Requerimientos: Es una condición o capacidad necesaria de un usuario para solucionar un problema o alcanzar los objetivos, esto es lo que debe poseer o cumplir nuestro sistema

Capítulo 3

Arquitectura del Software

Motivación: Es fácil manejar un programa chiquito, como de 100 líneas, pero a medida que el programa se agranda, este es más propenso a errores y a romper el código, es ahí cuando la arquitectura entra al juego.

Todo sistema complejo se compone de subsistemas que interactúan entre sí, La arquitectura del software tiene como objetivo analizar la manera óptima para dividir un sistema y manipularlo de mejor manera.

La arquitectura del software de un sistema es la estructura del sistema que comprende los elementos del software, las **propiedades externamente visibles** de tales elementos y la relación entre ellas. A la arquitectura **no** le interesan los detalles de cómo se aseguran dichas propiedades.

La arquitectura es el diseño del sistema al más alto nivel, a este nivel se hacen las elecciones de tecnología, productos a utilizar, servidores, etc. Por lo que no es posible diseñar los detalles del sistema antes de incorporar estas elecciones.

¿Por qué es útil la arquitectura del software?: Al ser de tan alto nivel, la complejidad está oculta y facilita la comunicación, ya que define un marco de comprensión común entre los distintos interesados(usuarios,cliente,arquitecto,diseñador,etc).
También ayuda mucho para el **Reuso,** ayuda a identificar qué partes pueden ser reutilizadas para otras tareas.

Construcción y evolución: La división provista por la arquitectura servirá para guiar el desarrollo del sistema, dictando cuales partes son necesarias construir primero y cuales ya están construidas, a su vez ayuda a asignar equipos de trabajo, haciendo módulos que interactúan de manera óptima entre sí, ya que la arquitectura también ayuda a decidir cuáles partes necesitan cambiarse y decidir cual es el impacto de tales cambios en otros componentes.

Análisis: Es deseable que propiedades de **confiabilidad y desempeño** puedan determinarse en el diseño.

"No hay una única vista de arquitectura de un sistema"

Hay distintas vistas de un sistema de software, una vista consiste de elementos y relaciones entre ellos y describe una estructura, los elementos de una vista dependen de lo que se quiera destacar.

Hay distintos tipos de vista:
- Módulo
- Componentes y conectores
- Asignación de recursos

Vista de Módulos: Un sistema es una colección de **unidades de código (**los elementos son módulos, por ej clases, paquetes, funciones), la relación entre ellos está basada en el código (parte de, usa a, depende de)

Vista de componentes y conectores:
1. Los elementos son **Entidades de ejecución** denominados componentes (objetos, procesos, .exe, .dll.etc)

2. Los conectores proveen el medio de **interacción** entre las componentes (pipes,sockets,memoria compartida, protocolos, etc)

   Componentes: son elementos computacionales o de almacenamiento de datos
   - Cada componente tiene un nombre que representa su rol y le provee una identidad.

- Cada componente tiene un tipo, los distintos tipos se representan con distintos símbolos.
- Las componentes utilizan interfaces o puertos para comunicarse con otras componentes.

Es importante ser consistentes con los símbolos que se usan para cada componente.

Conectores: son mecanismos de interacción entre las componentes. Describen el **medio** en el cual la interacción entre componentes toma lugar. Un conector puede proveerse por medio del entorno de ejecución (llamada a procedimiento/función). Los conectores pueden también ser mecanismos de interacción más complejos (puertos TCP/IP, RPC).
Los conectores tienen:

- Nombre, que identifica la naturaleza de la interacción, y Tipo, que identifica el tipo de interacción (binaria, n-aria, unidireccional o bidireccional,etc).

Una vista de C & C define los componentes y cómo se conectan entre ellas a través de los conectores. La vista C & C describe una estructura en ejecución del sistema: que componentes existen y cómo interactúan entre ellos en **tiempo de ejecución**.

Vista de asignación de recursos: Se enfoca en **cómo** las unidades de software se asignan a recursos como hardware, sistemas de archivos, gente, etc. Exponen propiedades estructurales como que proceso ejecuta en qué procesador o donde estan los archivos.

Una descripción arquitectónica consiste de vistas de distintos tipos, cada una mostrando una estructura diferente. Distintos sistemas necesitan distintos tipos de vistas dependiendo de las necesidades.

Ej:
análisis de desempeño => C&C
asignación de recursos => asignación de recursos
planeamiento => módulos

La vista C & C (Componentes y conectores) es la que casi siempre se hace, y se ha transformado en la vista principal.
La vista de los módulos se verá más adelante, en diseño de alto nivel.

Estilos arquitectónicos para la vista de C & C: Algunas estructuras son generales y son útiles para una clase de problemas.

Un **estilo arquitectónico** define una familia de arquitecturas que satisface las restricciones de ese estilo. Distintos estilos pueden combinarse para definir una nueva arquitectura.

El más clásico es el de **Tubos y Filtros** (pipe and filter):

Adecuado para sistemas que fundamentalmente realizan **transformaciones de datos**, un sistema que utiliza este estilo utiliza una red de transformadores para realizar el resultado deseado, este tiene un solo tipo de componente: filtro. Tiene un solo conector: el tubo, Un filtro realiza transformaciones y le pasa los datos a otro filtro a través de un tubo.

Un filtro es una entidad independiente y asíncrona (se limita a consumir y producir datos). No necesita saber la identidad de los filtros que envían o reciben los datos

Un **tubo** es un canal unidireccional que transporta un flujo de datos de un filtro a otro, este solo conecta 2 componentes. Los filtros deben hacer buffering y sincronización para asegurar el correcto funcionamiento como productor y consumidor.

Restricciones:

- Cada filtro debe trabajar sin conocer la identidad de los filtros productores o consumidores
- Un tubo debe conectar un puerto de salida de un filtro a un puerto de entrada de otro filtro
- Un sistema puro de tubos y filtros usualmente tiene su propio hilo de control

Estilo de datos compartidos: Tiene dos tipos de componentes:

1. Repositorio de datos: Provee almacenamiento permanente confiable
2. Usuario de datos: Acceden a los datos en el repositorio, realizan cálculos y ponen los resultados otra vez en el repositorio.

La comunicación entre los usuarios de los datos solo se hace a través del repositorio.
En este estilo solo hay un tipo de conector: lectura/escritura.

Dos variantes principales:

- Estilo pizarra: Cuando se agregan/modifican datos en el repositorio, se informa a todos los usuarios
- Estilo repositorio: El repositorio es pasivo.

Estilo cliente-servidor:

Dos tipos de componentes:

1. Clientes: Sólo se comunican con el servidor, pero no con otros clientes, La comunicación siempre es iniciada por el cliente quien le envía una solicitud al servidor y espera una respuesta de este. (comunicacion asincrona)
2. Servidores

Solo un tipo de conector: solicitud/respuesta (asimétrico).

Usualmente el cliente y el servidor residen en distintas máquinas.

Este estilo tiene una estructura multi-nivel:

1. Cliente: Contiene a los clientes
2. Intermedio: Contiene las reglas de servicio
3. Base de datos: reside la información.

Otros Estilos

Estilo publish-subscribe

Estilo peer-to-peer

Estilo de procesos que se comunican.

Documentación de diseño arquitectónico:

Los diagramas son un medio adecuado para discutir y crear diseños, sin embargo no son suficientes para documentar el diseño arquitectónico, El documento debe expresar las vistas y restricciones.

Organización del documento:

1. Contexto del sistema y la arquitectura: Provee el contexto general, establece el alcance del sistema, los actores principales, las fuentes y consumidores de datos.

2. Descripción de las vistas de la arquitectura: Uno por cada una de los distintos tipos de vistas que se eligieron representar.

a. Presentación principal de la vista: Casi siempre contiene la descripción gráfica.

b. Catálogo de elementos: Provee más información sobre los elementos que se muestran en la presentación principal. Por cada elemento. Describe su propósito y sus interfaces.

c. Fundamento de la arquitectura: Justificaciones de las decisiones/elecciones realizadas. Podría proveer también una discusión sobre las alternativas consideradas y descartadas.

d. Comportamiento: Las vistas describen la estructura pero no el comportamiento. Algunas veces es necesario dar una idea del comportamiento real del sistema/componente en algunos escenarios.

e. Otra información: Decisiones dejadas intencionalmente para el futuro.

3. Documentación transversal a las vistas: Describe cómo los elementos de las distintas vistas se relacionan entre sí. También justificación de las vistas elegidas.

Método de análisis ATAM (Architecture Tradeoff Analysis Method): Analiza las propiedades y las concesiones entre ellas.

Pasos Principales:

1. 1. Recolectar escenarios:
    a. Los escenarios describen las interacciones del sistema.
    b. Elegir los escenarios de interés para el análisis.
    c. Incluir escenarios excepcionales sólo si son importantes.

2. Recolectar requerimientos y/o restricciones:
    a. Definir lo que se espera del sistema en tales escenarios.
    b. Deben especificar los niveles deseados para los atributos de interés (preferiblemente cuantificados).
3. 3. Describir las vistas arquitectónicas:
    a. Las vistas del sistema que serán evaluadas son recolectadas.
    b. Distintas vistas pueden ser necesarias para distintos análisis.

4. Análisis específicos a cada atributo:
    a. Se analizan las vistas bajo distintos escenarios separadamente para cada atributo de interés distinto.
    b. Esto determina los niveles que la arquitectura puede proveer en cada atributo.
    c. Se comparan con los requeridos.
    d. Esto forma la base para la elección entre una arquitectura u otra o la modificación de la arquitectura propuesta.
    e. Puede utilizarse cualquier técnica o modelado.
5. Identificar puntos sensitivos y de compromisos:
    a. Análisis de sensibilidad: cuál es el impacto que tiene un elemento sobre un atributo de calidad.
    b. Los elementos de mayor impacto son los puntos de sensibilidad.
    c. Análisis de compromiso:
    d. Los puntos de compromiso son los elementos que son puntos de sensibilidad para varios atributos.

Capítulo 4
Diseño del Software

Definición/Prefacio: El diseño es el lenguaje intermedio entre los requerimientos y el código, este comienza una vez que los requerimientos están definidos y antes de la implementación (codificación).
Es el desplazamiento del dominio del problema al dominio de la solución, por lo que es una actividad muy creativa.

Objetivo: Crear un “plano del sistema” que satisfaga los requerimientos.

Tiene un gran impacto en el testing y mantenimiento.

Niveles en el proceso de diseño:

1. Diseño arquitectónico (Capítulo pasado): Identifica las componentes necesarias del sistema, su comportamiento y relaciones.
2. Diseño de alto nivel: Es la vista de módulos del sistema, es decir, cuales son los módulos del sistema, que deben hacer y cómo se organizan/interconectan
3. Diseño detallado o diseño lógico: Establece cómo se implementan las componentes/módulos de manera que satisfagan sus especificaciones. Incluye detalles del procesamiento lógico (i.e algoritmos) y de las estructuras de datos.

Principales criterios para evaluar:

- Corrección:
    - Es fundamental
    - ¿El diseño implementa los requerimientos?
    - ¿Es factible el diseño dada las restricciones?

- Eficiencia:
    - Le compete el uso apropiado de los recursos del sistema (cpu y memoria)
    - Debido al abaratamiento del hardware toma un segundo plano.

- Simplicidad:
    - Tiene impacto directo en mantenimiento
    - El mantenimiento es caro
    - Un diseño simple facilita la compresión del sistema => Hace el software mantenible
    - Facilita el testing
    - Facilita el descubrimiento y corrección de bugs
    - Facilita la modificación del código

Eficiencia y simplicidad no son independientes => el diseñador tiene que encontrar un balance.

Principios de diseño:

- **Partición y jerarquía:** Parte del principio “Divide and conquer”, Consta en dividir el problema en pequeñas partes que sean manejables, cada parte debe poder solucionarse y modificarse separadamente. Esto implica tratar de tener la menor comunicación entre módulos, ya que a medida que se agregan módulos, se agrega complejidad.

- **Abstracción:** La abstracción de una componente describe el comportamiento externo sin dar detalles internos de cómo se produce dicho comportamiento. Representa a los componentes como cajas negras, es útil para comprender sistemas existentes => tiene un rol importante en mantenimiento. Permite considerar una componente sin preocuparse por las otras, por lo que es necesaria para solucionar los problemas separadamente.

  Hay 2 mecanismos comunes de abstracción:
    - Abstracción funcional:
        - Especifica el comportamiento funcional de un módulo, estos se tratan como funciones de entrada/salida. Estos pueden especificarse con una pre y una postcondición.
    - Abstracción de datos:
        - Se esperan ciertas operaciones de un objeto de dato
        - Los detalles internos no son relevantes
        - Los datos se tratan como objetos junto a sus operaciones
        - Las operaciones definidas para un objeto solo pueden realizarse sobre este objeto
        - Desde fuera, los detalles internos del objetivo permanecen ocultos y solo sus operaciones son visibles.

- **Modularidad:**

    - Un sistema se dice modular si consiste de componentes discretas (separadas unas de otras) tal que puedan implementarse separadamente y un cambio a una de ellas tenga mínimo impacto sobre las otras
    - Provee la abstracción del software
    - Es el soporte de la estructura jerárquica de los programas
    - Mejora la claridad del diseño y facilita la implementación.
    - Reduce los costos de testing, debugging y mantenimiento.
    - No se obtiene simplemente recortando el programa en módulos, se necesitan criterios de descomposición.

Estrategias top-down y bottom-up: Un sistema es una jerarquía de componentes, hay 2 enfoques para diseñar tal jerarquía.

- Top Down: Comienza en la componente de más alto nivel (más abstracta); Prosigue construyendo las componentes de niveles más bajos descendiendo en la jerarquía.

    1. El diseño comienza con la especificación del sistema
    2. Define el módulo que implementa la especificación
    3. Especifica los módulos subordinados.
    4. Iterativamente, trata cada uno de estos módulos especificados como un nuevo problema
    5. En cada paso existe una clara imagen del diseño.
    6. La factibilidad es desconocida hasta el final.

- Bottom-up: Comienza por las componentes de más bajo nivel en la jerarquía (las más simples); Prosigue hasta la componente más alta.

Top down o bottom-up puros no son prácticos, en general se utiliza una combinación de ambos.

Módulo: Es una parte lógicamente separable de un programa, es una unidad discreta e identificable respecto a la compilación y la carga (Macro, procedimiento, función, package).

Criterios utilizados para seleccionar módulos que soportes abstracciones bien definidas y solucionables/modificables separadamente:

- Acoplamiento: Los módulos deben estar tan débilmente acoplados como sea posible, cuando sea posible => Módulos independientes.
  El acoplamiento depende del tipo de flujo de información, Hay 2 tipos de información: Control o dato.

- Transferencia de información de control:
    - Las acciones de los módulos dependen de la información.
    - Hace que los módulos sean más difíciles de comprender

- Transferencia de información de datos: Los módulos se pueden ver simplemente como funciones de entrada/salida.

Bajo Acoplamiento: Las interfaces solo contienen comunicación de datos
Alto acoplamiento: Las interfaces contienen comunicacion de informacion hibrida (datos+control).

- Cohesión: Caracteriza el vínculo intra-modular. Con la cohesión intentamos capturar cuan cercanamente están relacionados los elementos de un módulo entre sí.
  Da una idea de si los distintos elementos de un módulo tienen características comunes.

  Hay varios niveles de cohesión:
    - Casual: Relación entre los elementos del módulo no tiene significado.

    - Lógica: Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.

    - Temporal: Parecido a cohesión lógica pero los elementos están relacionados en el tiempo y se ejecutan juntos

    - Procedural: Contiene elementos que pertenecen a una misma unidad procedural.

    - Comunicacional: Tiene elementos relacionados por una referencia al mismo dato.

    - Secuencial: Los elementos están juntos porque la salida de un elemento corresponde a la entrada del otro.

    - Funcional: Es la más fuerte de las cohesiones: Todos los elementos del módulo están relacionados para llevar a cabo una sola función.

  BUSCAMOS: Menor acoplamiento y menor cohesión.

  ¿Cómo determinamos la cohesión de un módulo?
  Se realiza este test.

    - Si la oración es compuesta, tiene comas o más de un verbo => el está probablemente realizando más de una función.
      Probablemente tenga cohesión secuencial o comunicacional.

- Si la oración contiene palabras relacionadas al tiempo (ej.: primero, luego, cuando, después) => probablemente el módulo tenga cohesión secuencial o temporal.
- Si el predicado no contiene un único objeto específico a continuación del verbo (como es el caso de “editar los datos”) => probablemente tenga cohesión lógica.
- Palabras como inicializar/limpiar/... implican cohesión temporal.

Los módulos funcionalmente cohesivos siempre pueden describirse con oraciones simples.

Diagrama de estructura:
- Presenta una notación gráfica para tal estructura
- Representa módulos y sus interconexiones
- La invocación de A a B se representa con una flecha.
- Cada flecha se etiqueta con los ítems que se pasan.

Metodología de diseño estructurado (SDM):
- La estructura se decide durante el diseño.
- La implementación no debe cambiar la estructura.
- La estructura tiene efectos sobre el mantenimiento.
- La metodología de diseño estructurado apunta a controlar la estructura
- El objetivo de las metodologías de diseño es proveer pautas para auxiliar al diseñador en el proceso de diseño.
- SDM ve al software como una función de transformación que convierte una entrada dada en la salida esperada
- El foco en el diseño estructurado es la funcion de transformacion => SDM es una metodología orientada a funciones
- Utiliza abstracción funcional y descomposición funcional.
- Objetivo de la SDM:
    - Especificar módulos de funciones y sus conexiones siguiendo una estructura jerárquica con bajo acoplamiento y alta cohesión.
- Los módulos con módulos subordinados no realizan mucha computación.
- La mayoría de la computación se realiza en los módulos subordinados
- El módulo principal se encarga de la coordinación.
- Así sucesivamente hasta los modelos atómicos.
- La factorización es el proceso de descomponer un módulo de manera que el grueso de la computación se realice en los módulos subordinados.

Pasos principales de esta metodología:
1. Reformular el problema como un DFD:
    a. El diseño estructurado comienza con un DFD que capture el flujo de datos del sistema propuesto.
    b. DFD es una representación importante: provee una visión de alto nivel del sistema
    c. Enfatiza el flujo de datos a través del sistema

d. Ignora aspectos procedurales
e. La idea es identificar las entradas, salidas, fuentes, sumideros del sistema
f. Trabajar consistentemente desde la entrada hacia la salida o al revés, si se complica, cambiar el sentido.
g. No mostrar nunca lógica de control; si se comienza a pensar en término de loops/condiciones: parar y recomenzar.
h. Etiquetar cada flecha y burbuja. Identificar cuidadosamente las entradas y salidas de cada transformador.
i. Hacer uso de +(decisión) y *(Unión de dos entradas)
j. Ignorar las funciones menores al principio
k. En sistemas complejos realizar DFD de manera jerárquica
l. Dibujar DFD`s alternativos antes de definirse uno por uno.

2. Identificar las entradas y salidas más abstractas.
    a. Generalmente los sistemas realizan una función básica.
    b. Pero usualmente no se realiza sobre la entrada directamente
    c. Primero la entrada debe convertirse en un formato adecuado.
    d. Similarmente las salidas producidas por los transformadores principales deben transformarse a salidas físicas adecuadas
    e. Se requieren varios transformadores para procesar las entradas y las salidas.
    f. Objetivo de este paso: separar tales transformadores de los que realizan las transformaciones reales.
    Entradas más abstractas (MAI): Elementos de datos en el DFD que están más distantes de la entrada real, pero que aún puede considerarse como entrada.
    En general son items de datos obtenidos luego del chequeo de errores, formateo, validación de datos, conversión, etc.
    Para encontrarla:
        1. Ir desde la entrada física en dirección de la salida hasta que los datos no puedan considerarse entrantes.
        2. Ir lo más lejos posible sin perder la naturaleza entrante.
    Es dual para obtener las salidas más abstractas (MAO)

3. Realizar el primer nivel de factorización.
4. Factorizar los módulos de entrada, salida y transformadores.
5. Mejorar la estructura (heurísticas, análisis de transacciones)

Parte 2

Codificación:

Trata de reducir el tiempo de testing y debugging, El código debe ser fácil de leer y comprender, lo cual lo hace más difícil de escribir.

Programación Estructurada:
Los constructores de la **programación estructurada** son de única entrada y única salida.

La programación estructurada simplifica el flujo de control, facilitando en consecuencia tanto la comprensión de los programas así como el razonamiento formal o informal sobre estos.

Cualquier programa tiene dos estructuras:

1. **Estructura estática**: Es el orden de las sentencias en el código (lineal)
2. **Estructura dinámica**: es el orden en el cual se ejecutan las sentencias.

Cada estructura define un orden de las sentencias,
Para mostrar que un programa es **correcto,** debemos mostrar que el comportamiento dinámico es el esperado, pero debemos razonar sobre el código del programa, i.e la estructura estática.

Ocultamiento de información:
Las soluciones de software siempre contienen estructuras de datos que guardan información, en general, solo ciertas operaciones se realizan sobre la información.
En consecuencia la información debería ocultarse de manera que solo quede expuesta a esas pocas operaciones. Esta práctica reduce el acoplamiento.

Practicas de programacion:

- Constructores de control: Utilizar algunos pocos constructores estructurados
- Gotos: No usar
- Ocultamiento de la información: Usar
- Tipos definidos por el usuario: Usar para facilitar lectura
- Tamaño de los módulos: Hacerlos cortitos
- Interfaz del módulo: Hacerlo simple
- Robustez: Manipular situaciones excepcionales
- Efectos secundarios: Evitarlos , documentar
- Bloque catch vacío
- If o while vacio: Mala práctica
- switch case: Usar default
- Valores de retorno en lecturas: leer para lograr robustez
- “return” en “finally”: no usar
- Fuentes de datos confiables: Desconfiar

- Dar importancia a las excepciones: Los casos excepcionales son los que tienden a hacer que el programa funcione mal.

Estandares de codificacion:

Proceso de codificación incremental:
- Escribir código del módulo.
- Realizar test de unidad.
- Si error: arreglar bugs y repetir tests.

Proceso de codificación incremental (largo):
- Escribir el codigo del modulo para implementar un poco de funcionalidad
- Escribir tests
- Correr el test
- Hay errores?: Arreglar y correr el test
- No hay errores?: Fueron cubiertas todas las especificaciones?
- Si: Fin
- No?: Volver al punto 1.

Refactorización:

Es una técnica para mejorar el diseño del código existente, se realiza durante la codificación pero su propósito no es agregar nuevas características sino mejorar el diseño.
El objetivo tampoco es corregir bugs, se aplica al código que ya está funcionando.
Es la tarea que permite realizar cambios en el programa con el fin de simplificarlo y mejorar su comprensión, sin cambiar el comportamiento observacional de este.

Los fines de esto son:
- Reducir acoplamiento
- Incrementar cohesión
- mejorar respuesta al principio abierto cerrado

El principal riesgo es romper la funcionalidad existente, Para disminuir esta posibilidad:
- Refactorizar en pequeños pasos
- Disponer de scripts para tests automatizados para testear la funcionalidad existente.

Malos olores:
- Código Duplicado.
- Métodos largos.
- Clase grande.
- Lista larga de parámetros.
- Usar switch
- Generalidad especulativa: La subclase es la misma que la superclase, no hay razón aparente para esta jerarquía
- Demasiada comunicación entre objetos: No hay cohesión.

- Encadenamiento de mensajes: Un método llama a otro que llama a otro, Acoplamiento.

Refactorizaciones más comunes:
Para mejorar el diseño se enfocan en:
- Métodos
- Clases
- Jerarquía de clases

Mejoras de métodos:
- **Extracción de métodos:**
    - Se realiza si el método es demasiado largo, **Objetivo:** separar en métodos cortos cuya signatura indique lo que el método hace.
    - Partes de código se extraen como nuevos métodos
    - Variables referenciadas en esta parte se transforman en parámetros
    - Variables declaradas en esta parte pero utilizadas en otras partes deben definirse en el método original.
    - También se realiza si un método retorna un valor y también cambia el estado del objeto

- **Agregar/eliminar parámetros.**
- **Desplazamiento de métodos:**
    - Mover un método de una clase a otra
    - Se realiza cuando el método actúa demasiado con los objetos de la otra clase
    - Inicialmente puede ser conveniente dejar un metodo en la clase inicial que delegue al nuevo

- **Desplazamiento de atributos:**
    - Si un atributo se usa más en otra clase, moverlo a esa.

- **Extracción de clases:**
    - Si una clase agrupa múltiples conceptos, separa cada concepto en una clase distinta

- **Reemplazar valores de datos por objetos:**
    - Algunas veces, una colección de atributos se transforma en una entidad lógica
    - Separarlos como una clase y definir objetos para accederlos

Mejoras de jerarquías:
- **Reemplazar condicionales con polimorfismos:**
    - Si el comportamiento depende de algún indicador de tipo, no se está explotando el poder de la OO.
    - Realizar tan analisis de casos a través de una jerarquía de clases apropiada

- **Subir métodos / atributos:**
    - Los elementos comunes deben pertenecer a la superclase
    - Si la funcionalidad o atributo está duplicado en las subclases, pueden subirse a la superclase

Verificacion: El codigo necesita verificarse antes de que sea utilizado por otros

Distintas técnicas:
- Inspección de código
- Test de unidad
- Verificación de programa

Inspección de código:
Es un proceso de revisión como cualquier otro, el equipo de revisión se enfoca en encontrar defectos y bugs en el código, esta se hace una vez que el código fue compilado, testeado algunas veces y chequeado con herramientas de análisis estático.

Ítems de la lista de control:
- ¿Todos los punteros apuntan a algún lado?
- ¿Se inicializan todas las variables y punteros?
- ¿Los índices de los arreglos están dentro de sus cotas?
- ¿Terminan todos los loops?
- ¿Hay defectos de seguridad?
- ¿Se verificaron los datos de entrada?
- ¿Se satisfacen los estándares de codificación?

Testing de unidad:
Se enfoca en el módulo escrito por un programador, usualmente lo realiza el mismo que lo escribió. El TU requiere la escritura de drivers que ejecuten el módulo con los casos de test.

Análisis Estático:
Son herramientas para analizar los programas fuentes y verificar la existencia de problemas. Los analizadores estáticos no pueden encontrar todos los bugs en ocasiones dan “falsos positivos”.

Métodos formales:
Estos enfoques apuntan a demostrar la corrección de los programas.
Es decir, a demostrar que el programa implementa la especificación dada.

Proceso de desarrollo

Un **modelo de proceso** especifica un proceso general, usualmente como un conjunto de etapas, Este modelo es adecuado para una clase de proyectos.
Es decir, un modelo de proceso provee una estructura genérica de los procesos que puede seguirse en algunos proyectos con el fin de alcanzar los objetivos.

Si se elige un modelo para un proyecto, usualmente será necesario adecuarlo al proyecto, esta adecuación produce la especificación del proceso del proyecto, indicando cuál será el proyecto a seguir.

Es decir:
- Modelo del proceso: Especificación genérica del proceso
- Especificación del proceso: Plan de lo que debe ejecutarse
- Proceso: lo que realmente se ejecuta

Modelos comunes:
- Cascada: el modelo mas viejo - ampliamente usado
- Prototipado
- Iterativo : Ampliamente utilizado en la actualidad
- Timeboxing

Cascada: Secuencia inicial de las distintas fases
1. Análisis de requerimientos
2. Diseño de alto nivel
3. Diseño detallado
4. Codificación
5. Testing
6. Instalación

Una fase comienza solo cuando la anterior finaliza, Las fases dividen al proyecto; cada una de ellas se encarga de distintas incumbencias

Productos de trabajos usuales en este modelo
- Documento de requisitos / SRS
- Plan de proyecto
- Documentos de diseño (arquitectura, sistema, diseño detallado)
- Plan de test y reportes de test
- Código final
- Manuales del software (usuario, instalación, etc)
- Reportes de revisión, reportes de estado

Ventajas:
- Conceptualmente simple: Divide el problema en distintas fases que pueden realizarse de manera independiente
- Enfoque natural a la solución del problema
- Fácil de administrar en un contexto contractual: Existen fronteras bien definidas entre cada fase

Desventajas:
Una vez que terminamos una fase no podemos volver a ella, eso se soluciona con el modelo de cascada con feedback

Este modelo es ampliamente utilizado, Es muy adecuado para proyectos donde los requerimientos son bien comprendidos y las decisiones sobre tecnología son tempranas.

Prototipado:
El prototipado aborda las limitaciones del modelo de cascada en la especificación de los requerimientos.
El prototipo tiene que ser simple, no perder tiempo en detalles pequeños, el usuario “juega” con nuestro prototipo y nos da feedback.
La idea es reducir el testing, y calcular el costo económico del proyecto

Ventajas:
- Mayor estabilidad en los requerimientos
- Los requerimientos se congelan más tarde

Desventajas:
- Costo y tiempo

Desarrollo iterativo:

- Aborda el problema de “todo o nada” de cascada
- Combina beneficios del prototipado y del cascada
- Desarrolla y entrega el SW incrementalmente
- Cada incremento es completo en sí mismo
- Provee un marco para facilitar el testing
- Puede verse como una “secuencia de cascadas”
- El feedback de una iteración puede usarse en iteraciones futuras

Primer paso:
- Implementación simple para un subconjunto del problema completo
- Crear **lista de control del proyecto (LCP)** que contiene las tareas que se deben realizar para lograr la implementación final
- Cada paso consiste en eliminar la siguiente tarea de la lista haciendo diseño implementacion del sistema parcial y actualizar la LCP
- El proceso se repite hasta vaciar la lista
- LCP: guía los pasos de iteración y lleva las tareas a realizar
- Cada entrada en LCP es una tarea a realizarse en un paso de iteración y debe ser lo suficientemente simple como para comprenderla completamente

Aplicación:
Muy efectivo en el desarrollo de productos:
- Los desarrolladores mismos proveen la especificación
- Los usuarios proveen el feedback en cada release

- Basado en esto y experiencia previa => Nueva versión

Se suele aplicar cuando el tiempo de respuesta es importante, cuando no se puede tomar el riesgo de proyectos largos o cuando no se conocen todos los requerimientos

Beneficios:

- Pagos y entregas incrementales
- Feedback para mejorar lo desarrollado

Inconvenientes:

- La arquitectura y el diseño pueden no ser óptimos
- La revisión del trabajo hecho puede incrementarse
- El costo total puede ser mayor

Nuevo(rst) enfoques (que no pienso describir pero si listar por las dudas):

- Extreme programming
- Test driven development
- Desarrollo ágil
- Proceso unificado

SCRUM:

Hace entregas parciales y regulares del producto final

Usado donde la necesidad de tener resultados pronto, Hay requisitos cambiantes o poco definidos o donde hay alta rotación del personal

Ejecutado: Bloques temporalmente cortos y fijos (un mes a una semana). Cada iteración proporciona un resultado completo, incremento del producto final.

Actividades:

- Planificación de la iteración
- Ejecución de la iteración
- Inspección y adaptación

Planificación de la iteración (primer dia):

- Selección de los requisitos: El cliente presenta al equipo la lista de requisitos priorizada del producto. Se aclaran y se selecciona los requisitos prioritarios a completar en la iteración.

- Planificación: El equipo elabora la lista de tareas para desarrollar los requisitos. La estimación de esfuerzo se hace de manera conjunta y los miembros del equipo se auto asignan las tareas.

Ejecución de la iteración: Reunión de sincronización (diarias): Cada miembro del equipo inspecciona el trabajo que el resto esta realizando para poder hacer las adaptaciones necesarias y respondiendo a:

- ¿Qué he hecho desde la última reunión de sincronización?
- ¿Qué voy a hacer a partir de este momento?

- ¿Qué impedimentos tengo o voy a tener?

El scrum master debe encargarse de que el equipo pueda cumplir con su compromiso de no ser improductivo. Elimina los obstáculos que el equipo no puede resolver por sí mismo. Protege de interrupciones externas que puedan afectar la productividad.

El cliente y el equipo deben refinar la lista de requisitos preparándose para las siguientes iteraciones. Si es necesario, cambiar o replanificar los objetivos del proyecto para maximizar la utilidad de lo que se desarrolla y el retorno de inversión

Inspección y adaptación (ultimo dia):

Demostración: El equipo presenta al cliente los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo.
En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, el cliente realiza las adaptaciones necesarias de manera objetiva, replanificando el proyecto.

Retrospectiva: El equipo analiza cómo ha sido su manera de trabajar y cuales son los problemas que podrían impedirle progresar adecuadamente. El scrum master se encargará de ir eliminando los obstáculos identificados.

Proceso de software

El proceso es distinto del producto: el producto es el resultado de ejecutar un proceso.

IS se enfoca en el proceso

PREMISA: Un proceso adecuado ayuda a lograr los objetivos del proyecto con alta C&P

Procesos y modelos de proceso:

Un **Proceso exitoso** es el que satisface las expectativas en costo tiempo y calidad,

Un **Modelo de proceso** especifica un proceso general, usualmente con fases en las que el proceso debe dividirse, conjuntamente con otras restricciones y condiciones para la ejecución de dichas fases

Un modelo de proceso no se traduce directamente al proceso: en general el proceso real es una adaptación del modelo del proceso.

Componentes del proceso de software:

Dos procesos fundamentales:

- Desarrollo: Se enfoca en las actividades para el desarrollo y para garantizar la calidad necesarias para la ingeniería del sw.
- Administración del proyecto: Se enfoca en el planeamiento y control del proceso de desarrollo con el fin de cumplir los objetivos.

Especificación del proceso:

- El proceso generalmente es un conjunto de fases
- Cada fase realiza una tarea bien definida y produce una salida
- Tal salida intermedia se llama producto de trabajo
- Cada producto de trabajo es una entidad formal y tangible capaz de ser verificada
- Cada fase puede ser llevada a cabo usando distintas metodologías

Enfoque “ETVX”:

Cada fase sigue el enfoque ETVX (Entry-Task-Verification-Exit)

- Criterio de entrada: que condiciones deben cumplirse para iniciar la fase
- Tarea: Lo que debe realizar esa fase
- Verificación: Las inspecciones/controles/revisiones/verificaciones que deben realizarse a la salida de la fase
- Criterio de salida: Cuando puede considerarse que la fase fue realizada exitosamente.

Características deseadas:

- Proveer alta C&P
    - Debe producir un sw testable
    - Debe producir un sw mantenible
    - Debe eliminar defectos en etapas tempranas
    - Debe ser predecible y repetible
    - Debe soportar cambios y producir sw que se adapte a cambios

El costo de eliminar un defecto se incrementa a medida que perdura en el proceso de desarrollo

Para lograr alta C & P los errores deberían ser encontrados en la etapa en que se introdujeron, Para eso está el control de calidad.

Los procesos deben conseguir repetir el desempeño cuando se utilizan en distintos proyectos, es decir, el resultado de utilizar un proceso debería poder predecirse, No solo para estimar costos y esfuerzos, sino para estimar la calidad.

El proceso de desarrollo de software **es un conjunto de fases**, cada fase a su vez es una secuencia de pasos que definen la metodología de la fase.

¿Por qué utilizar fases?

- Dividir y conquistar
- Cada fase ataca distintas partes del problema
- Ayuda a validar continuamente el proyecto

Usualmente está compuesto por las siguientes actividades:

- Análisis de requerimientos y especificación
- Arquitectura y diseño
- Codificación
- Testing
- Entrega e instalación

Procesos extra que rondan a los principales:

- Administración del proyecto
- Inspección
- Administración de configuración
- Administración de cambios
- Administración de proceso

Administración de proyecto:

El proceso de desarrollo divide el desarrollo en fases y actividades.
Para ejecutarlas eficientemente, se deben asignar recursos, administrarlos, observar el progreso, tomar acciones correctivas, etc.

Tiene **3** fases:

- Planeamiento
- Seguimiento y control
- Análisis de terminación

Tareas claves:

- Estimación de costos y tiempos
- Seleccionar el personal
- Planear el seguimiento
- Planear el control de calidad

Seguimiento y control: Acompaña el proceso de desarrollo:

Tareas:

- Seguir y observar parámetros claves como costo, tiempos, riesgo, así como los factores que los afectan.
- Tomar acción correctiva si es necesario

Proceso de inspección:

Mejora la calidad y la productividad.

Los defectos pueden introducirse en el SW en cualquier etapa => debe eliminarse en cada etapa

- Este proceso es realizado por personal técnico para personal técnico.
- Es un proceso estructurado con roles definidos para cada participante
- Foco en encontrar problemas, no en resolverlos

Roles y responsabilidades:
- Moderador: Tiene la responsabilidad general
- Autor: Quien realizó el producto de trabajo
- Revisor: Quien identifica los defectos (generalmente es más de uno)
- Lector: lee línea a línea el producto de trabajo para enfocar el progreso de la reunión.
- Escriba: Registra las observaciones indicadas

Pasos:
- Identificar al moderador
- Seleccionar el equipo de revisión
- Preparar el paquete para la distribución
    - El producto de trabajo a revisar
    - Las especificaciones del producto de trabajo
    - Lista de control con ítems relevantes
    - estándares

Preparación y repaso previo (overview):
Breve reunión (opcional):
- Se entrega el paquete
- Se explica el propósito de la revisión
- Se da una breve intro señalando áreas de cuidado

Reunion de revision grupal:
- Propósito: definir la lista final de defectos
- **Criterio de entrada:** cada miembro debe haber hecho apropiadamente la revisión individual
- La reunión:
    - El lector lee línea a línea el producto de trabajo
    - En cualquier linea, cualquier observación que hubiere es efectuada
    - Se sigue una discusión para identificar el defecto
    - La decisión es registrada por el escriba

Al final de la reunión el escriba presenta la lista de defectos, SI hay pocos el producto de trabajo se acepta, sino se puede requerir otra revisión

El moderador esta a cargo de la reunión y juega el rol central de:
- Asegurar que el foro permanece sobre la identificación de defectos
- Lo que se está revisando es el producto de trabajo y no el autor de este
- Debe garantizar que la reunión se ejecute ordenada y amigablemente

Pautas para la revisión de los productos de trabajo:

Proceso de administración de configuración:

Un proyecto de software produce muchos items: Programas, documentos, datos, manuales, etc.

- Cualquiera de ellos puede cambiar fácilmente, es necesario saber el progreso del estado de cada ítem.
- Administración de configuración del software (SCM): Controla sistemáticamente los cambios producidos

SCM es usualmente independiente del proceso de desarrollo: Los procesos de desarrollo miran el gran esquema, pero no los cambios individuales de ítems/archivos.
A medida que los ítems se producen, se introducen en la SCM

Mecanismos principales:

- Control de acceso
- Control de versiones
- Identificador de la configuración
- Otros mecanismos incluyen: convenciones de nombres, estructuras de directorios, etc

Testing

Principal objetivo de un proyecto: desarrollar un producto de alta calidad con alta productividad.

Habla de las posibilidades que el software falle, mas defectos => más chances de que falle => menor confiabilidad.

Objetivo de la calidad: que el producto entregado tenga la menor cantidad de defectos como sea posible

Desperfecto: Ocurre si el comportamiento del software es distinto al esperado

Defecto/bug: Lo que causa el desperfecto.

- Un desperfecto implica la presencia del defecto
- La existencia del defecto no implica la ocurrencia del desperfecto
- Pero: un defecto tiene el potencial para causar el desperfecto

Rol del testing:

- Las revisiones son procesos humanos: no pueden encontrar todos los defectos
- Estos defectos tienen que identificarse por medio del testing

Durante el testing, un programa se ejecuta siguiendo un conjunto de casos del test, Si hay desperfectos en la ejecución de un test, entonces hay defectos en el software.
Si no ocurren desperfectos, entonces la confianza en el software crece.
Para **detectar** los defectos debemos causar desperfectos durante el testing, mientras que para **identificar** el defecto real (que causa el desperfecto) debemos recurrir al debugging

Oráculos del test:
Para verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos conocer el comportamiento correcto para este caso.
Muchas veces el oráculo es humano y por lo tanto propenso a cometer errores. El oráculo humano usa la especificación para decidir el comportamiento correcto, pero las mismas especificaciones pueden contener errores.
En algunos casos, los oráculos pueden generarse automáticamente de la especificación.

Si existen defectos, deseamos que los casos de test los evidencian a través de fallas.

Idealmente deseamos construir un conjunto de casos de test tal que la ejecución satisfactoria de todos ellos implique la ausencia de defectos. Además, como el testing es costoso, deseamos que sea un conjunto reducido

Estos dos deseos son contradictorios, por eso usamos algún **criterio de selección de tests.**
El criterio de selección específica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o a la especificación

Hay dos propiedades fundamentales que esperamos de los criterios de test:

Confiabilidad: Un criterio es confiable si todos los conjuntos de casos de test que satisfacen el criterio detectan los mismos errores

Validez: Un criterio es válido si para cualquier error en el programa hay un conjunto de casos de test que satisfagan tal criterio y detecten el error.

Es prácticamente imposible obtener un criterio que sea confiable y válido al mismo tiempo y que también sea satisfecho por una cantidad manejable de casos de test.

Dos enfoques para diseñar casos de test:

- **Caja negra o funcional**
- **Caja blanca o estructural**

Ambos son complementarios

Caja negra:
El software a testear se trata como una caja negra, no se que pasa ahí dentro.

Para diseñar los casos de test, se utiliza el comportamiento esperado del sistema.
No se utiliza la estructura interna del código.

Premisa: El comportamiento esperado está especificado

Para el testing de módulos: la especificación producida en el diseño define el comportamiento esperado

Para el testing de sistema: la SRS define el comportamiento esperado

El testing funcional más minucioso es el exhaustivo: El software está diseñado para trabajar sobre un espacio de entrada => Testear el software con todos los elementos del espacio de entrada.

No es viable: demasiado costoso (si no imposible)

Particionado por clase de equivalencia:
Dividir el espacio de entrada en clases de equivalencias

Parte de esta idea: si el software funciona para un caso de test en una clase => Muy probablemente funcione de la misma manera para todos los elementos de la misma clase.

Base lógica: La especificación requiere el mismo comportamiento en todos los elementos de una misma clase => es muy probable que el software se construya de manera tal que falle para todos o para ninguno.

Análisis de valores límites:
Los programas fallan generalmente sobre valores especiales, Estos valores usualmente se encuentran en los límites de las clases de equivalencia.
Los casos de test que tienen valores límites tienen alto rendimiento, también se denominan casos extremos.
Un caso de test de valores límites es un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida

Grafo de causa-efecto:
Los análisis de clase de equivalencia y valores límites consideran cada entrada separadamente, Para manipular las entradas distintas combinaciones de las clases de equivalencia deben ser ejecutadas.
La cantidad de combinaciones puede ser grande: Si hay n condiciones distintas en la entrada que puedan hacerse válidas o inválidas => 2^n clases de equivalencia.
El **Grafo de causa-efecto** ayuda a seleccionar las combinaciones como condiciones de entrada.

Lo que hace es identificar las causas y efectos en el sistema:

- Causa: Distintas condiciones en la entrada que pueden ser verdaderas o falsas.
- Efecto: Distintas condiciones de salidas (V/F también)

Identificar cuáles causas pueden producir qué efectos; las causas se pueden combinar

- Causas y efectos son nodos en el grafo.
- Las aristas determinan dependencia: hay aristas positivas y negativas
- Existen nodos and y or para combinar las causas

A partir del grafo causa-efecto se puede armar una tabla de decisión. Lista las combinaciones de condiciones que hacen efectivo cada efecto.

Testing de a pares:
Usualmente muchos parámetros determinan el comportamiento del sistema.
Los parámetros pueden ser entradas o seteos y pueden tomar distintos valores (o distintos rangos de valores).
Muchos defectos involucran sólo una condición.
Pero no todos los defectos son de modo simple: el software puede fallar en combinaciones.
Los defectos de modo múltiple se revelan con casos de test que contemplan las combinaciones apropiadas, Esto se denomina test combinatorio.

De todas formas el test combinatorio no es factible (muy grande), Se investigó que la mayoría de tales defectos se revelan con la interacción de pares de valores. Para modo doble necesitamos ejercitar cada par => se denomina testing de a pares.

Casos especiales: Los programas usualmente fallan en casos especiales, que dependen de la naturaleza de la entrada, tipos de estructuras de datos que manejan, etc.
No existen buenas reglas para identificarlos, pero una buena forma es adivinar, que se denomina "adivinanza del error".

Testing basado en estados: Algunos sistemas no tienen estados; para las mismas entradas se exhiben siempre las mismas salidas, pero en muchos sistemas el comportamiento depende del estado del sistema.
El testing basado en estado está dirigido a este tipo de sistemas.

Un sistema puede modelarse como una máquina de estados. El espacio de estados puede ser muy grande, pero puede particionarse en pocos estados, cada uno representando un estado lógico de interés del sistema.

Un **modelo de estados** tiene 4 componentes:

- Un conjunto de estados: Estados lógicos representando el impacto acumulativo del sistema
- Un conjunto de transiciones: representa el cambio de estado en respuesta a algún evento de entrada
- Un conjunto de eventos: son las entradas al sistema
- Un conjunto de acciones: Son las salidas producidas en respuesta a los eventos de acuerdo al estado actual

El modelo de estado muestra la ocurrencia de las transiciones y las acciones que se realizan, usualmente se construye a partir de las especificaciones o los requerimientos.

Los casos de test se seleccionan con el modelo de estado y se utilizan posteriormente para testear la implementación.

Existen varios **criterios para generar los casos de test:**

- Cobertura de transiciones: El conjunto T de casos de test debe asegurar que toda transición sea ejecutada
- Cobertura de par de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado
- Cobertura de árbol de transiciones: T debe ejecutar todos los caminos simples.

El test basado en estados se enfoca en el testing de estados y transiciones.

El modelo de estado se realiza usualmente luego de que la información de diseño se hace disponible, en este sentido se habla a veces de **testing de caja gris (**dado que no es una caja negra pura)

Testing de caja blanca:

El testing de caja negra se enfoca solo en la funcionalidad: lo que el programa hace, no lo que este implementa
El testing de caja blanca se enfoca en la implementación: El objetivo es ejecutar las distintas estructuras del programa con el fin de descubrir errores.

Los casos de test se derivan a partir del código.
Se denomina también **testing estructural.**

Existen varios criterios para hacer la selección de los casos de test.

Tipos de testing estructural:

- Criterio basado en el flujo de control: Observa la cobertura del grafo de flujo de control.
- Criterio basado en el flujo de datos: Observa la cobertura de la relación definición-uso en las variables
- Criterio basado en mutación: Observa a diversos mutantes del programa original

Criterio basado en flujo de control: Considerar el programa como un grafo de flujo de control.
Los **nodos** representan bloques de código (i.e conjuntos de sentencias que siempre se ejecutan juntas)

Una **arista (i,j)** representa una posible transferencia de control del nodo **i** al **j**
Suponemos la existencia de un nodo inicial y un nodo final.
Un **camino** es una secuencia del nodo inicial al nodo final.

Criterio de cobertura de sentencia:

Sentencia (dilema): una línea, desde donde empieza hasta el ;.

Cada sentencia se ejecuta al menos una vez durante el testing.

Limitación: puede no requerir que una decisión evalúa a falso en un if si no hay else.

El conjunto de casos de test {x = 0, 0} tiene el 100% de cobertura pero el error pasa desapercibido

Notación: {{x = 0, 0} **}** La primera parte de este par nos dice

1. El valor de las entradas del sistema
2. El valor que nos devuelve

No es posible garantizar 100% de cobertura debido a que puede haber nodos inalcanzables.

Criterio de cobertura en ramificaciones: cada arista debe ejecutarse al menos una vez en el testing.

La cobertura de ramificaciones implica cobertura de sentencias

Si hay múltiples condiciones en una decisiones luego no todas las condiciones se ejercitan como verdadera y falsa.

Criterio de cobertura de caminos:

- Todos los posibles caminos del estado inicial al final deben ser ejercitados.
- Cobertura de caminos implica cobertura de bifurcación
- **Problema:** Cantidad de caminos puede ser infinita (considerar loops)
- Notar además que puede haber caminos que no son realizables

Existen criterios intermedios (entre el de caminos y el de bifurcación)

Proveen alguna idea cuantitativa de la amplitud del test suite, Se utiliza más para evaluar el nivel de testing que para seleccionar los casos de test

Criterio basado en flujo de datos:
Se construye un grafo de definición-uso etiquetando apropiadamente el grafo de flujo de control.

Una sentencia en el grafo de flujo de control puede ser de tres tipos:

- Def: representa la definición de una variable
- uso-c: cuando la variable se usa para computo
- uso-p: cuando la variable se utiliza en un predicado para transferencia de control.

El grafo de def-uso se construye asociando variables a nodos y aristas del grafo de flujo de control:

- Por cada nodo i,def(i) es el conjunto de variables para el cual hay una definición en i.
- Por cada nodo i, c-use(i) es el conjunto de variables para el cual hay un uso-c
- Para cada arista (i,j),p-use(i,j) es el conjunto de variables para el cual hay uso-p.

Un camino de i a j se dice libre de definiciones con respecto a una var x si no hay definiciones de x en todos los nodos intermedios

Criterios:

- Todas las definiciones: por cada nodo i y cada x en def(i) hay un camino libre de definiciones con respecto a x hasta un uso-c o uso-p de x
- Todos los usos-p: Todos los usos-p de todas las definiciones deben testearse
- Otros criterios: Todos los usos-c, algunos usos-p, algunos usos-c

Soporte con herramientas:

Una vez elegido el criterio surgen dos problemas:
¿El test suite satisface el criterio?
¿Cómo generar el test suite que asegure cobertura?

Para determinar cobertura se pueden usar herramientas
Usualmente son de asistencia. El problema de generación de test que cubra un criterio es habitualmente indecidible
Las herramientas dicen que sentencias o ramificaciones quedan sin cubrir.

Comparación y uso:

Se deben utilizar tanto test funcionales (caja negra) como estructurales (caja blanca)

Ambas técnicas son complementarias:

- Caja blanca => bueno para detectar errores en la lógica el programa
- Caja negra => Bueno para detectar errores de entrada/salida

- Los métodos estructurales son útiles a bajo nivel solamente, donde el programa es manejable
- Los métodos funcionales son útiles a alto nivel, donde se busca analizar el comportamiento funcional del sistema o partes de este.

Testing incremental:

Los objetivos del testing son: detectar tantos defectos como sea posible y hacerlo a bajo costo.

- Objetivos contrapuestos: incrementar el testing permite encontrar más defectos pero a la vez incrementa el costo.
- Testing incremental: Agregar partes no testeadas incrementalmente a la parte ya testeada
- El testing incremental es esencial para conseguir los objetivos antedichos:
    - Ayuda a encontrar más defectos

    - Ayuda a la identificación y eliminación

Niveles de testing:

- El código contiene defectos de requerimiento, de diseño y de codificación
- La naturaleza de los defectos es diferente para cada etapa de inyección del defecto
- Un solo tipo de testing sería incapaz de detectar los distintos tipos de defectos, por lo tanto se utilizan distintos niveles de testing para revelar los distintos tipos de defectos.

Testing de unidad:
Los distintos módulos del programa se testean separadamente contra el diseño, que actúa como especificación del módulo. Se enfoca en los defectos inyectados durante la codificación => el objetivo es testear la lógica interna de los módulos.

Testing de integración:
Se enfoca en la interacción de módulos de un subsistema. Los módulos que ya fueron testeados unitariamente se combinan para formar subsistemas, los que son sujetos a testing de integración.

Testing de sistema:
El sistema de software completo es testeado.
Se enfoca en verificar si el software implementa los requerimientos. Realiza el ejercicio de validar el sistema con respeto a los requerimientos.
Generalmente es la etapa final del testing antes de que el software sea entregado.
Debería ser realizado por personal independiente, pero los defectos son eliminados por los desarrolladores.

Testing de aceptación:
Se enfoca en verificar que el software satisfaga las necesidades del usuario.
Generalmente se realiza por el usuario/cliente en el entorno del cliente y con datos reales.

Otras formas de testing:

Testing de desempeño: Requiere de herramientas para medir el desempeño

Testing de estrés (stress testing): El sistema se sobrecarga al máximo; requiere de herramientas de generación de carga

Testing de regresión:

- Se realiza cuando se introduce algún cambio al software
- Verifica que las funcionalidades previas continúan funcionando bien
- Se necesitan los registros previos para poder comparar => tests deben quedar apropiadamente documentados

El plan de test:

El testing usualmente comienza con la realización del plan de test y finaliza con el testing de aceptación.

El plan de test es un documento general que define el alcance y el enfoque del testing para el proyecto completo.
Entradas plan del proyecto, SRS, diseño

Usualmente contiene:

1) Especificación de la unidad de test: que unidad necesita testearse separadamente
2) Características a testear: esto incluye funcionalidad, desempeño, usabilidad, restricciones de diseño
3) Enfoque: Criterios a utilizarse, cuando detenerse, cómo evaluar, etc
4) Entregables

Especificación de los casos de test:
Después de que hicimos el plan del test, nos dedicamos a hacer la especificación.
Con cada caso de test tenemos que escribir, las entradas a utilizar, las condiciones que este testeara y el resultado esperado

La **efectividad y costo** del testing dependen del conjunto de casos de test seleccionado.

¿Cómo determinar si un conjunto de casos de test es bueno? i.e que detecte la mayor cantidad de defectos y que ningún conjunto más pequeño también lo encuentre

No existe una manera de determinar la bondad; usualmente el conjunto de casos de test es revisado por expertos

Planeamiento del proyecto de software

Objetivos del proyecto de sw: Construir un sistema de sw que cumpla con los costos, tiempos y calidad.

SIn embargo muchos proyectos fallan:
Un tercio se desbocan con costos o tiempos superiores al 125% de los estipulados.
Razones principales:

1. Objetivos poco claros
2. Mal planeamiento
3. Administración del proyecto sin metodología
4. Nueva tecnología
5. Personal insuficiente

Todas ellas están relacionadas a la administración del proyecto.

Proceso para la administración del proyecto:

- Planeamiento
- Seguimiento y control
- Análisis de terminación

Nos vamos a centrar en **planeamiento**
El planeamiento se realiza antes de comenzar con el desarrollo del proyecto, requiere como entrada los requerimientos y la arquitectura.
Durante el planeamiento se planean todas las tareas que la administración del proyecto necesita realizar. Durante el seguimiento y control, el plan es ejecutado y actualizado.

Tiene 7 puntos:

1. Planeamiento del proceso
2. Estimación del esfuerzo
3. Estimación de tiempos y recursos
4. Plan para la administración de la configuración
5. Planeamiento de la calidad
6. **Administración del riesgo**
7. Plan para el seguimiento del proyecto

Planeamiento del proceso:
La idea es planear cómo se ejecutará el proyecto, esto incluye:

- Determinar el modelo de proceso a seguir
- Adecuarlo a las necesidades del proyecto
- Definir las etapas
- Criterios de entrada y de salida en cada etapa
- Actividades de verificación a realizar en cada etapa
- Definir las metas parciales (milestones)

Estimación del esfuerzo:
Dado un conjunto de requerimientos es deseable/necesario saber cuánto costará en tiempo y dinero el desarrollo del sw.
El esfuerzo se mide usualmente en personas/mes.
No hay una forma fácil de estimarlo, la estimación mejora a medida que se incrementa la información sobre el proyecto. Las estimaciones más tempranas son más propensas a inexactitud que las avanzadas en el proyecto

Construcción de modelos: Un modelo intenta determinar la estimación del esfuerzo a partir de valores de ciertos parámetros, tales valores dependen del proyecto => Este modelo reduce el problema de estimar el esfuerzo del proyecto al de estimar ciertos **parámetros claves** del proyecto

Dos enfoques:

- **Top-down:** Determinar el esfuerzo total y luego calcular el esfuerzo de cada parte del proyecto.

- **bottom-up:**
    a. Identificar los módulos del sistema y clasificarlos como simples,medios o complejos
    b. Determinar el esfuerzo promedio de codificación para cada tipo de módulo
    c. Obtener el esfuerzo total de codificación en base a la clasificación anterior y al conteo de cada tipo
    d. Utilizar la distribución de esfuerzo s de proyectos similares para estimar el esfuerzo de cada tarea y finalmente el esfuerzo total
    e. Refinar los estimadores anteriores en base a factores específicos del proyecto.

Modelo COCOMO (COnstructive COst MOdel): estimación con un error dentro del 20% en el 68% de los casos.
Es un enfoque top-down que utiliza tamaño ajustado con algunos factores.
Procedimiento:

1. Obtener el estimador inicial usando el tamaño
2. Determinar un conjunto de 15 factores de multiplicación representando distintos atributos.
3. Ajustar el estimador de esfuerzo escalando según el factor de multiplicación final.
4. Calcular el estimador de esfuerzo de cada fase principal.

Para obtener el estimador inicial:
a*tamaño^b
Los sistemas se pueden clasificar en:

- Orgánico: Simple y desarrollados por pequeños equipos
- Semi-rigido: Mezcla entre orgánico y rígido
- Rígido: Sistema complicau

Planificación y recursos humanos:

Dos niveles de planificación

- Global: abarca metas parciales (milestones) y la fecha final
- Detallada: asignación de las tareas de más bajo nivel a los recursos

Planificación global: Para una estimación dada hay cierta flexibilidad, dependiendo de los recursos asignados.
Un método es estimar el tiempo programado del proyecto M (en meses) como una función del esfuerzo en personas-mes.
Seguidamente: determinar la duración de cada meta parcial principal del proyecto

Planificación detallada: Para alcanzar cada meta, muchas tareas deben llevarse a cabo.

- Tareas de bajo nivel: aquellas realizadas por una persona en no más de 2 o 3 días.
- Planificación: decidir las tareas y asignarlas preservando siempre la planificación de alto nivel
- Es un proceso iterativo: si no se pueden acomodar todas las tareas => se realiza la planificación global

- La planificación detallada no se realiza de manera completa al comienzo: esta evoluciona
- La planificación detallada es el documento más activo de la administración del proyecto.

Estructura del equipo de trabajo:
Para asignar las tareas en la planificación detallada es necesario un equipo de trabajo estructurado.

La organización jerárquica es la más común, Hay un administrador de proyecto con la responsabilidad global, Tiene programadores, testers y administradores de configuración para ejecutar las tareas detalladas.

Equipos democráticos: Liderazgo rotativo para pequeños grupos

Una nueva alternativa: Para el desarrollo de grandes productos, Tiene tres tareas principales: desarrollo, testing y administración del programa, Cada una tiene su equipo y cada equipo su líder.

Planeamiento de la administración de la configuración del software:
Se deben identificar los ítems de configuración y especificar los procedimientos a usar para controlar e implementar los cambios de estos ítems.
Se realiza cuando el proyecto ya ha sido iniciado y se conoce la SRS y el entorno de operación.

Algunas actividades: Min 30:20

Planeamiento del control de calidad:
Objetivo básico: entregar un sw de alta calidad
Unidad de medida de calidad estándar: densidad de defectos entregados

Defecto: Algo que causa que el sw se comporte de manera inconsistente

Objetivo del proyecto: entregar sw con baja densidad de defectos entregados

Introducción y eliminación de errores: El desarrollo de sw es una actividad altamente dependiente de personas => es propensa a errores

Como el objetivo de calidad es la baja densidad de defectos => los defectos deben eliminarse.

Esto se realiza fundamentalmente mediante las actividades de control de calidad incluyendo revisiones y testing

Enfoque **ad hoc:** tests y revisiones de acuerdo a cuándo y cómo se necesiten

Enfoque **de procedimiento:** El plan define qué tareas de QC se realizarán y cuando

Enfoque **cuantitativo:** Va más allá de requerir que se ejecute el procedimiento, analiza los datos recolectados de los defectos y establece juicios sobre la calidad

Plan de calidad: Establece que actividades deben realizarse.
El nivel del plan depende de los modelos de predicción disponibles.

Administración de riesgos:

Cualquier proyecto puede fallar debido a eventos no previstos, La administración de riesgo es un intento de minimizar las chances de fallas.

Riesgo: Cualquier condición o evento de ocurrencia incierta que puede causar la falla del proyecto.

Evaluación del riesgo:

Identificación del riesgo: Identificar los posibles riesgos del proyecto, es decir, aquellos eventos que podrían ocurrir y causar la falla del proyecto.
La forma de hacerlo es mediante listas de control, experiencias pasadas, brainstorming.etc.
10 factores de riesgo más importantes: Min 38

Análisis de riesgos y definición de prioridades: La cantidad de riesgos puede ser grande, por lo que se deben priorizar para enfocar la atención en las áreas de alto riesgo.
Para ello: establecer la probabilidad de materialización de los riesgos identificados y la pérdida que estos originarian

Ordenar de acuerdo al “valor de exposición de riesgo (RE)”

Control de riesgos:
Si es posible evitarlo => evitarlo
En los otros casos planear y ejecutar los pasos necesarios para mitigar los riesgos

Planificación del seguimiento del proyecto:

El plan de administración del proyecto es meramente un documento que sirve como guía.
Para asegurar que la ejecución se realiza como se planeó, este debe seguirse y controlarse
El seguimiento requiere de mediciones y métodos que las interpreten.
El plan de seguimiento incluye:

- Planificar que medir, cuando y como
- Como analizar y reportar estos datos

Principales medidas:

- Tiempo: principal medida
- Esfuerzo: principal recurso
- Defectos: determinan calidad
- Tamaño: mucha información se normaliza respecto al tamaño

Seguimiento a nivel de actividad:
Asegura que cada actividad se realiza apropiadamente y a tiempo.
• Realizado diariamente por los administradores de proyecto.
• Una tarea realizada se marca al 100%; las herramientas pueden determinar el estado de las tareas de más alto nivel.

Reporte de estado:
• Usualmente se prepara semanalmente.
• Contiene: resumen de actividades completadas y pendientes (desde el último reporte); cuestiones que necesitan atención o deben ser resueltas.

Espero nunca más en mi vida ver un video de laura hablando.
SANGRE, SUDOR Y LÁGRIMAS