Teoría de Ingenieria 2

[Teoría 1: Repaso, comunicación y SRS. 6](#_jdbw6cvf8ttf)

[Proceso de software 6](#_y6d16g3mir0p)

[Modelo 6](#_h9cnjfu1jlqr)

[Comunicación 6](#_17q6spkv70fd)

[Elicitación de requisitos 6](#_qibddi02kgs)

[Muestreo de la documentación, los formularios y los datos existentes 7](#_fdpw64xx2ybr)

[Observación del ambiente de trabajo 7](#_q7mlighmfiv1)

[Visitas al sitio 7](#_vw46625f0s3s)

[Cuestionarios 8](#_2sh11r2za028)

[Entrevista 8](#_ceawo6bm6vo2)

[JRP 8](#_3co8x0ds0trh)

[Brainstorming 9](#_yw4esifhwh1p)

[Requerimientos SRS 9](#_fg50m7xukq9e)

[Requerimientos 9](#_iv9gt8s81doa)

[ConOps (IEEE Std. 1362-1998) 9](#_71cnc4vbm8li)

[SRS (standard requirements specifications) IEEE Std. 830-1998 9](#_25ogh8597pz2)

[Sección 1 del SRS 11](#_fbk4yglst6fi)

[Sección 2 del SRS 11](#_srljucae8kq8)

[Sección 3 del SRS 11](#_tdivzd6gbu37)

[Sección 4 del SRS 12](#_8uelerw1n6vd)

[Sección 5 del SRS 12](#_fao4l5veq7ei)

[Comunicación 12](#_935t1kpwxcnb)

[Consejos de comunicación 12](#_hbdo07ao8pdk)

[Consejos de cierre 12](#_56y0d0cio5yp)

[Detalles estéticos: 12](#_c46y0bczd9ym)

[Elevator pitch 13](#_kd73qyaw60ov)

[Teoría 2: GCS, planificación organizativa. 13](#_ggq0vrqgnn73)

[Gestión de la configuración de software (GCS) 13](#_bjld1extk3f5)

[Cambios 13](#_4yvkswbwcyxx)

[GCS 13](#_lsp81sroxae2)

[Involucrados de la GCS 14](#_yav7ce6pot0v)

[Elementos de la GCS (ECS) 14](#_otv2fespfbhz)

[Organización de un repositorio de ECS 14](#_t46yl5fehplk)

[Origen del cambio 15](#_plxwd4iaxmu)

[Definición de Línea base 15](#_f4cb1sp570sc)

[Definición de la IEEE 15](#_qcoy08rus9im)

[En el contexto de la Ingeniería de Software 15](#_6l3lv2n03ccd)

[Importancia del GCS 16](#_dx8qd5x9bvzv)

[Proceso 16](#_mywz8hoh3pun)

[Identificación 17](#_r2pojh7f9jfy)

[Control de versiones 17](#_vvc4jmyfhg5a)

[Control de cambios 18](#_tf0zgdy5fzmm)

[Auditoría de la configuración 18](#_9mnr4sstu6vq)

[Generación de informes de estado de la configuración (auditoría) 19](#_7b1na7wk380l)

[Gestión de Proyectos 19](#_i0s4ws7h34sy)

[Las 4 P de la Gestión de Proyectos de Software 19](#_t08iv5e52mps)

[Resultados de la mala gestión 19](#_qlcj4ll6n2k4)

[Elementos clave de la gestión de proyectos 19](#_xtpe0jlvg6cr)

[Planificación 20](#_rc9gxsob78k)

[Planificación organizativa 20](#_45tqps1dfl62)

[Participantes 20](#_jamxliyr9ldx)

[Líderes de equipo 20](#_ja0wdcma5f8a)

[El equipo de software 21](#_k2m1upgtvcqx)

[Paradigmas Organizacionales del equipo 22](#_nitita3vf8xz)

[Paradigma cerrado 22](#_j1c0tru2s1pr)

[Paradigma abierto 22](#_6w098ads6a0l)

[Paradigma síncrono 22](#_2m3vjgfcfrn0)

[Paradigma aleatorio 23](#_fzku1bx7orva)

[El equipo de software 23](#_8t096adngc0p)

[Comunicación Grupal 24](#_adax3v4fai7j)

[Equipo ágil 24](#_2ykws6bij4qt)

[Teoría 3: Riesgos. 24](#_z90l1smpzws)

[Riesgo 24](#_yu8pno9okl41)

[Deuda técnica 25](#_dypr8b3rtwa3)

[Estrategias de gestión de riesgos: 25](#_h06d41vbrkhg)

[Categorización de riesgos 26](#_bg4prt65tjza)

[Proceso de gestión de riesgos 26](#_3bjdcqmx2c8p)

[Identificación de riesgos 26](#_ojq045hkhlyb)

[Análisis de riesgos 27](#_lesbiksewofc)

[Planeación de riesgos 29](#_oinieewuxd2i)

[Ejemplo 30](#_wwv4o55jsv9v)

[Supervisión 31](#_fclgmtqo18kn)

[Clase 4: Conceptos de diseño de software y planificación temporal 31](#_8uz5fxotfqni)

[Planificación temporal 31](#_1ttrz4qchpr2)

[Calendarización del proyecto 32](#_9b7642mh7iij)

[Tarea 32](#_e2wd9r79aft3)

[Red de tareas: 32](#_rvv9iklu57rx)

[PERT: Método de planificación temporal 33](#_rzbazrc4uj2l)

[CPM: Critical path method 33](#_gptk7yh3gddu)

[Método del camino crítico: CPM y PERT juntos. 33](#_aazs74wfna1l)

[Fechas tempranas y tardías 34](#_qohk1xhvsfdu)

[Encontrando el camino crítico 36](#_g546tfl4obyp)

[Conceptos de diseño de Software 36](#_xrrr96rdrzk)

[Tipos de diseño 37](#_12njd62850ub)

[Características a evaluar del diseño 37](#_2m2zo1hkm7iw)

[Criterios técnicos para un buen diseño 37](#_89z24syxe1ur)

[Evolución del diseño de software 37](#_ef9jj9vesmcq)

[Conceptos de diseño 38](#_bkw6xqfo5rvs)

[Abstracción 38](#_j85gmpmgrfmr)

[Arquitectura del software 38](#_z1nom41t5a72)

[Patrones 38](#_pjzcgxdppell)

[Modularidad 38](#_tc8uo4avopfb)

[Ocultamiento de información 38](#_3tl72lnx4dx9)

[Independencia funcional 38](#_ez13umhg27bc)

[Refinamiento 39](#_wupymwb9thy)

[Refactoring 39](#_jqcxln6bfe4n)

[Principios del modelado de diseño 39](#_xhkbfg8r2fn)

[Clase 5: Diseño de la interfaz de usuario 40](#_fdi8g29f376s)

[Formas de consumir 40](#_xutocbjqshfm)

[Proceso iterativo 40](#_w4d9es1c43y0)

[● Análisis y modelado 40](#_efilsnrswmm5)

[● Diseño de interfaz 40](#_3c7tknhpoksk)

[● Construcción de la interfaz 40](#_a8xs1p7qj3lm)

[● Validación 41](#_dwv57syc6qh8)

[UX 41](#_bb2sycbu5eov)

[Tipos 41](#_74ftmmgfc7he)

[Investigación de usuarios 41](#_qkd40ugizb86)

[Relevamiento de la tarea 42](#_rfgcs2ftz0xx)

[UI vs UX 42](#_gcdhfkv2g6od)

[Aspectos de diseño de interfaz 42](#_7osfndkcmdn1)

[Reglas doradas del diseño 42](#_c3itcggr3hxs)

[Usabilidad 43](#_4dqbcciadvtg)

[Principios de Nielsen 43](#_guy7csd7o02e)

[Tipos de interfaces 44](#_d676kvlbu02y)

[Presentación de info 44](#_uk9l512t2rz)

[Clase 6: Diseño arquitectónico 45](#_5p91h6s7tyzn)

[Diseño arquitectónico 45](#_f48vmflgbhjo)

[Arquitectura y req. no funcionales 45](#_bznmqyrctix6)

[● Protección 45](#_l6qliyqarsow)

[● Disponibilidad 45](#_hvffhc4p6iec)

[● Mantenibilidad 45](#_kfxxdj49t2va)

[Organización del sistema 45](#_rfvlsypgvcnm)

[➔ Subsistema 45](#_6326tfmwdvkq)

[➔ Módulo 46](#_rmzhwnrf2kp2)

[Patrón: Repositorio 46](#_sbagnbfq8hp4)

[Patrón: Cliente Servidor 46](#_s9jt9kq68qvp)

[Patrón: Arq. en capas 47](#_9y4xjjc1c8d1)

[Descomposición modular 47](#_f5nf3pfc6xek)

[Modelos de Control 48](#_ubl9m8nw8w95)

[● Control centralizado 48](#_8q8jv7f52nhu)

[● Basado en eventos 49](#_9lcc9s21kr1p)

[Arquitectura de los Sistemas Distribuidos 49](#_i3gf7r6zeeui)

[Características 49](#_8tpc89g3gjca)

[Desventajas 50](#_h415km8xxyw2)

[Arq. Multiprocesador 50](#_fvvgl7m3afe6)

[Arq. Cliente-Servidor 50](#_dp37as1xcplj)

[Arquitectura de componentes distribuidos 52](#_vd1v4dtz6b36)

[Computación distribuida interorganizacional 52](#_l0ohy3cfz03h)

[Codificación 52](#_mqxxf6vl6q4p)

[Clase 7: Gestión de proyecto(métricas y estimaciones) 53](#_gx65adfn21rg)

[Métricas 53](#_3hvk14u3fr5z)

[Objetivos 53](#_he5o0n4szz0c)

[Definiciones 54](#_ck1rrep2ka5l)

[Métricas de proyecto 54](#_cdksb234fns5)

[Métricas de proceso 54](#_ipu8kgnedbcl)

[Métricas de producto dinámicas y estáticas 55](#_9oa8ay9nq7bi)

[Dinámicas 55](#_smyons2z9wkb)

[Estáticas 55](#_eqz9r5wpau3i)

[Métrica de Punto función 57](#_ck4v96np1vju)

[GQM 58](#_puh7av38fnje)

[Estimaciones 59](#_4k6al9c2k8a8)

[Estimaciones de recursos 59](#_jo8za6g4u7r8)

[Estimaciones de costos 60](#_rjo0xxdam1q)

[Estimaciones de tiempo 60](#_qz4scqiwpfn0)

[Técnicas de estimación 61](#_de3rqhpxqw78)

[Planning poker 61](#_dbbisjc83v2)

[Modelos empíricos de estimación 62](#_tk5bld91za1v)

[COCOMO II 62](#_o77qq1q09q5p)

[Teoría 8: Caja negra y caja blanca 62](#_3y41j9kzkvgt)

[Tipos de defectos 63](#_ycu8hoveqglm)

[Clasificación ortogonal de defectos 63](#_o8u4ubwfjoh)

[Prueba de software 63](#_z9bazbru2cq8)

[Caja negra 64](#_4rxwlvgxvc9p)

[Partición equivalente 64](#_uek3h0ogl547)

[Análisis de valores límite 64](#_wmzarau08fd)

[Caja blanca 64](#_k1bm2g4mwvbp)

[Prueba del camino básico 65](#_k99looxa17x9)

[Teoría 9: Estrategias de prueba 66](#_3n7v7e43675q)

[Validacion y verificacion 67](#_8anlcmuwmjkb)

[Pruebas de unidad 67](#_renuc6z0d1fc)

[Procedimiento 68](#_2ao2sz52q2u9)

[Pruebas de integración 68](#_h78oyppn289s)

[Pickear un tipo de prueba 70](#_v0ofh6sp3lbj)

[Pruebas de regresión 70](#_jvww4qx4ltm4)

[Criticidad 70](#_2s71d1nlvd7p)

[Integración y unidad en OO 70](#_hrufe6hui7y)

[Pruebas del sistema 71](#_d8u5c0tvbdvy)

[Pruebas de validación 71](#_ej2f2o53g4p0)

[Pruebas ALFA y BETA 71](#_uc2neeal7bca)

[ALFA 71](#_uczvv52rpcbz)

[BETA 71](#_8js1xw3d2ywe)

[Prueba de entornos especializados 72](#_d2suzlbonqxt)

[Prueba de arquitectura cliente-servidor 72](#_krv2ysr7rx5l)

[Prueba de la documentación y funciones de ayuda 72](#_6tu47ql9as2a)

[Pruebas de sistemas de tiempo real 72](#_hfov5f6sse4p)

[Depuración 73](#_xrjy65p95nw8)

[Teoría 10: Mantenimiento 74](#_d1jicl85xm9j)

[Leyes de Lehman 74](#_yv8weu5y3d8w)

[Características del mantenimiento 74](#_rin7ylckg3ef)

[Ciclo de mantenimiento 75](#_9kdynxn1jiuy)

[Tipo de mantenimiento: Mantenimiento correctivo 76](#_qf44v8dwcoi3)

[Tipo de mantenimiento: Mantenimiento Adaptativo 76](#_xc4n6lv68chh)

[Tipo de mantenimiento: Mantenimiento Perfectivo 76](#_o44q5n59zdgr)

[Tipo de mantenimiento: Mantenimiento Preventivo 77](#_lxg3ta2c20m)

[Rejuvenecimiento de software 78](#_3rrjcuyuu5cs)

[Reestructuración 78](#_kkiwm5dw2pka)

[Re-documentación 78](#_ni9mt0z1kco)

[Ingeniería Inversa 79](#_vteniiz1q1gv)

[Re-ingeniería 79](#_2p9w2xrzp9z8)

[Teoría 11: Auditoría Informática 79](#_c0ts7pk4s0td)

[Auditor informático 80](#_eqj9sld6zd2k)

[Donde trabaja un auditor 80](#_alddu4odf3pe)

[Etapas de la auditoría informática 81](#_lwsn9sgw0ml)

[Planeación 81](#_a2hygdazeml8)

[Ejecución 81](#_x6c6ceme761n)

[Análisis de los datos recabados y condiciones observadas 81](#_fyafi7gut7sy)

[Elaboración de un informe escrito y emisión de opinión 81](#_3uan0788se52)

[Finalización de auditoría 81](#_1pbw95l8p3po)

[Tipos de auditoria 81](#_9t4fahl1p8d0)

[Interna 81](#_g9ytvgir20xy)

[Externa 82](#_uwc2hl63xqf3)

[Principios de la auditoría 82](#_ovkc459kzw4u)

[Metodología Octave para auditoría informática 83](#_w8j3o2f2n3t0)

[Metodología Margerit para auditoría informática 84](#_hpwd0t38xrwi)

[Controles chequeados 84](#_nh6cgbeoxcf3)

[Ejemplo de reporte  
Recomendaciones y plan de acción 84](#_68x8wtidarw9)

[Auditor vs Perito vs Consultor 85](#_k39vgeo9vhff)

# Teoría 1: Repaso, comunicación y SRS.

## **Proceso de software**

Un proceso de software es un conjunto de actividades y resultados asociados que producen un producto de software.

Estos fueron vistos el año pasado (la etapa elicitación y especificación de requerimientos). Específicamente nosotros usamos SCRUM.

## **Modelo**

Representación abstracta de un proceso de software.

Los modelos de proceso de software cuentan con actividades fundamentales (se presentan en todos los modelos):

1. Especificación del software.
   1. Técnicas de elicitación.
   2. Especificación de requerimentos.
2. Desarrollo del software.
3. Validación del software.
4. Evolución del software.

## **Comunicación**

La comunicación es el proceso más importante de la interacción humana, que se produce de forma verbal y no verbal.

## **Elicitación de req**uisit**os**

Proceso que trata de adquirir toda la información relevante para poder producir un modelo de los requerimientos de un dominio de problema.

Objetivos:

1. Conocer el dominio del problema para facilitar comunicación con clientes y usuarios y entender sus necesidades.
2. Conocer el sistema actual.
3. Identificar necesidades (explícitas e implícitas) de clientes, usuarios y las expectativas que tienen del sistema a desarrollar.

### Muestreo de la documentación, los formularios y los datos existentes

Es una técnica de elicitación que recolecta hechos a partir de documentación preexistente. Por ejemplo de:

1. Organigrama (identificación de propietario, usuarios, claves).
2. Memos, notas internas, minutas, registros contables.
3. Solicitudes de proyectos de sistemas informáticos previos.

Permite conocer el historial que origina el proyecto.

Útil cuando el sistema está en funcionamiento y tenemos acceso al mismo. (Por ejemplo si necesitamos crear una transición entre el sistema nuevo y el actual mediante capacitación).

### Observación del ambiente de trabajo

Técnica en la cual el analista se convierte en observador de cómo los trabajadores y sistemas interactúan de forma natural.

Lineamientos de la observación:

1. Determinar quién será observado y cuando.
2. Determinar si pido permiso al observado y explicó la necesidad de la observación, o no (avisar puede intimidar al observado, y conviene hacer una sin avisar y otra si).
3. Mantener bajo perfil (evitar intimidar al trabajador puesto que no actuara como suele trabajar en realidad).
4. Tomar nota.
5. Revisar notas con alguien apropiado.
6. No interrumpir.

### Visitas al sitio

1. Investigar el dominio del problema.
2. Patrones de soluciones (mismo problema en otra organización).
3. Revistas especializadas.
4. Buscar problemas similares en internet.
5. Consultar otras organizaciones.

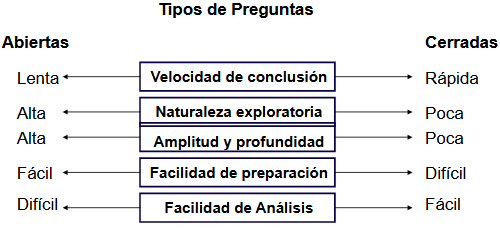
Extracto del video que explica para qué sirve esta técnica:

“Supóngase que es el software para una fabrica, si, yo pienso y desarrollo el software pero después ¿las máquinas donde van a estar?, ¿donde va a estar el operador para cargar los datos? … ¿Puedo poner la computadora en cualquier lado?¿Voy a estar cerca del lugar que necesito?”

### Cuestionarios

2 tipos de preguntas, abiertas y cerradas

Cerradas son las más fáciles de concluir y analizar pero no tienen mucha naturaleza exploratoria ni son amplias o profundas. Las abiertas son lo contrario.



Está bueno tener ambas, lo que está cerrado y categorizado lo hago cerrado y sin lo hago abierto para la próxima.

Sirven para cuando queremos preguntar a muchas personas y también cuando están separadas geográficamente, por lo que en ambos casos sería costoso encontrarse en persona con todos.

### Entrevista

Para preparar una entrevista debemos:

1. Leer los **antecedentes**.
2. Establecer los **objetivos** de la misma.
3. **Seleccionar** los **entrevistados.**
4. **Planificar entrevista** (fecha, hora lugar, duración)
5. **Selección** de **tipo** de **preguntas** y su **estructura**. Definir guión, con lenguaje claro y sin opiniones en las preguntas, que sean concisas, y evitar que sean largas y complejas.

Claramente: Si sabemos poco vamos a tener que explorar un poco más, sino alcanza una sola entrevista habrá que solicitar otra.

3 tipos de organización de entrevistas:

1. Piramidal (inductiva, cerradas a abiertas).
2. Embudo(deductiva, abiertas a cerradas).
3. Diamante(combinación, cerradas a abiertas a cerradas).

El profe recomienda diamante si falta calle.

### JRP

Requiere de una instalación física, más compleja de llevar a cabo al tener que reunir mucha gente. Tiene de bueno que al tener todos los actores involucrados en un mismo lugar se genera un ambiente que permite resolver muchas dudas y discrepancias entre los requerimientos de una.

### Brainstorming

Consiste en que todos los participantes digan todas las ideas que tengan sobre un cierto tema para luego revisarlas para quedarse con las mejores. Promueve el desarrollo de ideas creativas.

## Requerimientos SRS

### Requerimientos

Los requerimientos pasan por 4 etapas:

1. Solicitud
2. Definición
3. Análisis
4. Especificación (acá usamos HU u otro visto por ejemplo).

Tipos de requerimientos:

* Funcionales: Definen el comportamiento del sistema, describen las tareas que el sistema debe realizar, es importante que se defina manteniendo un equilibrio entre generalidad y exceso de detalle al pedo.
* No funcionales: Cosas fuera del sistema, definen aspectos, deseables desde el punto de vista del usuario. O también restricciones como tiempos de respuesta, usabilidad, mantenimiento, etc.

### ConOps (IEEE Std. 1362-1998)

Estándar de documento para describir características del sistema escrito desde el punto de vista del usuario, dirigido para el usuario.

Es la forma en que se puede comunicar la visión general, cualitativa y cuantitativa de las características del sistema entre cliente y desarrollador.

Ofrece formato específico para ser completado, no especifica técnica exacta sino que proporciona guías para poder hacerlo, identifica elementos que al menos deben estar.

### SRS (standard requirements specifications) IEEE Std. 830-1998

Un SRS especifica funciones de un producto de software, programa o conjuntos de programa en un entorno. El documento de especificación de requisitos puede desarrollarlo personal desarrollador o de la parte cliente (conviene ambos).

Características de SRS:

1. Alcance: Brindar una colección de buenas prácticas para escribir especificaciones. Se describen los contenidos y las cualidades de una buena especificación de requerimientos.
2. Naturaleza del SRS: El SRS es una especificación de un producto de software en particular, escrito por uno o más personas (desarrolladores o clientes).
3. Ambiente del SRS: El software puede contener toda la funcionalidad del proyecto o ser parte de un sistema más grande. En el último caso habrá un SRS que declarará las interfaces entre el sistema y su software desarrollado y pondrá qué función externa y requerimientos de funcionalidad tiene con el software desarrollado.
4. Correcto: Un SRS es correcto si cada requisito declarado se encuentra en el software.
5. No ambiguo: Un SRS es inequívoco si y sólo si cada requisito declarado tiene solo una interpretación.
6. Completo: Reconoce todos los requisitos externos impuestos por especificaciones del sistema
7. Consistente: Tiene que estar de acuerdo con los documentos de nivel superior (como una especificación de requerimientos).
8. Priorizado: Es priorizado por la importancia de sus requerimientos particulares.
9. Comprobable: Es comprobable si sus requisitos declarados son comprobables (no ambiguos).
   1. Requisito comprobable: Es comprobable si hay algún proceso para verificar que el producto de software cumple el requisito.
10. Modificable: Su estructura y estilo permite cambios a los requerimientos fácilmente y consistentemente conservando la estructura y estilo.
11. Trazabilidad: Claridad del origen de cada requerimiento y su trazabilidad hacia los requerimientos futuros.
12. Preparación conjunta del SRS: Preparar en conjunto con las partes involucradas y así formar un buen acuerdo.
13. Evolución de SRS: Debe evolucionar junto al software, registrar cambios y responsables y aceptar los mismos.
14. Prototipos: Usa prototipos para la definición de requerimientos.
15. Diseño incorporado en el SRS: Puede incorporar atributos o funciones externas como diseños para interactuar entre subsistemas.
16. Requerimientos incorporados en el SRS: Los detalles particulares son anexados como documentos externos.

#### **Se**cción 1 **del SRS**

* Propósito: Define el propósito del documento y se especifica a quien va dirigido.
* Alcance o ámbito del sistema: Nombra al sistema, explica que hace y no hace el sistema, describe los beneficios, objetivos y metas futuras.
* Referencias: Lista completa de referencias de los documentos usados para escribir el SRS. (título, número de reporte, fecha y publicación de cada documento y las fuentes de donde se obtuvieron).

#### Sección 2 del SRS

Factores generales que afectan al producto y sus requerimientos

* Perspectiva del producto: Se declara si el producto es independiente y totalmente autónomo. Si es parte de un sistema más grande se relacionan los requerimientos de este a la funcionalidad del software e identificar las interfaces entre ese sistema y el software.
* Funciones del sistema: Resumen de las funciones del sistema organizadas y puede usar gráficos para las relaciones entre las mismas.
* Características del usuario: Características generales de los usuarios del producto (nivel educativo, experiencia y especialización técnica).
* Evoluciones previsibles del sistema: Requerimientos a implementar en el futuro.

#### Sección 3 del SRS

Requerimientos no funcionales del software a detalle para el diseño del sistema y que se pueda probar que satisface estos requerimientos

* Requerimientos de rendimiento: Relacionados a la carga esperada para el sistema. Debe ser mesurable. Por ejemplo, tiempo esperado por transacción (por ejemplo, que todas tarden un 1 segundo)
* Seguridad: Relacionados a la protección de accesos, usos, sabotajes, modificaciones y destrucciones maliciosas o accidentales. Por ejemplo, logs de actividad
* Portabilidad: Relacionados a la facilitación de traslado a otras plataformas o entornos. Por ejemplo, el uso de un determinado lenguaje por su portabilidad.

#### Sección 4 del SRS

Identifica el tipo de mantenimiento necesario para el sistema y quien realiza estas tareas junto a cada cuanto se deben de hacer. Por ejemplo, generación de estadísticas de acceso semanal por el desarrollador.

#### Sección 5 del SRS

Información relevante que no forma parte del SRS. Por ejemplo, casos de uso.

## Comunicación

Menos es más, regla de Guy Kawasaki dice:

La presentación debe contar con no más de 10 diapositivas, no sobrepasar los 20 minutos de duración y no contener tipografías con cuerpo menor a 30. La idea es simplificar y alivianar las diapositivas.

### Consejos de comunicación

1. No digas textual la presentación o no la leas al pie de la letra.
2. Usa la zona de notas de las diapositivas.
3. No mires la pantalla.
4. Colocarse en el centro del escenario.
5. Una idea por diapositiva.
6. Tamaño de fuente superior a 32.
7. No animar el texto.
8. Usar fuentes legibles como Times New Roman.

### Consejos de cierre

* Cerrar con sesión de preguntas.
* Resumir lo expuesto.
* Invitar al contacto por medios electrónicos.

### Detalles estéticos:

* Colores: Conviene contraste entre el fondo y el texto, es decir colores oscuros para el texto y pasteles pálidos para el fondo, o al revez.
* Textos: Mantener el formato de la fuente, tipo de fuente clara y fácil de leer.
* Imágenes: Se deben usar como ayudas visuales para permitir visualizar conceptos que de otra forma serían difíciles de entender.
* Transiciones y animaciones: No pasarse con las animaciones, y usar transiciones naturales como texto que cae.
* Formato de las diapositivas: Cada párrafo representa una idea.

## Elevator pitch

Utilizado para comunicar una idea o proyecto dentro de un tiempo similar al de subir por un ascensor.

Pasos del video del Elevator pitch

1. Afirmación o pregunta sorprendente para llamar la atención
2. Presentación: contar quién eres
3. Problemas o necesidades que cubres
4. ¿Qué soluciones aportas?
5. ¿Qué beneficio principal obtiene la gente contigo?
6. ¿Por qué tu proyecto es el idóneo?
7. Termina con una llamada a la acción final

# Teoría 2: GCS, planificación organizativa.

## Gestión de la configuración de software (GCS)

### Cambios

Durante el proceso de software pueden ocurrir cambios grandes o chicos de ciertas partes del sistema en construcción que pueden ser un perjuicio para el desarrollo de no ser controlados correctamente. Estos cambios deben ser reportados a las personas correctas o llevará a que el proceso se nos vaya de las manos.

### GCS

Es un conjunto de actividades orientadas a gestionar el cambio.

Gestión de la configuración es el proceso de:

1. Identificar y definir elementos del sistema.
2. Controlar los cambios de estos elementos a lo largo de su ciclo de vida.
3. Registrar y reportar el estado de los elementos y las solicitudes de cambio.
4. Verificar que los elementos estén completos y correctos.

Actividad de **autoprotección** aplicada durante el proceso de software.

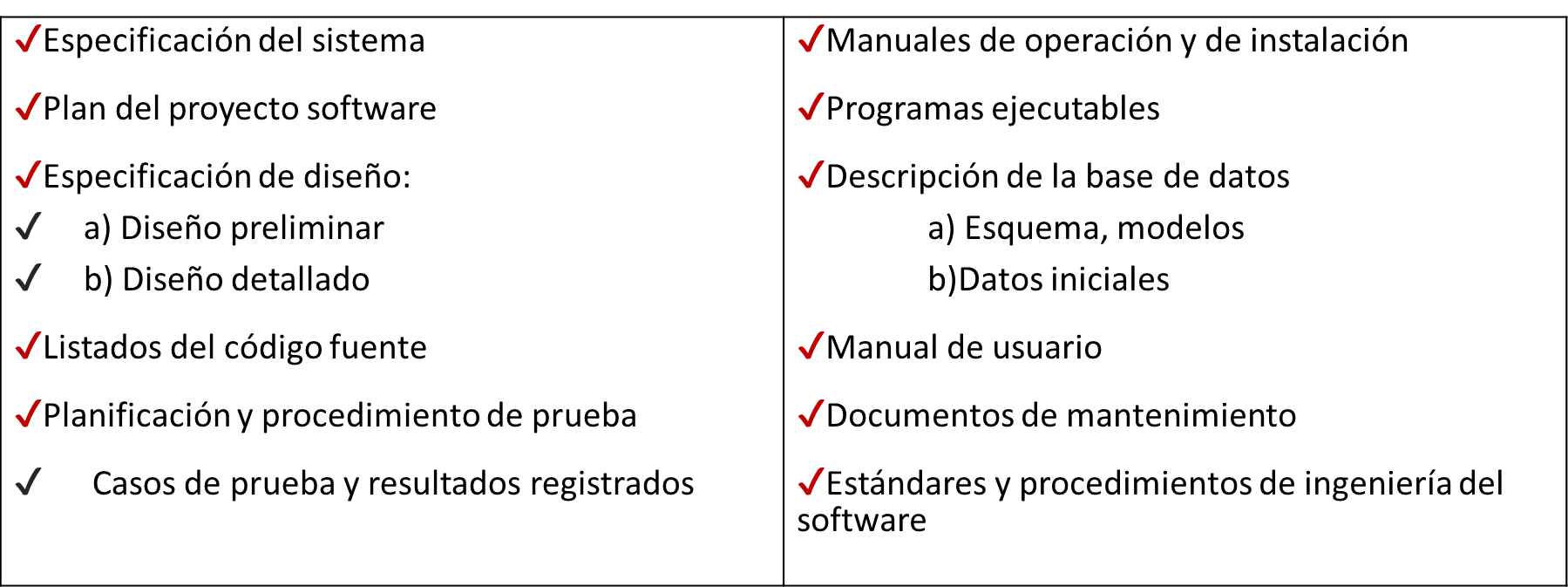
### Involucrados de la GCS

Todos los involucrados en el proceso de software se relacionan en cierta medida con la gestión del cambio, aunque a veces se crean de apoyo especializadas para el proceso ACS.

### Elementos de la GCS (ECS)

1. Programas.
2. Productos de trabajo.
3. Datos.

¿Cuáles podrían ser elementos de la configuración?



### Organización de un repositorio de ECS

Diagrama

Descripción generada automáticamente

### Origen del cambio

1. Condiciones de negocio o mercado cambiantes afectan los requerimientos.
2. Nuevas necesidades de las partes interesadas demandan modificaciones en funcionalidades, datos o servicios del sistema.
3. La reorganización, crecimiento o reducción de la empresa altera las prioridades del proyecto o la estructura del equipo.
4. Restricciones en el presupuesto o calendario llevan a redefiniciones del sistema.

Las actividades de la Gestión del Cambio en Software (GCS) se centran en:

1. Identificar el cambio.
2. Controlar el cambio.
3. Garantizar una implementación adecuada del cambio.
4. Informar a todas las partes afectadas por el cambio.

### Definición de Línea base

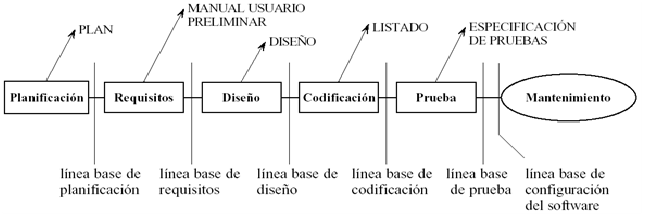
Punto de referencia en el desarrollo de software.

#### Definición de la IEEE

Una especificación o producto que se ha revisado formalmente y sobre el que se ha llegado a un acuerdo, y que de ahí en adelante sirve como base para un desarrollo posterior y que puede cambiarse solamente a través de procedimientos formales de control de cambio

#### En el contexto de la Ingeniería de Software

Una línea base es un punto de referencia en el desarrollo del software que queda marcado por el envío de uno o más ECS y su aprobación





### Importancia del GCS

A qué preguntas el GCS le da respuesta:

1. ¿Cómo identifica y gestiona una organización las diferentes versiones existentes de un programa (y su documentación) de forma que se puedan introducir cambios eficientemente?
2. ¿Cómo controla la organización los cambios antes y después de que el software sea distribuido al cliente?
3. ¿Quién tiene la responsabilidad de aprobar y de asignar prioridades a los cambios?
4. ¿Cómo podemos garantizar que los cambios se han llevado a cabo adecuadamente?
5. ¿Qué mecanismo se usa para avisar a otros de los cambios realizados?

### Proceso

Define tareas con objetivos precisos:

1. Identificación
2. Control de versiones
3. Control de cambios
4. Auditorías de la configuración
5. Generación de informes

Diagrama

Descripción generada automáticamente

#### Identificación

A cada elemento de la GCS se le da:

1. Nombre: cadena de caracteres sin ambigüedad
2. Descripción: lista de elementos de datos que identifican:
3. Tipo de ECS (documento, código fuente, datos)
4. Identificador del proyecto
5. Información de la versión y/o cambio

#### Control de versiones

Combinación de procedimientos y herramientas para gestionar las versiones de los ECS que se crean a lo largo del proceso de software.

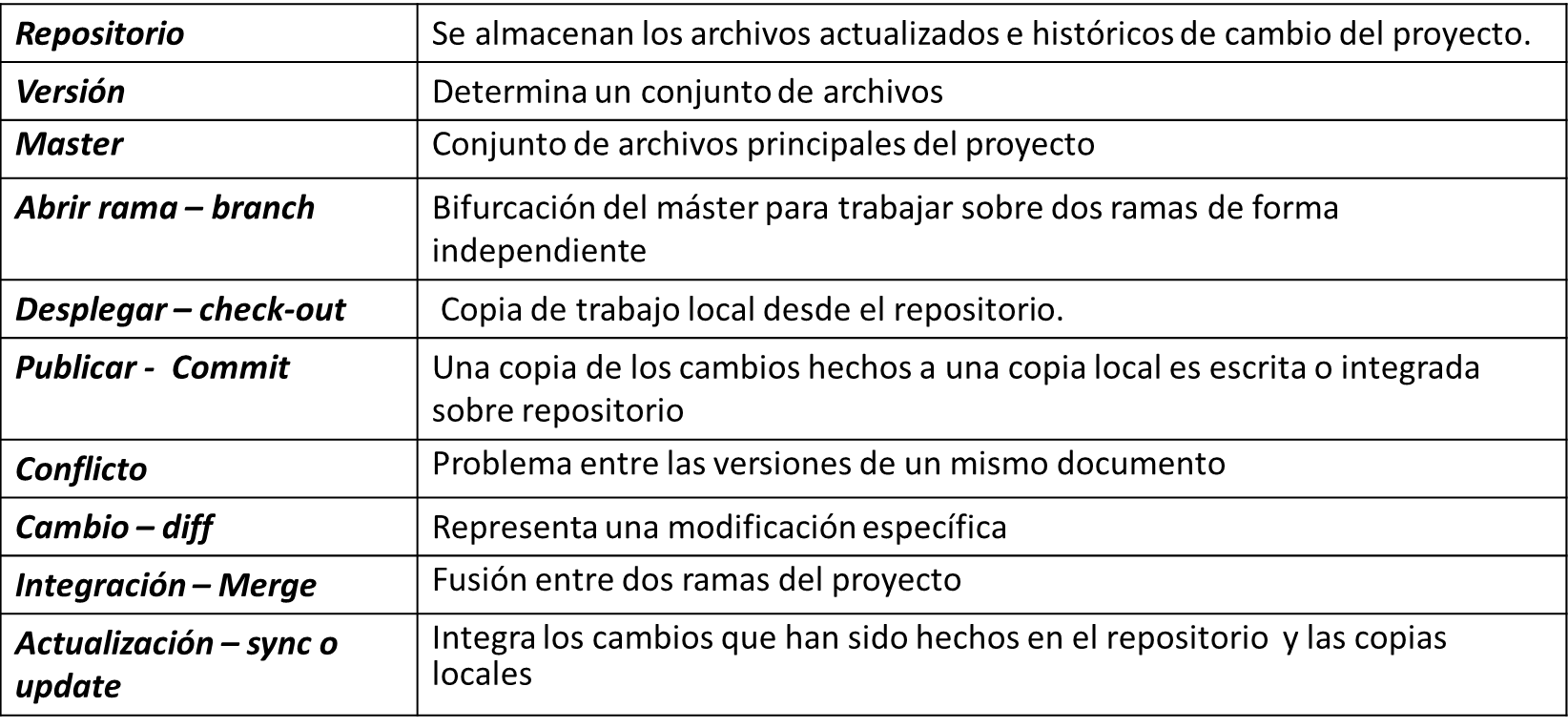
Ejemplo de versiones

Un programa puede contener los módulos 1-2-3-4-5

Una versión puede utilizar los módulos 1-2-3-5

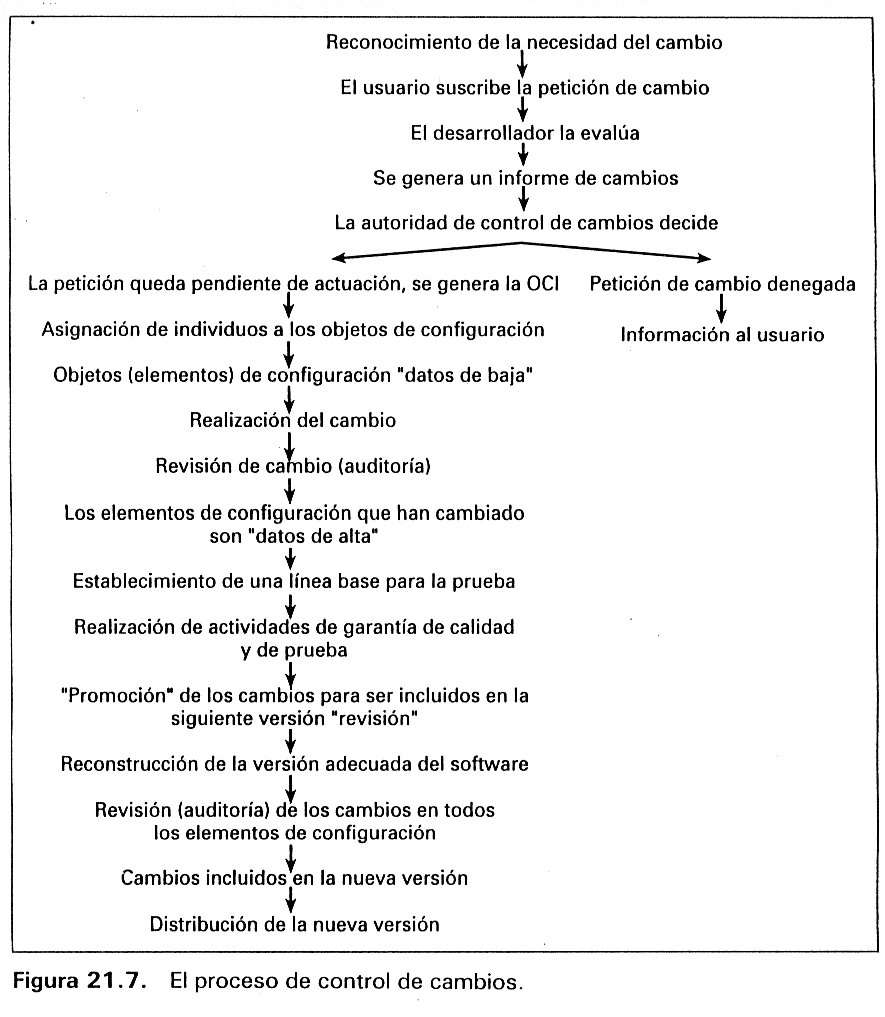
Otra versión puede utilizar los módulos 1-2-4-5

Dos variantes de un mismo programa



#### Control de cambios

En el desarrollo de proyectos, los cambios son inevitables y un control efectivo es crucial. Se emplea una combinación de procedimientos humanos y herramientas adecuadas para establecer un mecanismo eficiente de control de cambios.



#### Auditoría de la configuración

La identificación y control de versiones, junto con el control de cambio, son esenciales para mantener el orden en el equipo de desarrollo de software.

Para garantizar la generación de la orden de cambio usamos:

* Revisiones técnicas formales
* Auditorías de configuración

Auditoría de la configuración responde:

1. ¿Se ha hecho el cambio especificado en la Orden de Cambio?¿Se han incorporado modificaciones adicionales?
2. ¿Se ha llevado a cabo una RTF para evaluar la corrección técnica?
3. ¿Se han seguido adecuadamente los estándares de IS?
4. ¿Se han reflejado los cambios en el ECS: fecha, autor, atributos?
5. ¿Se han seguido procedimientos de GCS para señalar el cambio, registrarlo y divulgarlo?
6. ¿Se han actualizado adecuadamente todos los ECS relacionados?

#### Generación de informes de estado de la configuración (auditoría)

Responde

1. ¿Qué pasó?
2. ¿Quién lo hizo?
3. ¿Cuándo pasó?
4. ¿Qué más se vio afectado?

La generación de informes de estado de la configuración desempeña un papel vital en el éxito del proyecto.

## Gestión de Proyectos

Un proyecto es un esfuerzo temporal que se lleva a cabo para crear un producto, servicio o resultado único.

Características

1. Temporal: Tiene un comienzo y fin definido.
2. Resultado: Productos, servicios o resultados únicos
3. Elaboración gradual: Desarrollar en pasos e ir aumentando mediante incrementos

### Las 4 P de la Gestión de Proyectos de Software

1. Personal (RRHH): Es el elemento más importante.
2. Producto: El producto de software es intangible. A veces es difícil ver el progreso del proyecto.
3. Proceso: Un proceso de software proporciona el marco de trabajo desde el cual se puede establecer un plan detallado para el desarrollo del software.
4. Proyecto: Los proyectos deben ser planeados y controlados para manejar su complejidad.

### Resultados de la mala gestión

1. Incumplimiento de plazos,
2. Incremento de los costos,
3. Entrega de productos de mala calidad.

### Elementos clave de la gestión de proyectos

1. Métricas
2. Planificación
   1. Estimaciones
   2. Calendario temporal
   3. Organización del personal
3. Análisis de riesgos
4. Seguimiento y control

## Planificación

La planificación específica:

1. Que debe hacerse.
2. Qué recursos se van a usar.
3. Qué orden se va a tomar.

Establece una secuencia operativa.

### Planificación organizativa

El personal en una organización de software es su activo más valioso y capital intelectual. Una gestión deficiente del personal es un factor clave en el fracaso de proyectos. Por ello, se implementó el Modelo de Madurez de Capacidades de Personal (P-CMM), que destaca la necesidad de mejorar constantemente la capacidad de atraer, desarrollar, motivar, organizar y retener al personal.

### Participantes

1. Gerentes ejecutivos (dueños del producto): Definen los temas empresariales
2. Gerentes de proyecto (líderes de equipo): Planifican, motivan, organizan y controlan a los profesionales
3. Profesionales especializados: Aportan habilidades técnicas
4. Clientes: Especifican los requerimientos
5. Usuarios finales: Interactúan con el software

### Líderes de equipo

Los líderes de equipo en el desarrollo de software deben optimizar las habilidades individuales. Las prácticas de líderes destacados incluyen:

1. Modelar el camino: Practicar lo que predican y comprometerse con el equipo y el proyecto.
2. Inspirar y compartir visión: Motivar miembros del equipo y desde el inicio del establecimiento de metas involucrar a todos.
3. Desafiar el proceso: Buscar formas innovadoras de mejora, alentar la experimentación y asumir riesgos.
4. Permitir que otros actúen: Fomentar habilidades colaborativas, construir confianza y facilitar relaciones.
5. Alentar: Celebrar logros individuales y fomentar un sentido de comunidad en el equipo.

### El equipo de software

La estructura para nuestro equipo de software depende del estilo gerencial de la organización, la cantidad de gente que conformará el equipo, el nivel de habilidad, y la dificultad del problema.

Los equipos no deberían tener más de 10 miembros, aunque depende del tamaño del proyecto.

Factores a considerar cuando se planea una estructura de equipo

1. Dificultad del problema a resolver
2. Tamaño del programa resultante
3. Tiempo que el equipo permanecerá unido
4. Grado en que puede dividirse en módulos el problema a resolver
5. Calidad y confiabilidad requerida por el sistema a construir
6. Rigidez de la fecha de entrega
7. Grado de sociabilidad requerido en para el proyecto

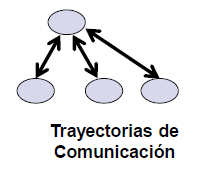
Problemas y cómo evitarlos :

1. Atmósfera de trabajo frenética: El equipo tiene acceso a toda la información y las principales metas y objetivos no se modifican a menos que sea absolutamente necesario.
2. Fricción entre los miembros del equipo: se define responsabilidad a cada uno .
3. Proceso de software fragmentado o mal coordinado: entender lo que se va a crear y permitir que el equipo seleccione el modelo de proceso.
4. Definición imprecisa de roles: definir enfoques correctivos ante un miembro que no cumple.
5. Exposición continua y repetida al fracaso: utilización de técnicas basadas en la retroalimentación y solución de problemas.

### Paradigmas Organizacionales del equipo

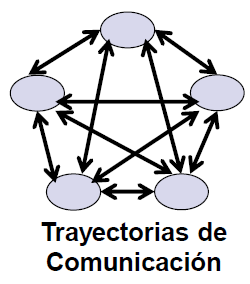
#### Paradigma cerrado

* Estructurado conforme a una jerarquía tradicional (comunicación vertical entre jefe y miembros del equipo).
* Trabajan bien produciendo software similar al de esfuerzos anteriores.
* Menos innovadores.



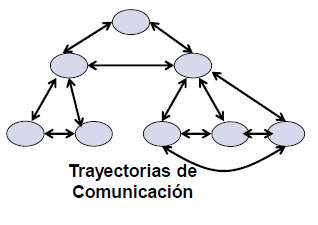
#### Paradigma abierto

* Se intenta estructurar para lograr algunos controles asociados con el paradigma cerrado pero mucha innovación como el paradigma aleatorio.
* Trabajo hecho de forma colaborativa.
* Gran comunicación.
* Toma de decisiones consensuadas. (Equipo democratico).
* Estructura útil para resolución de problemas complejos. Pueden no desempeñarse tan eficazmente como otros equipos.



#### Paradigma síncrono

* Apoyado en la compartimentalización natural de un problema.
* Organiza a los miembros del equipo para trabajar en trozos del problema con poca comunicación activa entre ellos.
* Equipo descentralizado y controlado a la vez (divide y vencerás).
* Bueno para problemas de gran tamaño y complejidad.
* Fraccionamiento puede llevar a que algunos subgrupos produzcan algo que no sirva debido a la comunicación.



#### Paradigma aleatorio

* Estructura el equipo de manera holgada y depende de la iniciativa individual de los miembros.
* Los miembros se organizan entre ellos y no requiere de un líder definido.
* Las trayectorias de comunicación pueden ser cualquiera de los tres paradigmas.
* Destaca cuando se requiere innovación. Pero es problemático cuando se requiere “desempeño ordenado”.

### El equipo de software

Sin importar la organización del equipo, el objetivo es un equipo con alta cohesión ya que estos suelen ser más productivos y motivados.

Beneficios de alta cohesión:

1. Pueden establecer propios estándares de calidad.
2. Los individuos aprenden de los demás y se apoyan mutuamente.
3. El conocimiento se comparte.
4. Se alienta la refactorización y el mejoramiento continuo.

Un equipo no consolidado suele sufrir toxicidad de equipo :

* + 1. Atmósfera de trabajo frenética.
    2. Fricción entre los miembros del equipo.
    3. Proceso de software fragmentado o mal coordinado.
    4. Definición imprecisa de roles.
    5. Exposición continua y repetida al fracaso.

#### Comunicación Grupal

* Las comunicaciones en un grupo se ven influenciadas por factores como: status de los miembros del grupo, tamaño del grupo, composición de hombres y mujeres, personalidades y canales de comunicación disponible.
* Se deben establecer mecanismos para la comunicación formal e informal entre los miembros del equipo.
* La comunicación formal: a través de reuniones, escritos y otros canales no interactivos.
* La comunicación informal: Los miembros comparten ideas sobre la marcha.

#### Equipo ágil

Muchas organizaciones defienden el desarrollo ágil de software, lo que conlleva a crear equipos pequeños y muy motivados, lo que se denomina equipo ágil.

* Se hace hincapié en la competencia individual y colaboración grupal.
* Son autoorganizados.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Se parece a un paradigma aleatorio con pocos miembros.

# Teoría 3: Riesgos.

## Riesgo

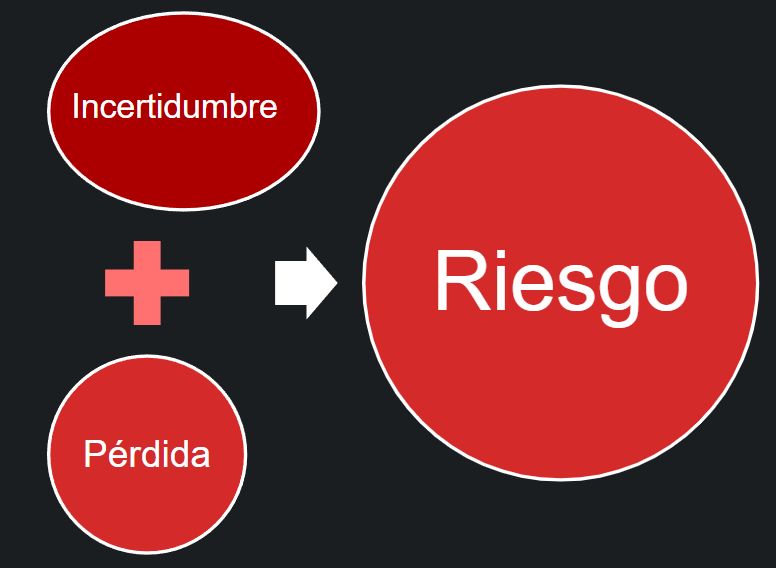
1. Evento no deseado que acarrea consecuencias negativas.
2. Los gerentes deben determinar la probabilidad de que ocurran durante el desarrollo y planear para evitarlos o mitigarlos.
3. El riesgo concierne a:
   1. Lo que ocurrirá en el futuro: ¿Cuáles son los riesgos que pueden hacer que fracase el proyecto?
   2. Cómo afectarán los cambios al desarrollo: ¿Como afectarán al éxito global y a los plazos los cambios en los requisitos del cliente, en las tecnologías de desarrollo, etc…?
   3. A las elecciones: ¿Qué métodos y herramientas debemos usar, cuánta gente debe estar involucrada, cuánta importancia hay que darle a la calidad?

### Deuda técnica

* Término que describe el costo asociado con aplazar actividades como documentación y refactorización.
* No sé paga literalmente sino que hace que el producto sea de mala calidad.
* Implica que el uso de recursos, tiempo y esfuerzo para luchar contra los temas técnicos puede reducirse si se afrontan los problemas al inicio.
* En Agile se sigue gestionando los riesgos, porque si no generaría deuda técnica.

### Estrategias de gestión de riesgos:

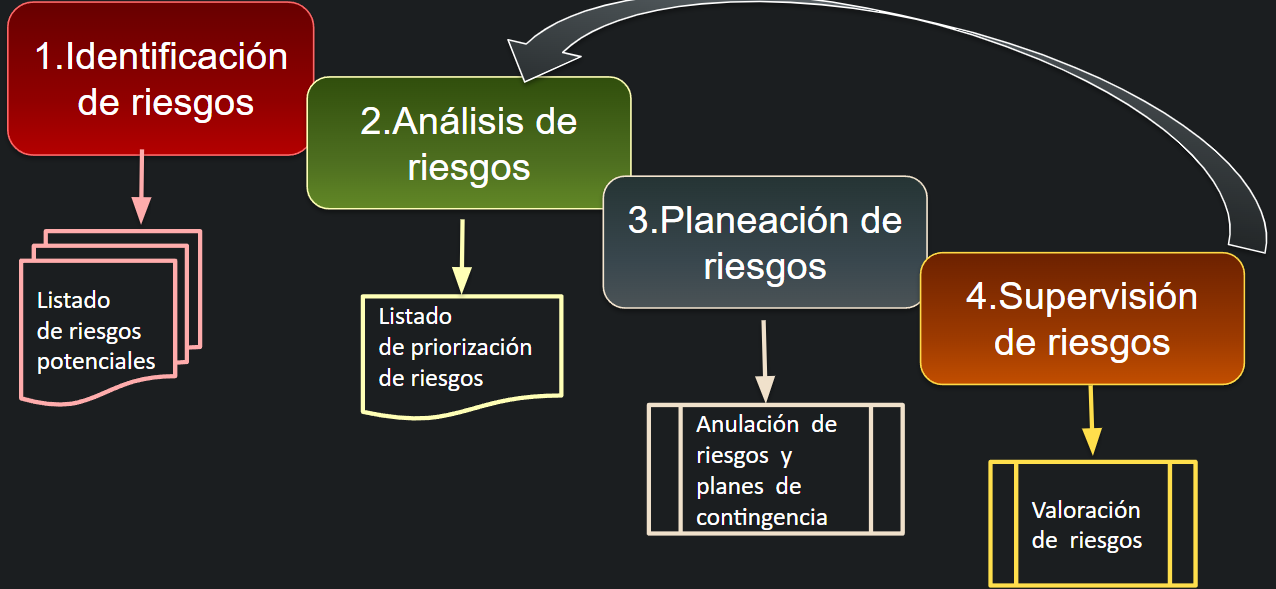
1. Reactiva: Reaccionar contra el problema y “gestionar la crisis” a lo Indiana Jones.
2. Proactivas: Plantear estrategias de tratamiento de riesgos antes de que ocurran.



### Categorización de riesgos

1. Proyecto: Por ejemplo presupuesto, recursos y calendario.
2. Producto: Por ejemplo calidad o cambios en requerimientos.
3. Negocio: Por ejemplo sistema que nadie quiere, sistema que no se adapta a la empresa, o la gerencia deja de apoyar el producto.

## Proceso de gestión de riesgos



El proceso de gestión de riesgos consiste en un esfuerzo grupal e iterativo, que sirve para gestionar los riesgos y la forma de lidiar con ellos a lo largo del proyecto.

### Identificación de riesgos

Generalmente se usa un enfoque un “Brainstorming” con nuestro grupo de trabajo y generamos este listado, o en base a la experiencia.

Los riesgos pueden ser:

* Conocidos.
* Predecibles.
* Impredecibles.

Además debemos determinar si cada uno es un riesgos de proyecto son de:

* Proyecto.
* Predecibles.
* Impredecibles.

Hay ciertas preguntas que nos podemos hacer, si la respuesta es negativa, muy probablemente estemos ante un riesgo:

1. ¿Los gerentes de software y de cliente se reunieron formalmente para apoyar el proyecto?
2. ¿Los usuarios finales se comprometen con el proyecto y sistema/producto que se va a construir?
3. ¿El equipo y sus clientes entienden por completo los requisitos?
4. ¿Los clientes se involucraron plenamente en la definición de los requisitos?
5. ¿Los usuarios finales tienen expectativas realistas?
6. ¿El ámbito del proyecto es estable?
7. ¿El equipo tiene la mezcla correcta de habilidades?
8. ¿Los requisitos del proyecto son estables?
9. ¿El equipo tiene experiencia con la tecnología que se va a implementar?
10. ¿El número de personas que hay en el equipo es adecuado para hacer el trabajo?
11. ¿Todos los clientes/usuarios están de acuerdo en la importancia del proyecto y en los requisitos para el sistema/producto que se va a construir?

Ejemplos de riegos:  


### Análisis de riesgos

Acá a cada riesgo del listado lo calificamos según su probabilidad de ocurrir e impacto. Luego elegimos aquellos a priorizar según las calificaciones que le dimos y hacemos una “línea de corte” (más tarde se explica), esos serán nuestros riesgos a tratar.

Calificaciones de probabilidad de riesgo:

* Bastante improbable : < 10%
* Improbable : 10-25%
* Moderado : 25-50%
* Probable :50-75%
* Bastante probable : >75%

Calificaciones de impacto de riesgo:

* Catastrófico: cancelación del proyecto
* Serio: reducción de rendimiento, retrasos en la entrega, excesos importante en costo
* Tolerable: reducciones mínimas de rendimiento, posibles retrasos, exceso en costo
* Insignificante: incidencia mínima en el desarrollo

Tabla de riesgos de ejemplo:

| Riesgos | Categoría | Probabilidad | Impacto |
| --- | --- | --- | --- |
| El cliente cambiará los requisitos | Proy | 80% | 2 |
| Falta de formación en las herramientas | Proy | 80% | 3 |

Boehm recomienda supervisar los 10 riesgos más altos (hacer la línea de corte a ese nivel), según el profe alrededor de la mitad si son poco, sino el 10%, siempre que sea un número manejable.

Aquellos elegidos para supervisar se le prestara atención, los otros serán reevaluados y tendrán un prioridad de segundo orden.

Los riesgos de gran impacto con poca probabilidad no toman un tiempo significativo

Los riesgos de gran impacto con media o mucha probabilidad y los de poco impacto con mucha probabilidad se toman en cuenta

Ejemplo:

| ***Riesgos*** | ***Categoría*** | ***Probabilidad*** | ***Impacto*** |
| --- | --- | --- | --- |
| El cliente cambiara los requisitos | Proyecto | 80% | 2 |
| Falta de formación en las herramientas | Producto | 80% | 3 |
| Menos reutilización de la entrevista | Proyecto | 70% | 2 |
| La estimación del tamaño puede ser muy baja | Proyecto | 60% | 2 |
| Habrá muchos cambios de personal | Proyecto | 60% | 2 |
| La fecha de entrega estará muy ajustada | Proyecto | 50% | 2 |
| **LINEA DE CORTE** | | | |
| Se perderán los presupuestos | Negocio | 40% | 1 |
| Los usuarios finales se resisten al sistema | Negocio | 40% | 3 |
| La tecnología no alcanzará las expectativas | Producto | 30% | 1 |
| Personal sin experiencia | Proyecto | 30% | 2 |
| Mayor número de usuarios de los previstos | Negocio | 30% | 3 |

### Planeación de riesgos

Aquellos riesgos que elegimos tratar, generamos una estrategia para tratarlo:

* Intentamos evitar el riesgo completamente.
* Si no podemos evitarlos, minimizamos la probabilidad de que ocurra.
* Si no podemos minimizar la probabilidad de que ocurran, minimizamos el impacto del mismo.

Supervisión de riesgos, A medida que va pasando el tiempo revisamos:

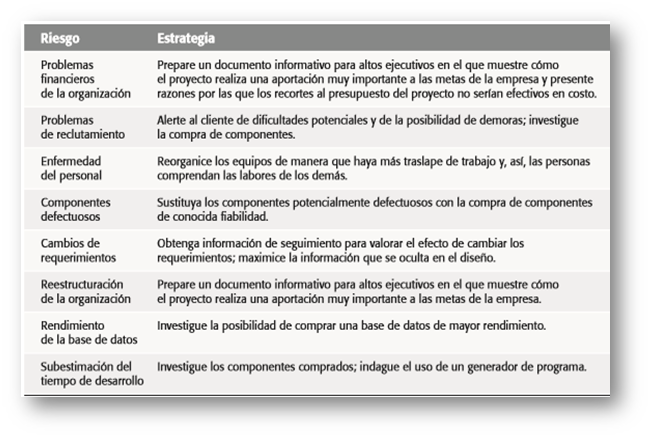
Aparecieron.

* 1. Cómo funcionó el plan de contingencia.
  2. Si hay que sacar o agregar riesgos.
  3. Además de cambiar las probabilidad de que ocurran.
  4. Volvemos al paso 2, análisis de riesgos.

Considera los riesgos sobre la línea de corte y les asigna una estrategia para evitarlos.

* Evitar el riesgo: Diseña el sistema de forma que el evento no pueda ocurrir
* Minimizar el riesgo: Reduce la probabilidad de que el evento ocurra
* Plan de contingencia: Acepta que el evento ocurra y busca minimizar las consecuencias

#### Ejemplo



Exposición de riesgo (E)

* Se determina con la ecuación E = P x C
* P es la probabilidad de ocurrencia para un riesgo
* C es el costo para el proyecto si ocurre el riesgo

Ejemplo:

Riesgo: sólo 70 por ciento de los componentes de software calendarizados para reutilización se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse a la medida.

Probabilidad: 80%

Impacto: Se planificaron 60 componentes de software reutilizables. Si sólo puede usarse 70 por ciento (42 componentes), tendrán que desarrollarse 18 componentes desde cero (además de otro software a la medida que se calendarizó para desarrollo).

Dado que el componente promedio es de 100 LOC (lines of code) y que el costo de la ingeniería del software para cada LOC es US$14.00, el costo global (impacto) para desarrollar los componentes sería 18 x 100 x 14 = US$25.200.

Exposición al riesgo. E= 0.80 X 25.200 = US$20.200.

### Supervisión

* Evaluar si la probabilidad de cada riesgo cambio
* Evaluar que tan efectivas son las estrategias
* Detectar la ocurrencia de un riesgo que fue previsto
* Asegurar que se están cumpliendo los pasos definidos para cada riesgo
* Recopilar información para el futuro
* Determinar si existen nuevos riesgos
* Reevaluar periódicamente los riesgos
* Monitorizar los riesgos en todas las etapas del proyecto. En cada revisión administrativa, es necesario reflexionar y estudiar cada uno de los riesgos clave por separado decidiendo si es más o menos probable que surja el riesgo, y si cambiaron la gravedad y las consecuencias del riesgo

# Clase 4: Conceptos de diseño de software y planificación temporal

## Planificación temporal

Actividad para distribuir el esfuerzo estimado a lo largo de la duración prevista del proyecto. Es muy importante la precisión.

### Calendarización del proyecto

La calendarización del proyecto de software distribuye el esfuerzo estimado a lo largo de la duración planificada, asignándolo a tareas específicas del desarrollo.

**Según el video** la calendarización del proyecto es la designación de una fecha final.

Casos:

1. Fecha establecida por el cliente: El esfuerzo será distribuido en ese plazo.
2. Fecha establecida por los desarrolladores: El esfuerzo se distribuye para conseguir un uso óptimo de los recursos y se define una fecha de fin luego de análisis.

La calendarización va entonces a estar compuesta por tareas:

* Tareas: Secuencia de acciones a realizar en un periodo de tiempo.
* Tarea crítica: Tarea que si se retrasa, retrasa todo el proyecto.
* Hito: Algo que debe estar hecho para una fecha.

#### Tarea

Parámetros:

1. Precursor: Evento o conjunto de eventos que deben pasar antes de que se pueda hacer.
2. Duración: Duración pa’.
3. Fecha de entrega: Fecha final.
4. Resultado: Hito o componente listo.

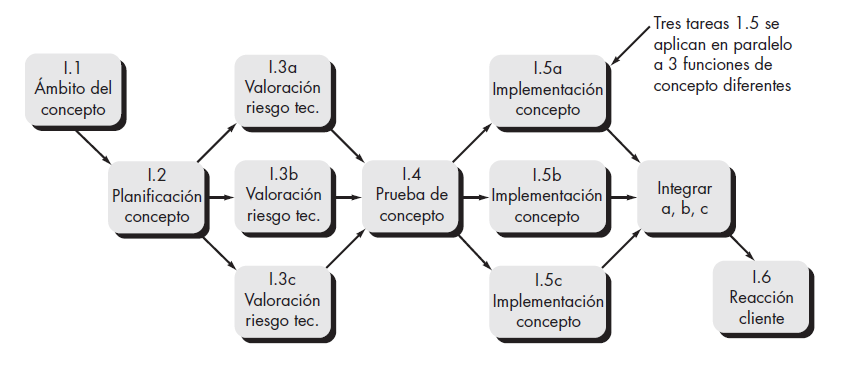
#### Red de tareas:

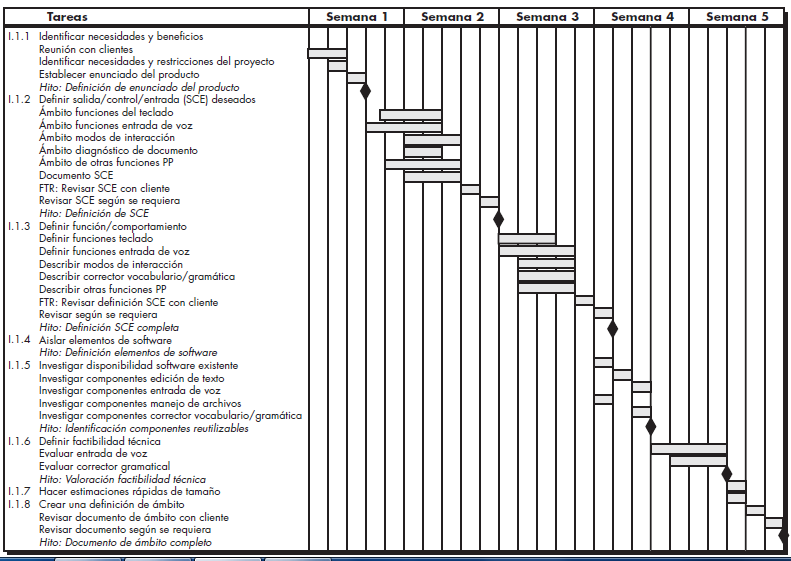
* Representación gráfica del flujo de tareas de inicio a fin.
* Permiten realizar actividades en paralelo.
* Representan la secuencia de tareas y su interdependencia.

Factores para elegir el conjunto de tareas:

1. Tamaño del proyecto
2. Número de usuarios posibles
3. Criticidad del proyecto
4. Estabilidad de los requerimientos
5. Facilidad de comunicación con el cliente/usuario
6. Madurez de la tecnología aplicable
7. Restricciones

(Entre otros).





### PERT: Método de planificación temporal

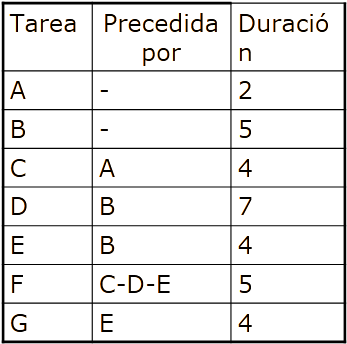
1. Usado para controlar la ejecución de proyectos con muchas actividades de investigación, desarrollo y pruebas.
2. Red de tareas con fechas tempranas, tardías, camino crítico.
3. Probabilístico.

### CPM: Critical path method

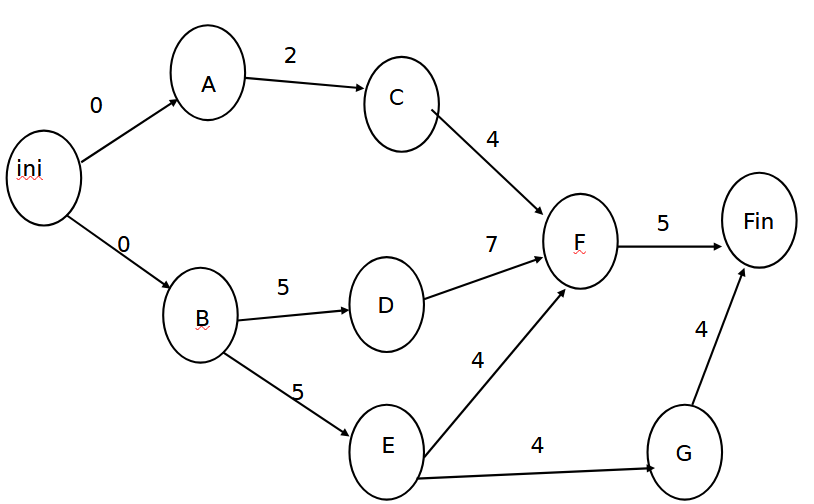
1. Usado para proyectos en los que hay poca incertidumbre en las estimaciones.
2. Tiempo de inicio temprano y tardío.
3. Determinístico.

### Método del camino crítico: CPM y PERT juntos.

1. Establecer lista de tareas.
2. Fijar dependencia entre tareas y duración.
3. Construir la red.
4. Numerar los nodos.
5. Calcular la fecha temprana y tardía de cada nodo:
   1. Tei: Fecha temprana del nodo I
   2. Tai: Tardía del I
6. Calcular el camino crítico que une a las tareas críticas
   1. ⇒ Tei = Tai

Lista de tareas:  


Red:



#### Fechas tempranas y tardías

Fecha temprana: TeJ = TeI y tIJ

TeJ= fecha más temprana del nodo destino.

TeI= fechas más temprana del nodo origen

tIJ= duración de la tareas desde el nodo I hasta el nodo J.

Si hay más de un camino es Max(TeJ1, TeJ2).

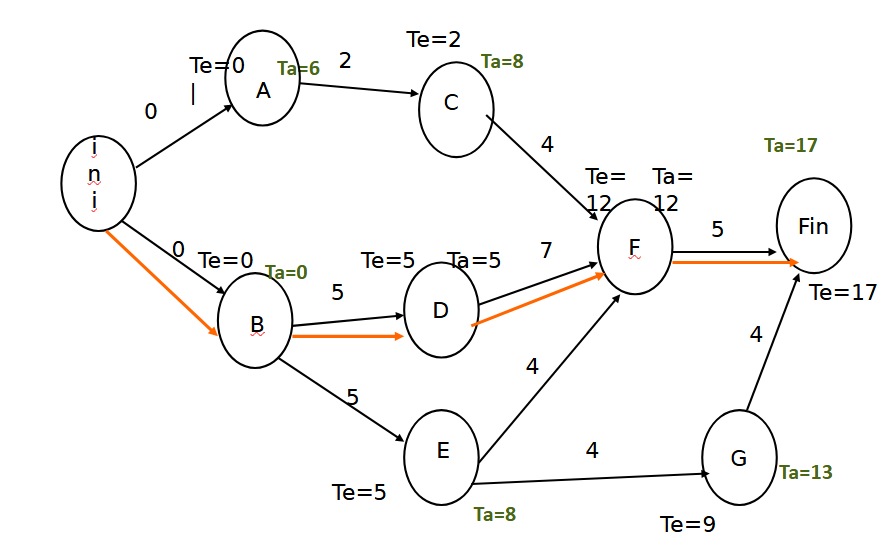
Es decir, si tenemos un nodo A, y tenemos un solo camino y la duración de A es 2. Y el camino solo tiene el nodo inicial (duración 0). Entonces la fecha temprana es TeA = 0 + 0 = 0.

Luego si hay un nodo que sigue a A, de duration 4 llamado C, la fecha temprana de C será = 0 + 2 = 2

De haber un nodo con más de un camino, entonces hay que elegir el máximo.

Imaginemos que a F se puede llegar de 3 lugares, con duración respectivamente de 6, 12 y 9. Entonces:

TeF = max (6, 12 y 9) = 12



Fecha tardía:

TaI = TaJ - tIJ

TaJ = fecha más tardía del nodo destino

tIJ = duración de la tarea desde el nodo I hasta al nodo J

de haber más de un camino sería el Min(TaJ1, Taj2)

Nosotros asumimos que la fecha más tardía de todo el proyecto es la misma que la fecha temprana del fin, las fechas tardías se calculan de atrás para adelante.

Por ejemplo: la de F sería TaF = TaFin - tF= 17 - 5= 12

la de C sería TaC= TaF - tCF = 12 - 4 = 7

Básicamente sería el Ta del anterior - la duración actual.

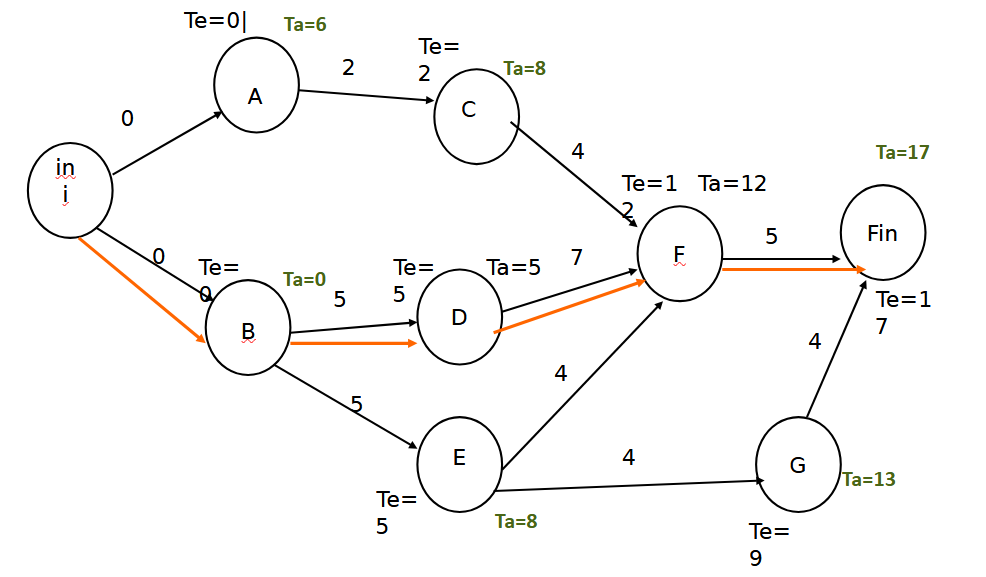
¿Qué pasa si yo tengo un nodo que le salen 2 flechas? por ejemplo B:

TaB = min ((TaE - tBE), (TaD-tBD)) = ((8-5), (5-5)) = 0

### Encontrando el camino crítico

Usamos el “margen total para esto” Que es básicamente:  
Mt = TaF - TeF

Si da 0, entonces es parte del camino crítico.



Con el PERT-CPM se obtiene:

* Camino crítico
* Ventana temporal para cada actividad
* Fecha temprana de inicio de una tarea
* Fecha tardía del inicio de una tarea sin retrasar la finalización del proyecto
* Final más temprano de la tarea
* Final más tardío de la tarea
* Margen total.

¿Qué pasa si una tarea se retrasa?

* Revisar el impacto sobre la fecha de entrega
* Reasignar recursos
* La inclusión de más personas en el desarrollo no siempre genera aumento en la productividad
* Reordenar tareas
* Modificar entrega

## Conceptos de diseño de Software

Siempre después del análisis sigue el diseño en cualquier modelo de proceso.

El diseño es

* El núcleo técnico de la ingeniería de software. Primera actividad técnica a realizar luego del análisis.
* Representación de algo que va a construir.
* Donde los requisitos, necesidades y consideraciones técnicas se unen.

#### Tipos de diseño

* Diseño de datos: Se transforma el modelo de dominio a estructura de datos, relaciones etc.
* Diseño arquitectónico: Define la relación entre los elementos estructurales más importantes del software, y se establecen estilos y patrones de diseño, etc.
* Diseño a nivel de componentes: Transforma los elementos estructurales de la arquitectura en una descripción procedimental de los componentes.
* Diseño de interfaz: Describe la forma de comunicación de nuestro sistema en sí mismo o con el exterior (otros sistemas o personas).

#### Características a evaluar del diseño

* Se debe implementar todos los requerimientos explícitos del modelo de req. además de los req. implícitos que desea el cliente.
* Debe ser una guía legible y comprensible para los que escriben y soportan el software.
* Debe ser completo al proporcionar la visión del software.

#### Criterios técnicos para un buen diseño

1. Debe tener un estructura arquitectónica que *utiliza patrones de diseño reconocibles, formado con componentes de buen diseño e implementado en forma evolutiva.*
2. Debe ser modular.
3. Debe contener distintas representaciones.
4. Debe conducir a estructuras de datos adecuadas y que procedan de patrones de datos reconocibles.
5. Debe conducir a componentes que presenten características funcionales independientes.
6. Debe conducir a interfaces que reduzcan la complejidad de las conexiones entre módulos y el exterior.
7. Debe derivarse mediante un método iterativo y controlado por la información obtenida durante el análisis.
8. Debe representarse por medio de una notación que sea entendible.

### Evolución del diseño de software

Todos lo modelos d sueño de software tienen en común:

1. Mecanismo para traducir el modelo de req. a un diseño
2. Notación para rep. componentes funcionales e interfaces.
3. Heurísticas para refinamiento
4. Lineamientos para evaluación de calidad

### Conceptos de diseño

#### Abstracción

Para no preocuparse por los detalles de bajo nivel cuando resolvemos un problema.

* Procedimental: Secuencia e interacción que tiene una funcionalidad específica.
* De datos: Datos que definen un objeto real.

#### Arquitectura del software

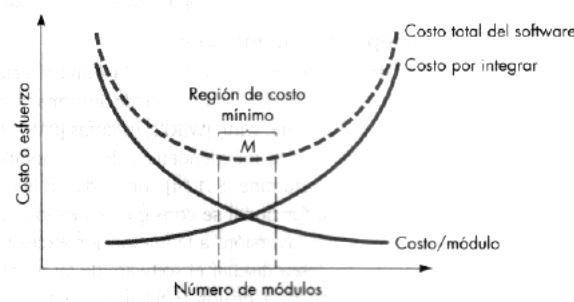
Estructura general del software y como esta da integridad conceptual al sistema.

#### Patrones

Estructura de diseño que resuelve un problema específico dentro del contexto.

#### Modularidad

El software se divide en módulos que se integran para satisfacer los requisitos.



#### Ocultamiento de información

Información dentro de un módulo inaccesible por otros.

#### Independencia funcional

Modularidad + Abstracción + Ocultamiento. Se busca alta cohesión (que todo se relacione para una misma función) y bajo acoplamiento (interconexión entre módulos).

|  |  |
| --- | --- |

#### Refinamiento

Complementario a la abstracción, se refina iterativamente y ayuda a revelar los detalles de menor grado.

#### Refactoring

Lo de objetos.

### Principios del modelado de diseño

1. Diseño debe ser rastreable hasta el modelo de req.
2. Considerar la arquitectura del sistema a crear.
3. Misma importancia entre el diseño de datos y funciones.
4. Interfaces diseñadas con cuidado.
5. La interfaz debe ajustarse a las necesidades del usuario. Fácil de usar.
6. Los componentes deben ser funcionalmente independientes.
7. Bajo acoplamiento de componentes.
8. Las rep. de diseño deben ser entendibles.
9. Debe hacerse iterativamente.
10. No se debe impedir la metodología ágil.

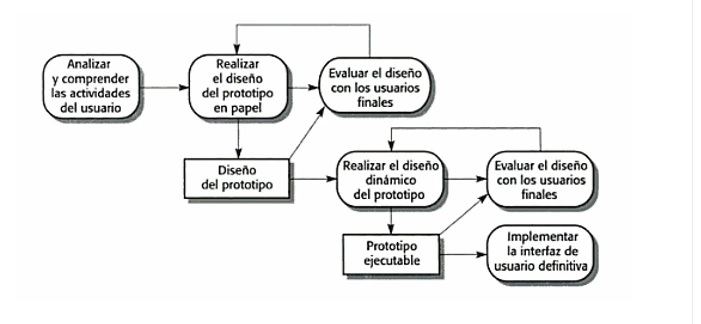
# Clase 5: Diseño de la interfaz de usuario

Se refiere a diseñar la experiencia de usuario de la interacción con el sistema. Multidisciplinar, se aplica diseño gráfico e industrial para que los usuarios aprendan lo más rápido posible el funcionamiento del software.

* Crea un medio de comunicación entre hombre y máquina.
* Con un conjunto de principios se crea el formato de pantalla.
* Es necesario estudiar las preferencias para adaptarnos a las personas. Diversidad de preferencias.
* Si la interfaz es tosca entonces las personas cometen errores o no usan el sistema.
* Debe dar un rápido acceso al contenido, sin pérdida de compresión mientras se desplaza a través de la información.

#### Formas de consumir

* Tecnologías que deben adaptarse al usuario: Hipertexto, sonido, video, etc.
* Conf. de hardware: Teclado, mouse, etc.
* Dispositivos: PC, celulares.



### Proceso iterativo

#### Análisis y modelado

* + Definición de objetos y acciones de interfaz con lo analizado.

#### Diseño de interfaz

* + Definir (y modelar) eventos que harán que cambie el estado de la interfaz.

#### Construcción de la interfaz

* + Ilustrar cómo vería el usuario final cada estado de la interfaz.

#### Validación

* + Chequear si el usuario interpreta como queremos nosotros el estado del sistema a partir de la interfaz.

### UX

Similar a UI design, se refiere a un conjunto de métodos aplicados durante el diseño para brindar una buena experiencia de usuario y satisfacer al cliente.

#### Tipos

1. Diseño de interacción: Diseñar la interaccion usuario-producto para que sea agradable.
2. Diseño visual: Se fija como se siente navegar por la aplicación, que transmite y si es eficiente. Maneja color, equilibrio, contraste, etc.
3. Investigación del usuario: Consiste en determinar que quieren y necesitan.
4. Arquitectura de la información: Se utiliza para estructurar y etiquetar el contenido para que los usuarios lo encuentren fácil.



#### Investigación de usuarios

Para buscar info se busca de:

* Encuestas
* Información de ventas
* Información mercadotecnia
* Información de charlas de apoyo al usuario.

Información para crear un perfil de usuario:

1. Rango etario.
2. Etnia.
3. Género.
4. Experiencia.
5. Nivel de ingresos.
6. Idioma.
7. Nivel de estudios.
8. Localización.
9. Ocupación o profesión.
10. Religión.

#### Relevamiento de la tarea

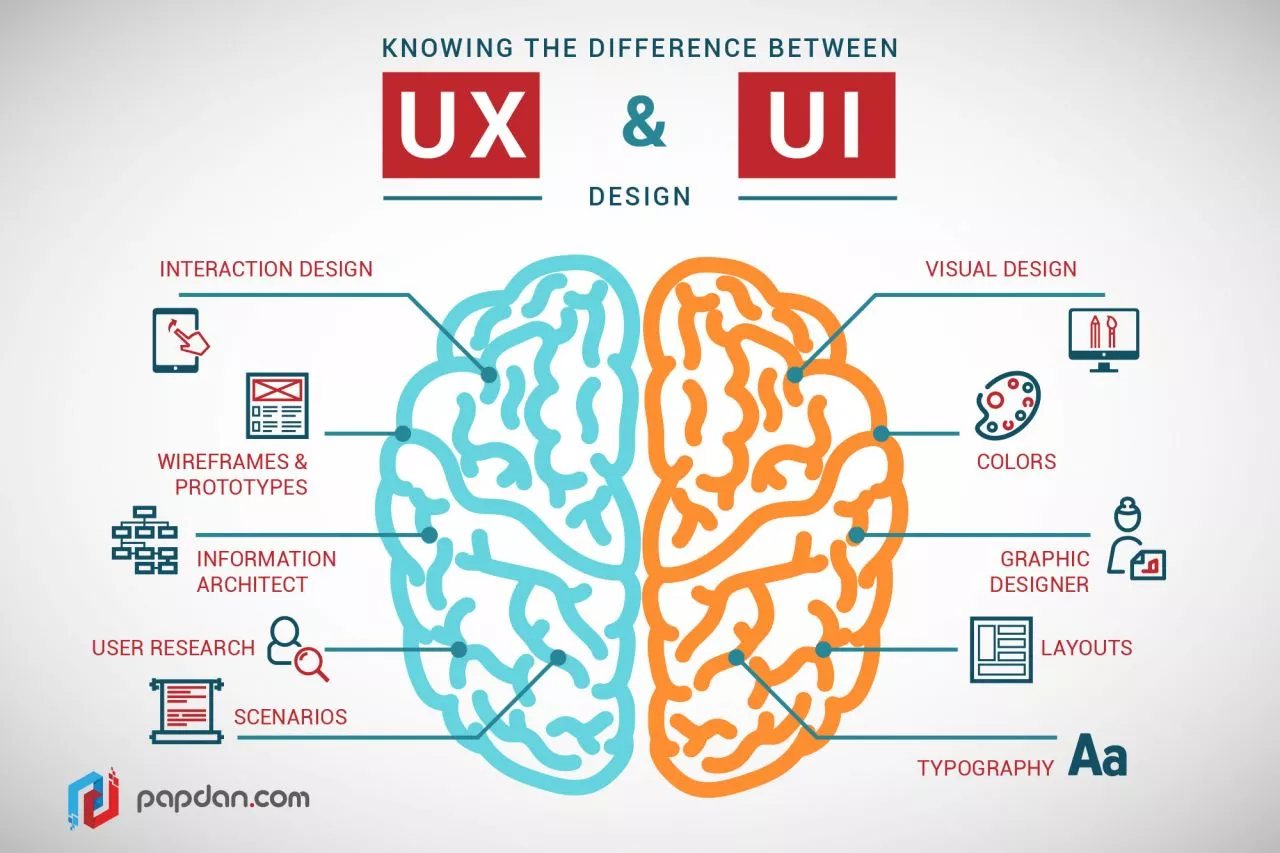
* Estudiar las partes implicadas en el sistema
* Revisión de las competencias del producto
* Recorridos del usuario dentro del sistema.
* ¿Qué trabajo hará el usuario según las circunstancias?
* ¿Que dominio de problema manipulará el usuario para hacer su labor?
* ¿Cual es la secuencia y jerarquía de tareas

Esto se hace en paralelo a la especificación de requerimientos.

### UI vs UX

En general diríamos que el UX diseña la interacción y cómo se percibe el sistema al usarlo fijándose en usabilidad, accesibilidad, funcionalidad, etc.

Y la UI como se ve los elementos con los que participa el usuario, fijándose que se agradable e intuitivo.



## Aspectos de diseño de interfaz

### Reglas doradas del diseño

1. Dar control al usuario
   1. Sin acciones innecesarias.
   2. Flexible.
   3. Acciones de interrumpir o deshacer.
   4. Ofrecer formas para simplificar la interacción a medida que el usuario mejora.
   5. Ocultar detalles técnicos.
   6. Interacción directa con los objetos que aparecen en pantalla.
2. Reducir la carga de memoria del usuario
   1. Reducir demanda a corto plazo.
   2. Definir valores por defecto con sentido.
   3. Definir accesos directos intuitivos.
   4. El formato visual debe basarse en una metáfora de la realidad.
   5. Desglosar la información progresivamente.
3. Lograr consistencia
   1. Permitir que la tarea actual del usuario se incluya en el contexto.
   2. Debe poder determinar de dónde viene y a dónde ir.
   3. Mantener consistencia entre familia de apps.
   4. Misma reglas de diseños para las interacciones.
   5. Mantener modelos prácticos para el usuario.

Factores humanos (4ta regla teóricamente): La memoria, capacitación, percepción visual etc afecta al diseño, un usuario casual necesita ser guiado y uno experimentado necesita agilidad.

#### Usabilidad

Se refiere a cuando la arquitectura de la interfaz se ajusta a las necesidades de las personas que la usan.

Para chequear la usabilidad un sistema le preguntamos a los usuarios:

1. ¿Es utilizable sin enseñanza?
2. ¿Un usuario preparado puede trabajar con eficiencia?
3. ¿Se vuelve más flexible a medida que los usuarios conocen los mecanismos?
4. ¿Está adaptado el sistema al ambiente físico y social donde se usará?
5. ¿El usuario está al tanto del estado del sistema y sabe dónde está?
6. ¿La estructura tiene lógica?
7. ¿Los mecanismos, iconos, y procedimientos son consistentes?
8. ¿Se prevén errores con la interfaz y ayuda a corregirlos?
9. ¿La interfaz es tolerante a los errores?
10. ¿Es sencilla la interacción?

#### Principios de Nielsen

1. Diálogo simple y natural: Forma en que la interacción con el usuario debe llevarse a cabo.
2. Lenguaje del usuario: Emplear en el sistema un lenguaje familiar para el usuario, usar el lenguaje del usuario.
3. Minimizar el uso de la memoria del usuario: Evitar que el usuario esfuerce su memoria para interactuar con el sistema.
4. Consistencia: Que no existan ambigüedades en el aspecto visual ni tecnológico en el diálogo o en el comportamiento del sistema.
5. Feedback: Es una respuesta gráfica o textual en la pantalla, frente a una acción del usuario. El sistema debe mantener al usuario informado de lo que está sucediendo.
6. Salidas evidentes: Que el usuario tenga a su alcance de forma identificable y accesible una opción de salida.
7. Mensajes de error: Información que brinda el sistema ante la presencia de un error. De qué forma se ayuda al sistema para que salga de la situación en la que se encuentra.
8. Prevención de errores: Evitar que el usuario llegue a una instancia de error.
9. Atajos: La interfaz debería proveer de alternativas de manejo para que resulte cómodo y amigable tanto para usuarios novatos como para usuarios experimentados.
10. Ayudas: Componentes de asistencia para el usuario. Un mal diseño de las ayudas puede llegar a entorpecer y dificultar la usabilidad.

## Tipos de interfaces

Todo esto es chamuyo pero bueno, tenes las terminales, las interfaces de comandos gráficas, interfaz de selección de menú, ventanas formularios, iconos y menús, interfaces de manipulación directa (cajero), manipulación directa táctil, voz, preguntas y respuestas IA.

Estas interfaces pueden ser aplicadas a distintos dispositivos (responsive).

## Presentación de info

Se refiere a cómo la información es representada en pantalla, que depende de quien lo usa, además se fija si utiliza números o imágenes, información precisa o relación entre valores, y si el cambio debe ser inmediato o no.

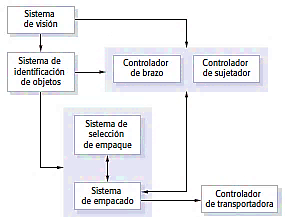
Sobre colores: No usar más de 4 o 5 colores en una ventana y no más de 7 en toda la interfaz, pickear colores complementarios y no asociarlos a un significado.

# Clase 6: Diseño arquitectónico

## Diseño arquitectónico

* Define la relación entre los elementos de la estructura.
* Es un proceso que se usa para identificar los subsistemas dentro del sistema y definir cómo se controlan y comunican entre sí.
* Los grandes sistemas están divididos en subsistemas que proporcionan un conjunto de servicios relacionados.

Ejemplo



### Arquitectura y req. no funcionales

#### Protección

* + Las operaciones relacionadas a la protección deberían estar en un solo pocos subsistemas para minimizar costos y problemas de validación.

### Disponibilidad

* + Debería haber componentes redundantes para que sea posible reemplazar sin parar el sistema.

### Mantenibilidad

* + Los componentes deberían ser autocontenidos de grano fino para fácil modificación.

### Organización del sistema

Hablamos de la estrategia básica usada para estructurar el sistema.

Los subsistemas de un sistema deben intercambiar datos de forma efectiva.

* Base de datos central.
* Cada subsistema con su información y se la intercambian entre sí.

Estilos organizacionales (Patrones) como repositorio, cliente-servidor, combinaciones, etc.

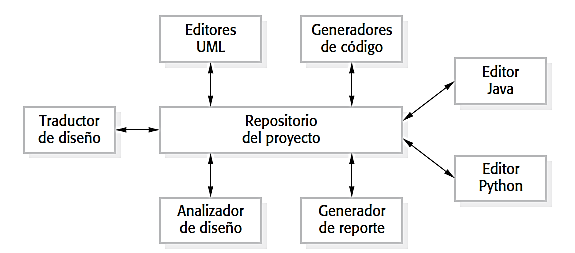
#### Subsistema

* + Parte de un sistema con la interfaz definida para comunicarse con otros subsistemas (acá dicen módulo pero abajo lo redefinen como otra cosa).

#### Módulo

* + Componente de un subsistema que da uno o más servicios a otros módulos, y recibe servicios de otros módulos. No suelen ser un sistema independiente.

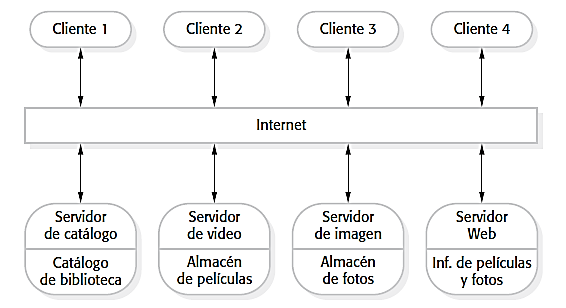
#### Patrón: Repositorio



Subsistemas organizados alrededor de una base de datos compartida.

| Ventajas | Desventajas |
| --- | --- |
| 1. Es eficiente para compartir datos ya que no se transmiten entre subsistemas. 2. Los subsistemas que producen datos no deben saber para qué son. 3. Las actividades de backup, protección y control están centralizadas. 4. El modelo compartido es visible a través del esquema del repositorio. Las herramientas se integran directamente ya que son compatibles con el modelo de datos. | 1. Los subsistemas deben estar acorde a los modelos de datos del repo afectando el rendimiento. 2. Cambiar el modelo y migrar puede hacerse imposible cuando se generaron muchos datos. 3. Los subsistemas podrían tener distintos req. de protección (en este modelo tienen siempre los mismos). 4. Difícil de distribuir entre máquinas. |

#### Patrón: Cliente Servidor



Se organiza como un conjunto de servicios con servidores asociados y unos clientes que los utilizan.

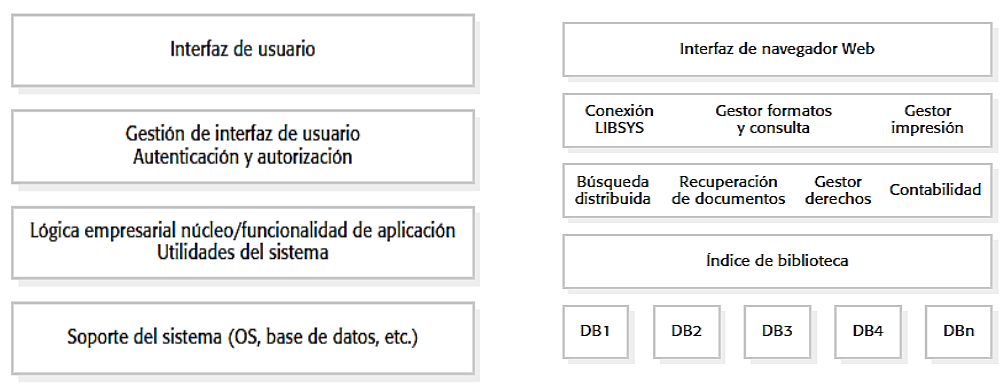
Compuesto de: Servidores que ofrecen servicios, clientes que los llaman, red que permite a los clientes acceder a los servicios.

Los clientes conocen el nombre del servidor y lo que brindan, pero no al revés.

#### Patrón: Arq. en capas

Cada capa en este sistema es un conjunto de servicios con capas adyacentes.

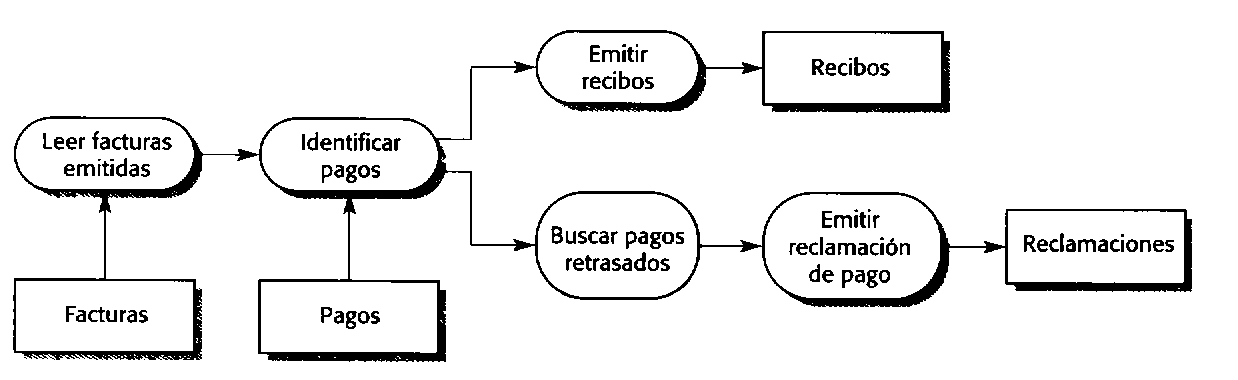
| Ventajas | Desventajas |
| --- | --- |
| 1. Soporta el desarrollo incremental. 2. Es portable y resistente a cambios. 3. Se puede reemplazar las capas si no cambia la interfaz, o adaptarse con un capa de adaptación. 4. Útil para sistemas multiplataforma, las capas más internas son dependientes de la plataforma. | 1. Difícil de estructurar. 2. Las capas internas proporcionan servicio req. por todos los niveles. 3. Los servicios requeridos por el usuario podrían tener que pasar múltiples niveles de capas. 4. Un servicio solicitado por la capa superior podrá ser interpretado varias veces en dif. capas. |



### Descomposición modular

Dividimos a los subsistemas que organizamos en módulos, utilizamos criterios similares pero a menor escala.

* Descomposición orientada a flujo de funciones:
  + Conjunto de módulos funcionales. Los datos fluyen de una función a otra transformándose a lo largo de la secuencia o en paralelo.

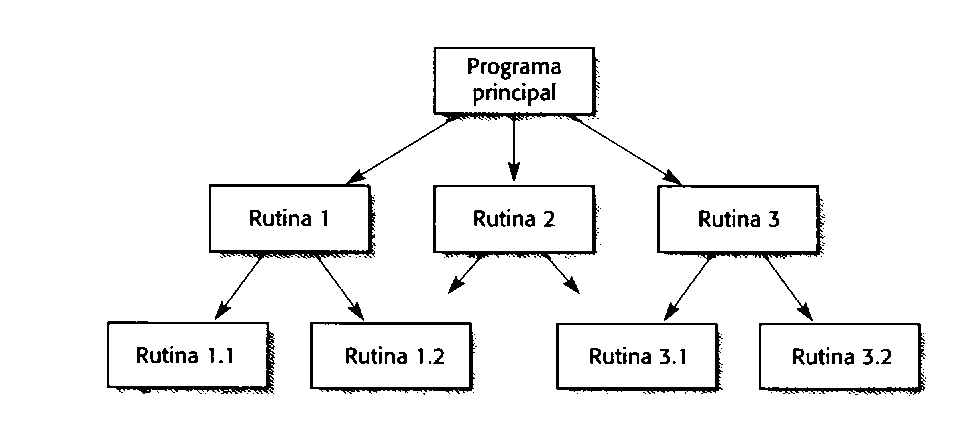
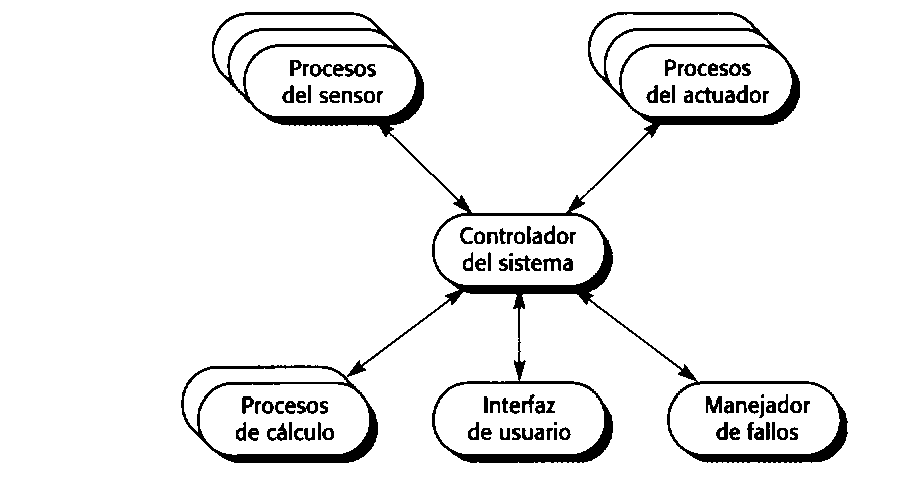


* Descomposición orientada a objetos.
  + Se estructura el sistema como un conjunto de objetos débilmente acoplados con interfaces bien definidas, el ejemplo es un UML.

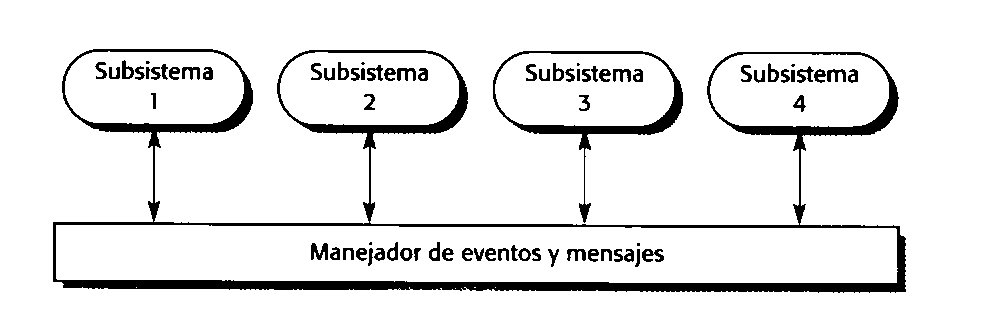
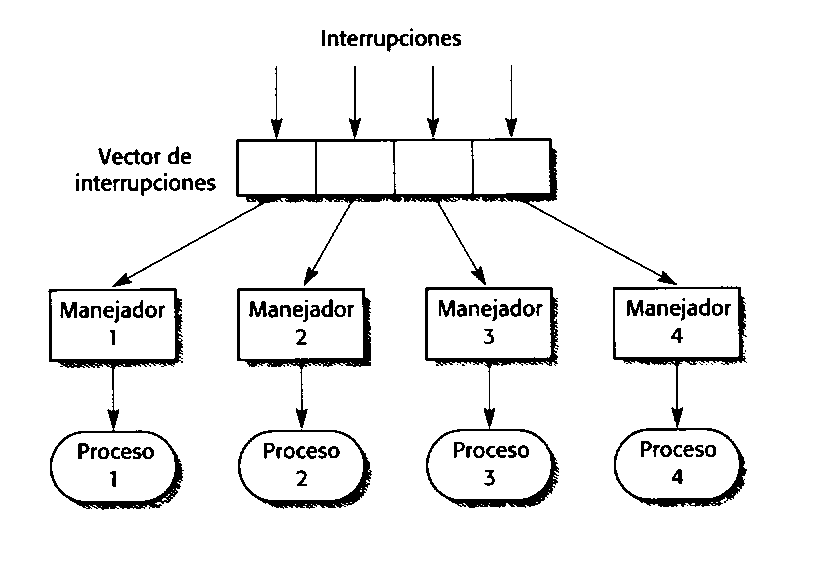
### Modelos de Control

Control para que se entreguen los servicios en donde es correcto y cuando es preciso.

#### Control centralizado

* + Un subsistema se dedica a iniciar y detener otros, ejecutandolos en secuencial o en paralelo.
  + Modelo de llamada y retorno:
    - Modelo que utiliza subrutinas descendentes.
    - Aplicables a modelos secuenciales.
    - 
  + Modelo de gestor
    - Un gestor controla el inicio y parada coordinado con el resto de los procesos.
    - Aplicable a modelos concurrentes.
    - 

#### Basado en eventos

* + Cada subsistema responde a eventos externos al subsistema.
  + Eventos:
    - Señales binarias.
    - Un valor dentro de un rango.
    - Una entrada de comando.
    - Selección de menú.
  + Modelos de transmisión (BroadCast)
    - Un evento se transmite a todos los subsistemas y responderán aquellos que sepan manejarlo.
    - 
  + Modelo dirigido por interrupciones.
    - Para sistemas en tiempo real, las interrupciones son detectadas por el manejador de interrupciones y redirige al componente que lo maneja.
    - 

## Arquitectura de los Sistemas Distribuidos

Sistemas donde el procesamiento se distribuye entre varias computadoras, por ejemplo cliente-servidor.

#### Características

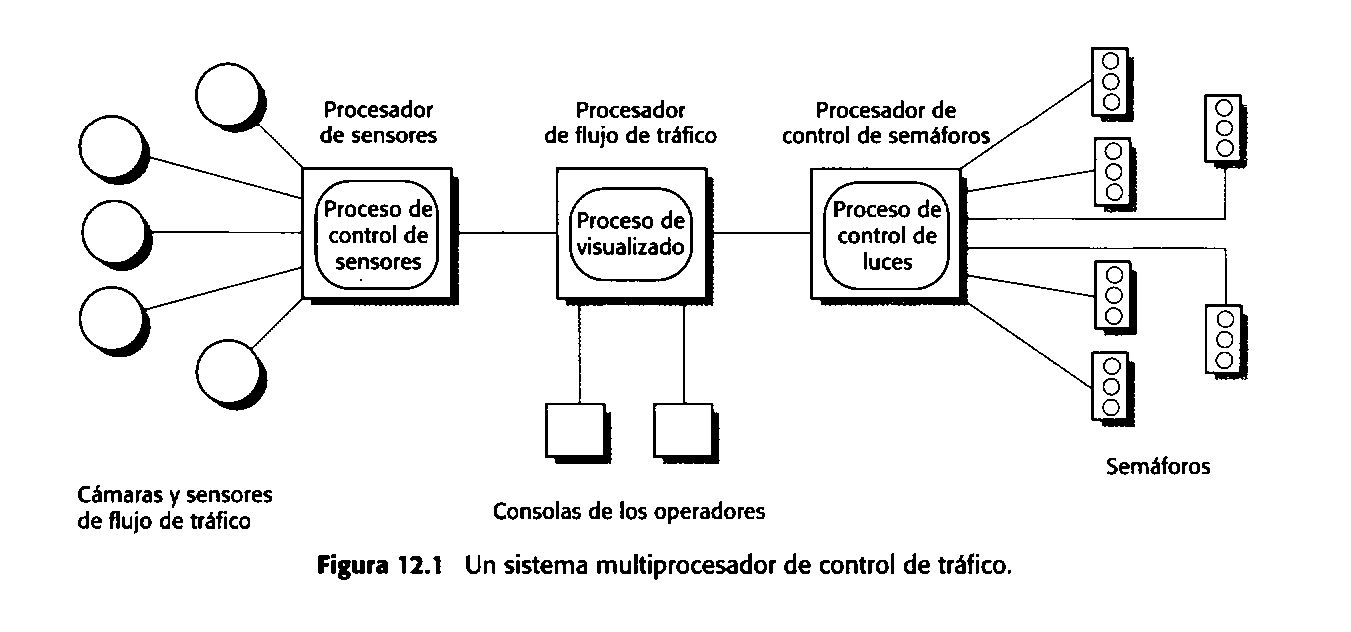
1. Comparten recursos
2. Apertura, son sistemas abiertos diseñados con protocolos estándar para simplificar la combinación de recursos.
3. Concurrencia, varios procesos pueden operar al mismo tiempo en distintas computadoras.
4. Escalabilidad, capacidad de añadir recursos para suplir demandas.
5. Tolerancia a fallos, al haber muchas computadoras y poder reproducir la información, los sistemas toleran más los fallos de hardware y software.

#### Desventajas

* Complejidad: Son más complejos y hay que tener en cuenta los problemas de comunicación y sincronización.
* Seguridad: El tráfico entre computadoras podría ser intervenido.
* Manejabilidad: Las computadoras pueden ser de diferentes tipos y diferentes S.O, generando dificultades de gestión.
* Impredecibilidad: El tiempo de respuesta depende de la carga del sistema y el estado de la red.

#### Arq. Multiprocesador

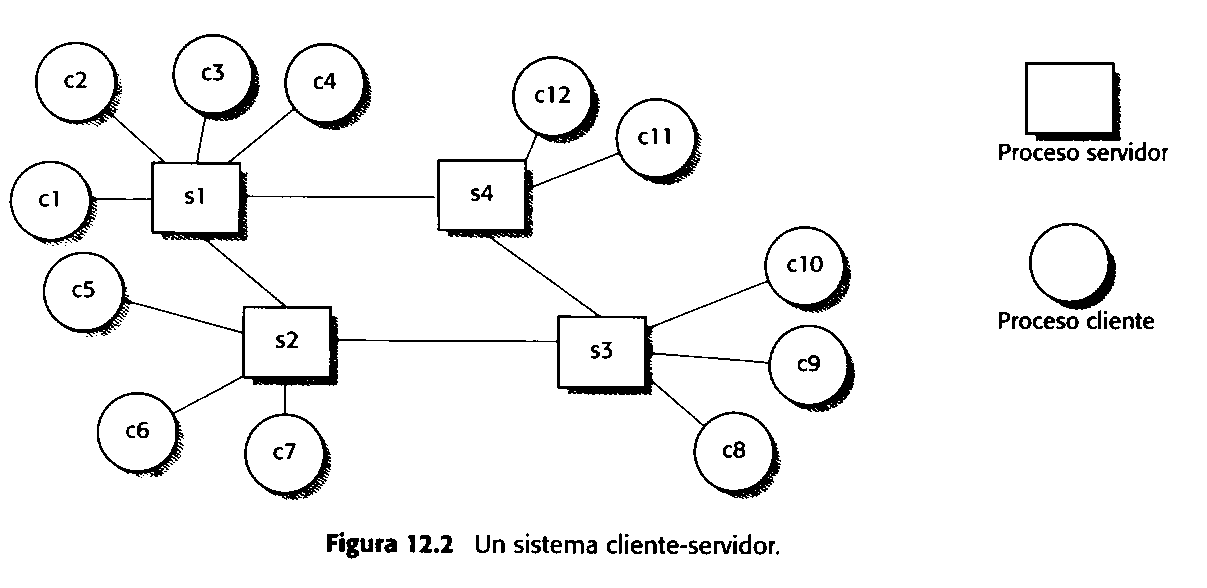
La arquitectura multiprocesador consiste en un sistema de software donde múltiples procesos pueden ejecutarse en diferentes procesadores, asignados de manera predeterminada o por un dispatcher. Es típica en sistemas grandes de tiempo real que recopilan datos, toman decisiones y envían señales para modificar el entorno.

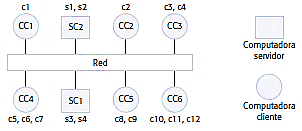


#### Arq. Cliente-Servidor

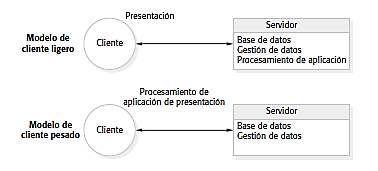
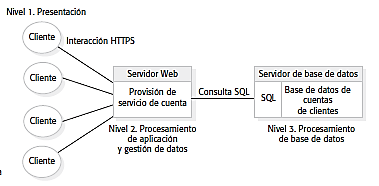
App como conjunto de servicios proporcionado por servidores y un clientes que los usan.

Clientes y servidores son procesos diferentes, los servidores atiende varios clientes y un servidor puede brindar varios servicios. Los clientes no se conocen entre sí.





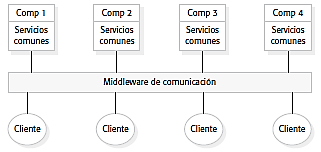
Se divide en:

* Dos niveles
  + Cliente ligero: Procesamiento y gestión de datos en el servidor.
  + Cliente pesado: Cliente implementa la lógica de aplicación y el servidor gestiona los datos.
  + 
* Multinivel:
  + La presentación, procesamiento y gestión de datos son procesos separados con posibilidad de ejecutarlo en procesadores diferentes
  + 

#### Arquitectura de componentes distribuidos

Conjunto de objetos con interfaz para darse servicios entre ellos. No hay distinción tajante entre cliente y server.

Los componentes se pueden distribuir entre máquinas usando un middleware.



##### **Computación distribuida interorganizacional**

La Organización tiene varios servidores y reparte la carga entre ellos.

Peer-to-peer: Descentralizado cualquier cálculo se puede hacer en cualquier nodo. Se aprovecha la potencia computacional y almacenamiento a través de una red.

* Descentralizado: Cada nodo rutea los paquetes a sus vecinos hasta hallar el destino.
* Semi centralizado: El servidor ayuda a conectar los nodos y coordina resultados.

Orientada a servicios:

* Un servicio es una representación independiente de un recurso computacional o de información, accesible para múltiples programas y organizaciones, permitiendo la construcción de aplicaciones mediante acoplamientos débiles en arquitecturas de servicios web.
* Un proveedor pública su servicio en un registro, y un solicitante lo integra en su aplicación mediante código para invocarlo y procesar sus resultados.

## Codificación

Una vez definido el diseño, se debe programar su implementación, tarea compleja por varias razones:

* Los diseñadores pueden omitir detalles de la plataforma
* Las estructuras diagramáticas no siempre se traducen fácilmente a código
* Es crucial la claridad del código para la comprensión ajena
* Se debe favorecer la reutilización del diseño en el código.

Buenas practicas:

1. **Separación de entrada y salida:** Es recomendable ubicarlas en componentes separados del código principal debido a su dificultad usual para probarlas adecuadamente.
2. **Uso de pseudocódigo:** Antes de comenzar la codificación, elaborar pseudocódigo ayuda a adaptar el diseño al lenguaje elegido y a planificar con mayor precisión.
3. **Revisión y reescritura exhaustiva:** Es preferible realizar borradores del código, revisarlos y reescribirlos cuantas veces sea necesario, en lugar de aplicar parches sobre errores.
4. **Reutilización de componentes:** Existen dos formas de reutilización:
   * **Productiva:** Crear componentes diseñados para ser reutilizados en otras aplicaciones.
   * **Consumidora:** Utilizar componentes desarrollados originalmente para otros proyectos.

**Documentación interna:** Es breve y está escrita para programadores. Contiene detalles sobre algoritmos, estructuras de control y flujos de ejecución que ayudan a entender el código fuente.

**Documentación externa:** Está diseñada para ser entendida por personas que pueden no ver el código directamente, como diseñadores evaluando modificaciones o mejoras.

# Clase 7: Gestión de proyecto(métricas y estimaciones)

## Métricas

Clave tecnológica para el desarrollo.

#### Objetivos

* **Entender** que ocurre durante el desarrollo y mantenimiento.
* **Controlar** qué es lo que ocurre en nuestro proyecto.
* **Mejorar** nuestro proyecto y producto.
* **Evaluar** la calidad.

#### Definiciones

* Medida: Valor cuantitativo de algún atributo de mi proyecto, por ejemplo qué cantidad de personas requiere mi proyecto.
* Medición: Acto de medir y determinar una medida.
* Métrica: Utilizó para para lograr esta medición y conseguir una medida, la métrica se refiere a la medida cuantitativa de un atributo específico de un proceso o de un producto del software *(¿Cuál sería entonces la diferencia con medida?).*
* Indicador: Combinación de métricas, usado para definir cuando ocurre algo. Por ejemplo si medimos la temperatura y se pasa cierto valor entonces hay problemas.

### Métricas de proyecto

Propósitos tácticos

Uso:

1. Ajustes en el calendario y evitar demoras
2. Valorar el estado de un proyecto en marcha
3. Rastrear riesgos
4. Descubrir áreas de problemas
5. Ajustar flujo de trabajo/tareas
6. Evaluar habilidad del equipo.

### Métricas de proceso

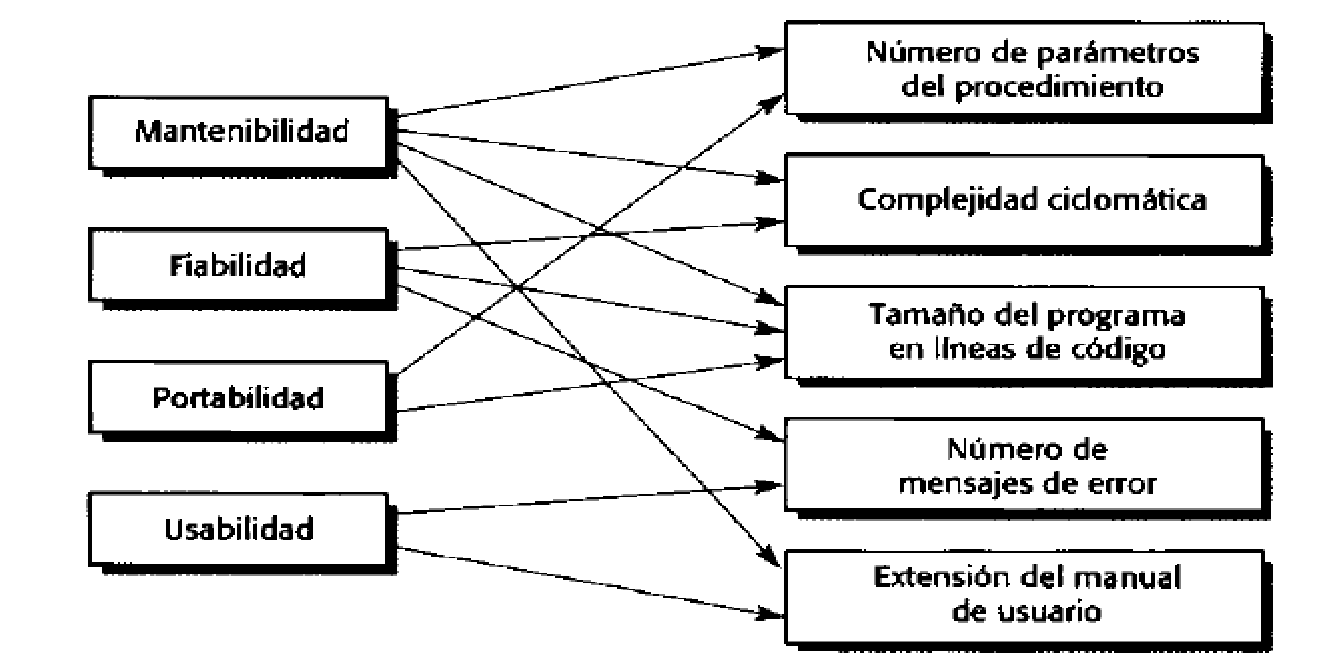
Propósitos estratégicos

* Recopilación a través de los proyectos durante un periodo de tiempo.
* Proporcionar un conjunto de indicadores para mejorar el proceso.​​

Uso:

1. Sentido común y sensibilidad organizacional.
2. Retroalimentación.
3. No usar métricas para valorar a los individuos.
4. Establecer metas y métricas claras.
5. No excluir métricas.

¿Cómo medimos las diferentes cosas?



Cada uno de los atributos externos (que también son requerimientos no funcionales) que queremos medir de nuestro proyecto deben estar conectados con atributos internos para poder ser medidos.

### Métricas de producto dinámicas y estáticas

#### Dinámicas

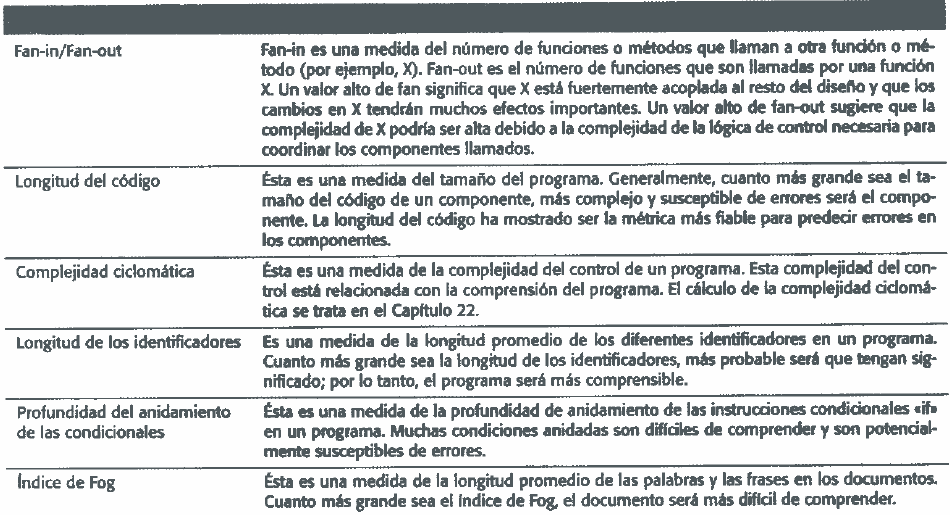
Hechas durante la ejecución del programa, ayudan a valorar la eficiencia y fiabilidad de un programa.

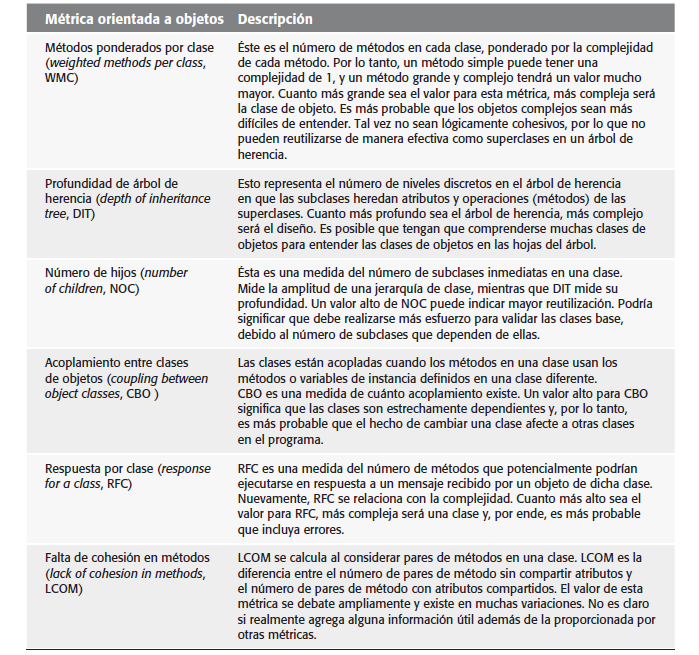
Se relacionan con las características de calidad del software

#### Estáticas

Hechas en base a representaciones del sistema, ayudan a valorar la complejidad, compresibilidad, y mantenibilidad.

Se relacionan de manera indirecta con los atributos de calidad del software





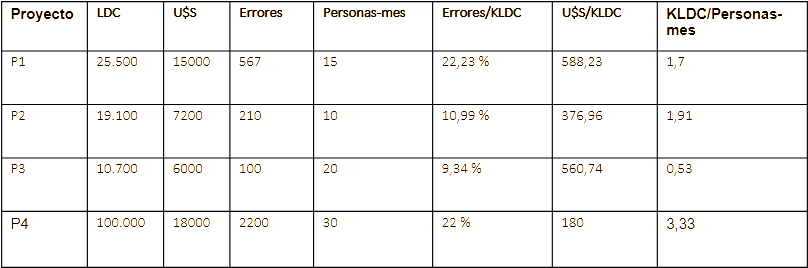
LDC: líneas de código, mide el tamaño de un producto. Medida muy polémica, el resultado verdadero se consigue postmortem. Podemos sacar cuanto tiempo va a tomar, cuanta gente usará, etc.

Utilidad de las métricas postmortem: Se reutilizan para futuros proyectos, o mantenimiento del sistema.

Métricas orientadas al tamaño

Se puede obtener :

* Productividad: relación entre KLDC / Persona mes
* Calidad: relación entre Errores / KLDC
* Costo: relación entre $ / KLDC

Ejemplo:   


Sirven para evaluar porque hubo más calidad, o menos errores, etc.

#### Métrica de Punto función

Mide la cantidad de funcionalidad de un sistema descrito en una especificación.

Factor de Ponderación, es subjetivo y está dado por la organización/equipo

Son valores de ajuste de la complejidad según las preguntas:  
 1. ¿Requiere el sistema copias de seguridad y de recuperación fiables?

2.¿Se requiere comunicación de datos?

3. Existen funciones de procesamiento distribuido?

4. ¿Es crítico el rendimiento?

5. ¿Se ejecuta el sistema en un entorno operativo existente y fuertemente utilizado?

6. ¿Requiere el sistema entrada de datos interactiva?

7. ¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples pantallas u operaciones?

8. ¿Se actualizan los archivos maestros de forma interactiva?

9. ¿Son complejas las entradas, las salidas, los archivos o las peticiones?

10. ¿Es complejo el procesamiento interno?

11. ¿Se ha diseñado el código para ser reutilizable?

12. ¿Están incluidas en el diseño la conversión y la instalación?

13. ¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?

14. ¿Se ha diseñado la aplicación para facilitar los cambios y para ser fácilmente utilizada por el usuario?

Métricas derivadas:

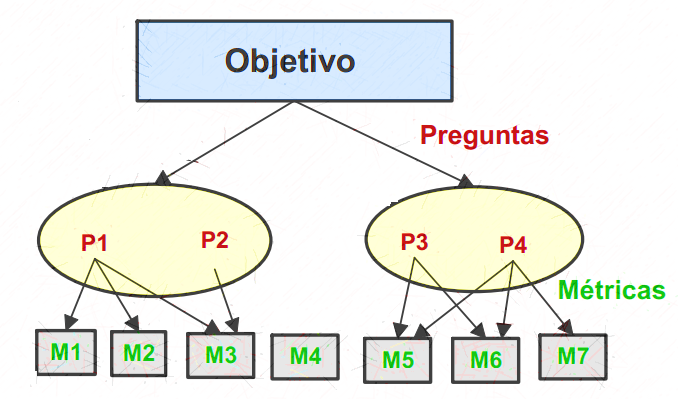
* Productividad: relación entre PF y Persona\_mes
* Calidad: relación entre Errores y PF
* Costo: relación entre $ y PF

#### GQM

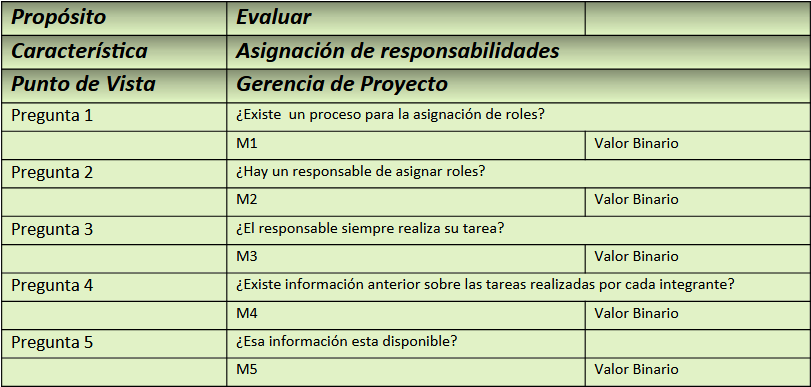
Dicho método está orientado a lograr una métrica que “mida” cierto objetivo. El mismo nos permite mejorar la calidad de nuestro proyecto. Muy amplia.

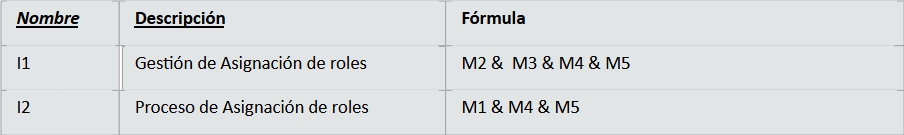
Estructura :

* Nivel Conceptual (Goal / Objetivo).
  + Se define un objetivo (en nuestro caso, para el proyecto).
* Nivel Operativo (Question / Pregunta).
  + Se refina un conjunto de preguntas a partir del objetivo, con el propósito de verificar su cumplimiento.
* Nivel Cuantitativo (Metric / Métrica).
  + Se asocia un conjunto de métricas para cada pregunta, de modo de responder a cada una de un modo cuantitativo.



Ejemplo:



Definimos si la meta se cumple en base a si los indicadores muestran valores verdaderos:  




* Es útil para decidir qué medir.
* Debe estar orientado a metas.
* Es flexible.

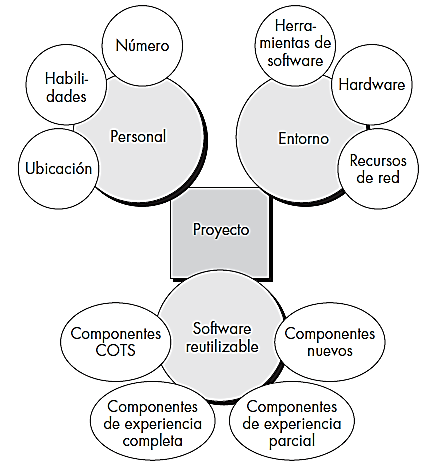
## Estimaciones

Valor aproximado, la diferencia con las métricas en sí que es aproximado, las métricas no lo son.

Hay que tener cuidado al usarlas. Normalmente se calcula con diferentes fórmulas de estimación y se hace un resultado más certero (no es una ciencia exacta y además los cambios en requerimientos pueden cambiarlas).

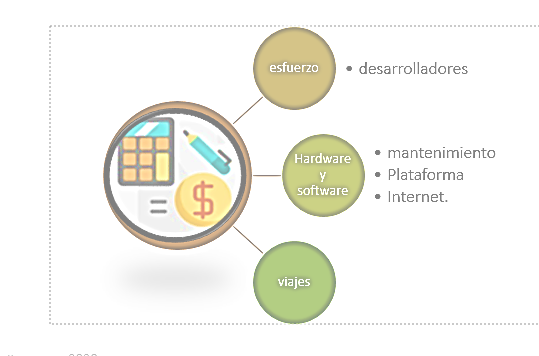
### Estimaciones de recursos

Cosas a tener en cuenta:

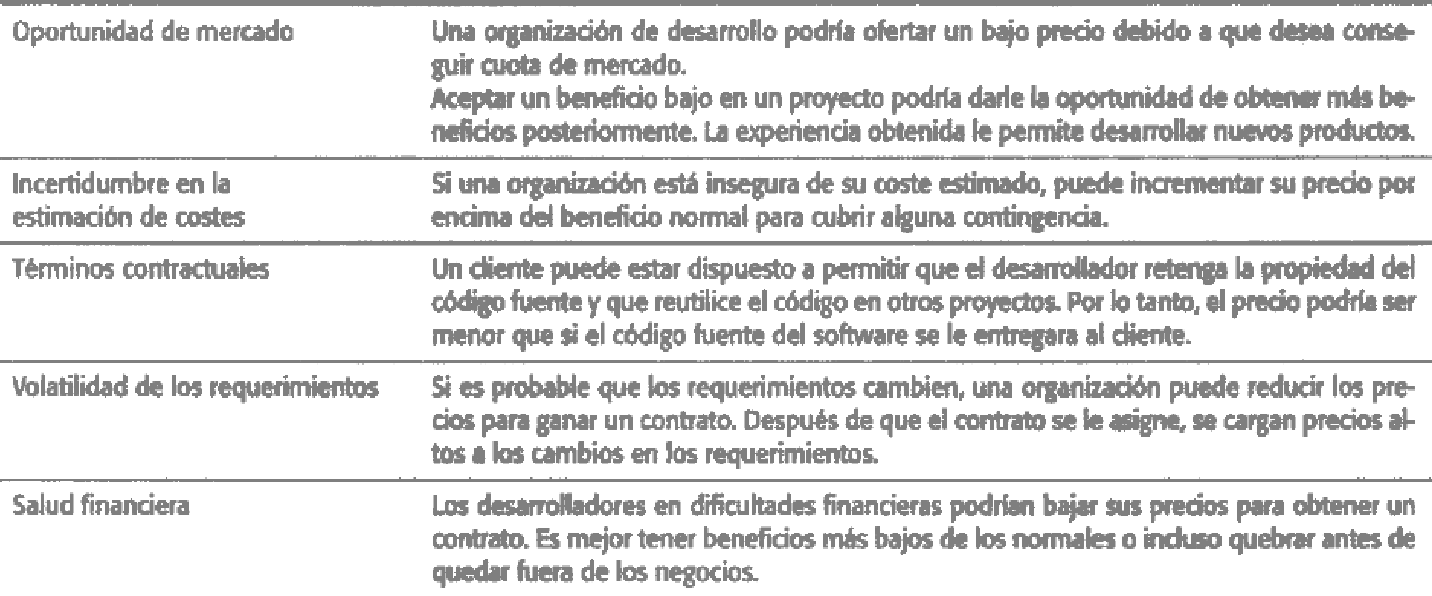


Debo ser muy preciso en la descripción de los recursos requeridos.

### Estimaciones de costos



Cosas a tener en cuenta a la hora de estimar un presupuesto, no es tan lineal como calcular el costo y sumarle un 20%:



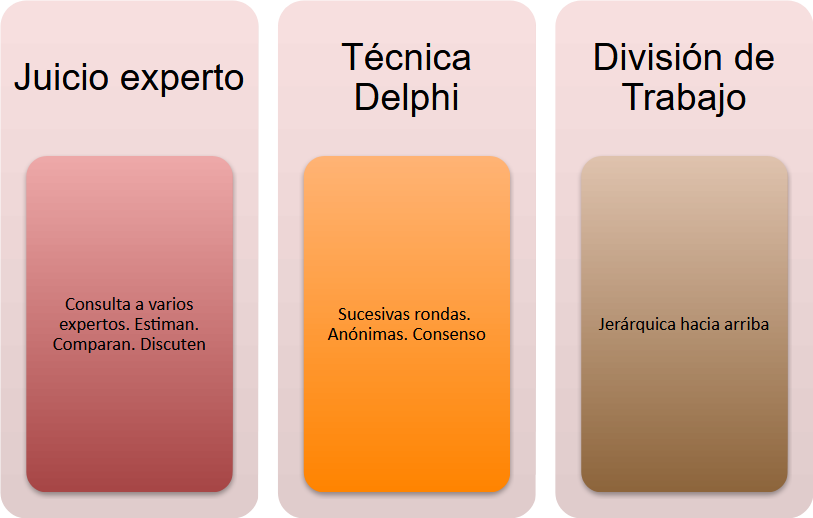
Debe pensarse en :

* Los intereses de la empresa.
* Los riesgos.
* El tipo de contrato.

### Estimaciones de tiempo

Lo de planeación temporal, ya lo vimos. Camino crítico.

### Técnicas de estimación



El juicio de expertos se utiliza para evaluar insumos, detalles técnicos y de gestión. Aunque proporciona una perspectiva externa valiosa, es subjetivo y necesita pautas. Es especialmente útil cuando hay mucho en juego, en proyectos de alto impacto, antes de la planificación, para la gestión de costos y para la gestión de riesgos.

Delphi vs Juicio experto: En Delphi no se conocen los expertos.

### Planning poker

* Participantes: Todos los involucrados en el desarrollo.
* Objetivo: Estimar cada historia de usuario para un Sprint.
* Por qué "Planning Poker": Se utilizan cartas numeradas, normalmente con la serie de Fibonacci.

Pasos:

1. Reunión del equipo: Todo el equipo se reúne, cada miembro con un juego de cartas numeradas con la serie de Fibonacci.
2. Lectura de la historia: El Product Owner lee una historia de usuario al equipo.
3. Resolución de dudas: Se aclaran dudas en este momento.
4. Selección de cartas: Cada miembro selecciona una carta en privado.
5. Revelación de cartas: Todos muestran sus cartas simultáneamente.
6. Comparación y justificación: Se comparan los resultados y se justifican. Si hay unanimidad, se asigna el peso elegido a la historia. Si no, se repite el proceso hasta llegar a un consenso.
7. Repetición: El proceso se repite con cada historia de usuario en el backlog del producto.

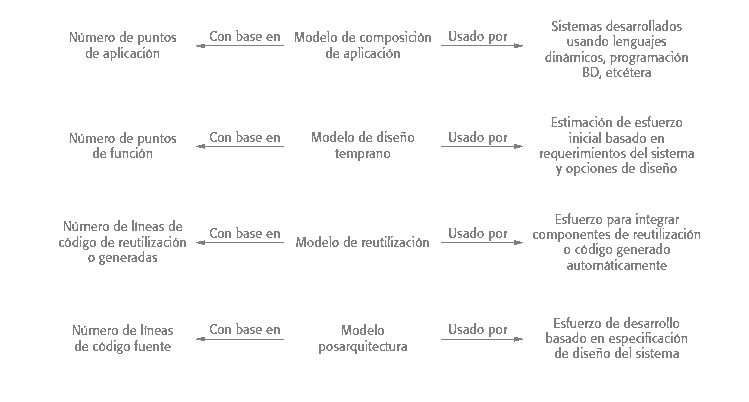
### Modelos empíricos de estimación

Utilizan fórmulas derivadas empíricamente para predecir costos o esfuerzo requerido en el proyecto.

#### COCOMO II

Modelo empírico que derivó de recopilar datos a partir de un gran número de proyectos de software.

Las fórmulas vinculan el tamaño del sistema y los factores del producto, proyecto y equipo con el esfuerzo para desarrollar el sistema.



# Teoría 8: Caja negra y caja blanca

Los errores aparecen en todas las etapas, antes incluso de las pruebas.

Que falle un sistema es que no haga lo que se requiere.

Posibles razones:

* Especificación errónea.
* Requerimientos imposibles con las estructuras previstas.
* Defectos en el diseño del sistema.
* Defectos en el diseño del programa.
* Defectos en código.

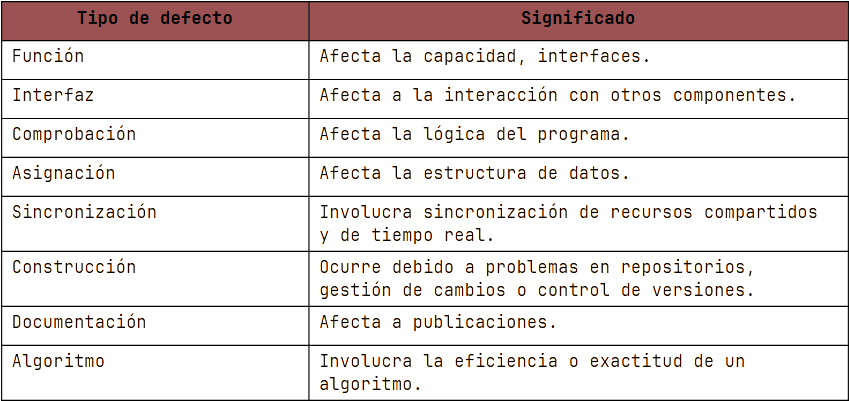
#### Tipos de defectos

1. **Algorítmicos**: No inicializar variables.
2. **De sintaxis**: Confundir un 0 por una O.
3. **De precisión**: Fórmulas mal implementadas.
4. **De documentación**: Documentación no acorde con el software.
5. **De sobrecarga**: El sistema funciona con 100 usuarios, pero no con 110.
6. **De capacidad**: El sistema funciona con ventas menores a 1.000.000.
7. **De coordinación o sincronización**: Fallas en la comunicación entre procesos.
8. **De rendimiento**: Tiempo de respuesta inadecuado.
9. **De recuperación**: No volver a un estado normal tras una falla.
10. **De relación hardware-software**: Incompatibilidad entre componentes.
11. **De estándares**: No cumplir con estándares y procedimientos definidos.

##### **Clasificación ortogonal de defectos**

* Defecto por omisión: Falta algo del código.
* Defecto de cometido: Hacer algo que no va.

Tipos de defectos del Pfleeger:



## Prueba de software

Para descubrir errores antes de entrar en producción y así ahorrar costos.

* Rastreables hasta los requisitos del cliente.
* Planificadas con antelación.
* 80% de los errores se encuentra en el 20% del código a testear.
* Iniciar por lo pequeño y luego ir a lo grande.
* Probar todas las condiciones a nivel de componente.

Un equipo independiente de pruebas es crucial para evitar conflictos de intereses y permitir pruebas concurrentes con la codificación.

## Caja negra

La prueba de caja negra se refiere a las pruebas que se llevan a cabo sobre la interfaz del software. Se centran en los requisitos funcionales del software.

1. Busca errores de tipo:  
   Funciones incorrectas o ausentes
2. Errores de interfaz
3. Errores en estructuras de datos o en accesos a bases de datos externas
4. Errores de rendimiento
5. Errores de inicialización y de terminación
6. Concentra la prueba en el dominio de información: por ejemplo las pruebas de partición equivalente

### Partición equivalente

Tipo de diseño de test que busca separar la posible entrada en particiones equivalentes donde teóricamente lo testeado actuaría igual.

Por ejemplo, si es un **rango** se define una **clase de equivalencia** válida y dos no válidas. Si fuera un **valor específico** lo mismo. Si es un **conjunto**, entonces sería una clase válida y una no válida. Y si la entrada es **lógica**, sería lo mismo que el conjunto.

Es decir, si me tienen que ingresar un número entre 0 y 9999, la válida es obvia y los no válidas serían > 9999 y < 0. Testeamos ahí.

### Análisis de valores límite

Complementario a la partición equivalente, esta dice que en vez de seleccionar cualquier elemento de las clases de equivalencia, elegimos los del extremo. En el ejemplo anterior serían -1 y 10.000.

## Caja blanca

La prueba de caja blanca examina detalladamente los caminos lógicos del software mediante casos de prueba que ejercitan condiciones y bucles específicos.

La prueba de caja blanca verifica detalles procedimentales, asegurando que:

* Se ejerciten todos los caminos y decisiones lógicas.
* Se ejecutan bucles en sus límites.
* Se validan estructuras internas de datos.

### Prueba del camino básico

Representa al flujo como un grafo.

|  |  |
| --- | --- |

La **complejidad ciclomática** es una métrica que cuantifica la complejidad lógica de un programa, definiendo el número de caminos independientes en su conjunto básico y proporcionando un límite superior para las pruebas necesarias para ejecutar cada sentencia al menos una vez. Un camino independiente introduce al menos un nuevo conjunto de sentencias o una nueva condición.

1. V(g)=Cantidad de regiones del grafo ó

2. V(g)= A - N + 2 ó

3. V(g)= P + 1

Hay que hacer las 3 fórmulas, si dan lo mismo esta gucci.

Pasos:

1. Dibujar el grafo de flujo correspondiente.
2. Determinar la complejidad ciclomática.
3. Determinar un conjunto básico de caminos independientes.
4. Preparar los casos de prueba que forzará la ejecución de cada camino del conjunto.
5. Ejecutar cada caso de prueba y comparar los resultados obtenidos con los esperados.

| Procedure Ordenar  (1) do while not eof() begin  (2) leer registros;  (3) if (campo 1 de registro = 0)  (4) then procesar registro  Guardar en buffer;  Incrementar contador;  (5) else if (campo 2 de registro = 0)  (6) then reiniciar contador  (7) else  procesar registro;  Guardar en archivo;  (8) endif  (9) endif  (10) end; |  |
| --- | --- |
| Regiones : 4  Nodos : 10  Aristas : 12  Nodos Predicados : 3  V(G) : A-N+2 = 12-10+2 = 4  V(G) : NP+1 = 3+1 =4  V(G) : R = 4 |

# Teoría 9: Estrategias de prueba

Una estrategia de pruebas de software guía los pasos, tiempos, esfuerzos y recursos necesarios para:

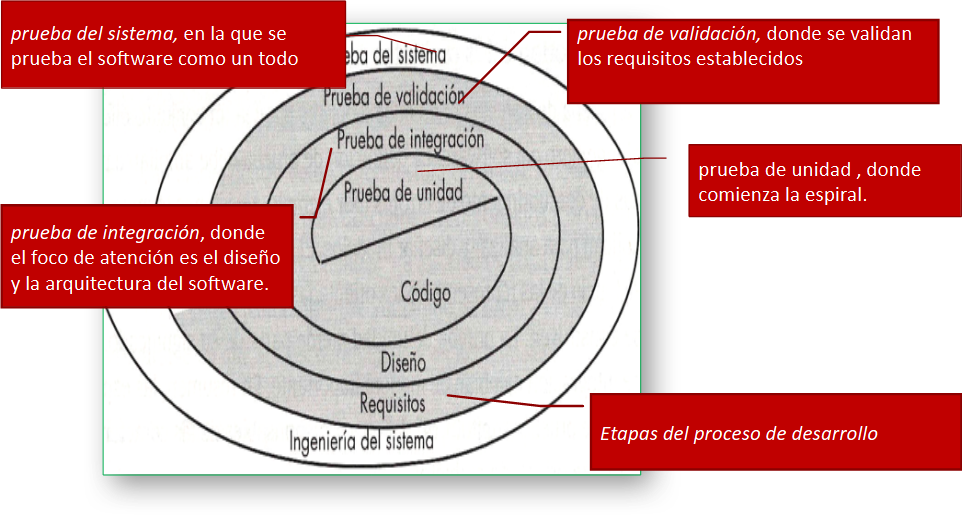
* Planificación de pruebas
* Diseño de casos de prueba
* Ejecución de pruebas
* Evaluación de datos resultantes

La prueba es un conjunto planificado y sistemático de actividades que utiliza técnicas y métodos específicos para diseñar casos de prueba. Una estrategia de pruebas debe incluir pruebas de bajo y alto nivel. Estas actividades forman parte de la Verificación y Validación en el aseguramiento de la calidad del software.

### Validacion y verificacion

La verificación asegura que el software implementa correctamente una función específica, mientras que la validación asegura que el software construido cumple con los requisitos del cliente.

* **Verificación:** "¿Estamos construyendo el producto correctamente?" Comprueba que el software cumpla con su especificación, abarcando tanto los requerimientos funcionales como los no funcionales.
* **Validación:** "¿Estamos construyendo el producto correcto?" Es un proceso general que busca asegurar que el software satisfaga las expectativas del cliente.



## Pruebas de unidad

* Probamos la interfaz de un módulo para asegurarnos el flujo correcto de la información.
* Examinamos estructuras de datos locales.
* Probamos condiciones límites.
* Se ejercitan todos los caminos independientes.

**Errores comunes detectados por las pruebas de unidad:**

* Cálculos incorrectos
  + Aplicación incorrecta de predecesores aritméticos
  + Operaciones mezcladas
  + Inicialización incorrecta
  + Falta de precisión
  + Representación simbólica incorrecta
* Comparaciones erróneas
* Flujos de control inapropiados

**Errores que los casos de prueba deben descubrir:**

* Comparaciones entre diferentes tipos de datos
* Operadores lógicos aplicados incorrectamente
* Expectativas de igualdad con grado de precisión
* Comparación incorrecta de variables
* Terminación inapropiada o inexistente de bucles
* Falla en la salida cuando se encuentre una iteración divergente
* Variables de bucle modificadas inapropiadamente

#### Procedimiento

* **Componente vs Programa independiente**: Un componente no puede funcionar por sí solo y requiere un entorno o soporte adicional.
* **Desarrollo de software para pruebas de unidad**: Es necesario desarrollar software adicional para cada prueba de unidad, para controlar o proteger el componente bajo prueba.
* **Controlador**: Es un programa principal que acepta datos de casos de prueba, los envía al módulo bajo prueba y muestra los resultados.
* **Resguardo**: Sustituye a los módulos subordinados del componente bajo prueba, proporcionando un entorno controlado para las pruebas.
* **Sobrecarga de trabajo**: Desarrollar y mantener controladores y resguardos requiere tiempo y esfuerzo adicionales.
* **Simplicidad de controladores y resguardos**: Si son simples, la carga de trabajo adicional será relativamente pequeña.
* **Alto grado de cohesión en el diseño del módulo**: Un módulo diseñado con alta cohesión simplifica las pruebas de unidad, ya que sus partes están bien relacionadas y organizadas internamente.

## Pruebas de integración

**Propósito de las pruebas de integración:**

* Las pruebas de integración se llevan a cabo después de las pruebas de unidad.
* Combina componentes individuales que han pasado las pruebas de unidad según el diseño establecido.

**Problemas comunes durante la integración:**

* **Pérdida de datos en interfaces:** Puede ocurrir que los datos no se transmitan correctamente entre componentes o módulos.
* **Efectos adversos entre módulos:** Un módulo puede afectar inadvertidamente el funcionamiento de otro módulo debido a la interacción entre ellos.
* **Resultados inesperados al combinar subfunciones:** La combinación de varias subfunciones puede no producir el resultado esperado debido a errores de interacción.

**Estrategia de desarrollo incremental:**

* El programa se construye y prueba en pequeños segmentos.
* Este enfoque facilita la detección y corrección temprana de errores.

**Tipos de integración:**

* **Integración descendente:** Se integran los módulos comenzando desde el programa principal y descendiendo por la jerarquía de control.
  + **En profundidad (first-in-depth):** Integra todos los módulos de un camino de control principal de la estructura de forma secuencial.
  + **En anchura (first-in-breadth):** Integra todos los módulos directamente subordinados a cada nivel de la estructura antes de pasar al siguiente nivel.
  + Pasos:
    - Conductor: Módulo principal, Resguardos: Para los módulos subordinados
    - Sustituir resguardos por módulos 1 a 1
    - Probar
    - Reemplazar otro resguardo
    - Pruebas de regresión
* **Integración ascendente:** Se integran los módulos comenzando desde los módulos más bajos hasta llegar al programa principal.
  + Al integrar los módulos de manera ascendente, siempre están los subordinados, lo que elimina la necesidad de salvaguardas.
  + Pasos:
    - Combinar módulos de bajo nivel
    - Hacer conductor para coordinar entrada y salida
    - Probar el grupo
    - Eliminar conductores

#### Pickear un tipo de prueba

La principal desventaja del enfoque Descendente es la necesidad de resguardos. Por otro lado, la principal desventaja del enfoque Ascendente es que el programa no existe como entidad hasta que se agrega el último módulo.

La elección del enfoque depende de las características del software, aunque un enfoque combinado como la prueba sándwich puede ser la mejor opción.

#### Pruebas de regresión

La Prueba de regresión es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados.

Tres clases diferentes de casos de prueba:

* Una muestra representativa de pruebas que ejerciten todas las funciones del software.
* Pruebas adicionales que se centren en las funciones del software que son probablemente afectadas por el cambio.
* Pruebas que se centren en los componentes del software que han cambiado.

### Criticidad

Se deben identificar los módulos críticos para probarlos primero, que pueden ser los que:

* Abordan muchos requerimientos de software
* Tienen alto nivel de control
* Es complejo o proclive a error
* Tiene requerimientos de rendimientos definidos.

## Integración y unidad en OO

**Prueba de Unidad:**

* Se testea normalmente una clase, y en esa clase sus métodos (que son la unidad mínima testada en este caso) y el comportamiento de estado.

**Prueba de Integración:**

* El software OO no sigue una estructura de control jerárquica clara.
* La prueba basada en hebra integra el conjunto de clases necesarias para manejar una entrada o evento.
* La prueba basada en uso comienza con clases independientes y luego agrega las dependientes según sea necesario.

## Pruebas del sistema

Múltiples pruebas distintas que se fijan en la integración de todos los elementos del sistema y que hagan su trabajo.

* **Pruebas de recuperación**: Se controla la recuperación de fallas y el modo de reanudación del procesamiento en un tiempo determinado.

Generalmente se fuerza el fallo para comprobarlo.

* **Pruebas de seguridad**: Se comprueban los mecanismos de protección integrados.
* **Pruebas de resistencia (Stress)**: Se diseñan para enfrentar a los programas a situaciones anormales.
* **Prueba de rendimiento**: Se prueba el sistema en tiempo de ejecución. A veces va emparejada con la Prueba de resistencia.

## Pruebas de validación

* Nos fijamos que los requisitos estén.
* En cada caso de prueba, si hay alguna desviación de requisitos se crea una **lista de deficiencias**.
* Después de las pruebas de integración.
* Nos aseguramos de que todos los elementos de la configuración del software se hayan desarrollado apropiadamente, estén catalogados y contengan detalle suficiente para reforzar la fase de soporte.

### Pruebas ALFA y BETA

Hechas por el usuario final del desarrollo del sistema, puede ser bien informal o bien planificado.

#### ALFA

1. Hecha por un cliente en el lugar de desarrollo. Entorno controlado.
2. El desarrollador observa y anota problemas, el cliente lo usa naturalmente.
3. Después de las pruebas unitarias, de integración y del sistema.
4. No se prueba la versión final.

#### BETA

1. Se hace por los usuarios finales en sus lugares de trabajo.
2. Desarrollador no está presente normalmente, entorno no controlado.
3. El cliente registra todos los problemas e informa en intervalos regulares.
4. Última fase de las pruebas. Se hace usando técnicas de caja negra.
5. La beta se puede liberar para conseguir feedback de ahí.

## Prueba de entornos especializados

### Prueba de arquitectura cliente-servidor

* Pruebas de funcionalidad de la aplicación
* Prueba de servidor
  + Probar las funciones de coordinación y manejo de datos del servidor.
  + Desempeño del servidor (tiempo de respuesta y procesamiento total de los datos)
* Prueba de base de datos
  + Probar la exactitud e integridad de los datos, examinar transacciones, asegurar que se almacenan, actualizan y recuperan los datos.
* Pruebas de transacciones
  + Se crea una serie de pruebas para asegurar que cada transacción se procese de acuerdo a los requisitos.
* Pruebas de comunicación de red
  + Verificar comunicación entre los nodos, que el paso de mensajes, transacciones y tráfico de la red se realice sin errores.

### Prueba de la documentación y funciones de ayuda

Prueba de documentación es en dos fases:

* Revisar e inspeccionar: examinar la claridad editorial del documento.
* Prueba en vivo: usar la documentación junto con el programa real.

## Pruebas de sistemas de tiempo real

**Diseño de casos de prueba:**

* Incluye aspectos convencionales y específicos para sistemas de tiempo real.
* Considera el manejo de eventos como interrupciones.
* Temporización precisa de datos.
* Gestión del paralelismo entre tareas.

**Pruebas de tareas:**

* Evaluación individual de cada tarea.
* Objetivo: detectar errores lógicos y verificar el funcionamiento.
* Exclusión de evaluaciones de tiempo y comportamiento en esta etapa.

**Pruebas de comportamiento:**

* Simulación del comportamiento del sistema de tiempo real.
* Evaluación de la respuesta del sistema ante eventos específicos.

**Pruebas inter-tareas:**

* Evaluación de tareas asincrónicas.
* Enfoque en la comunicación entre estas tareas.

**Pruebas de sistemas:**

* Integración y prueba del software y hardware como un sistema completo.
* Validación del funcionamiento integrado del sistema de tiempo real.

## Depuración

Proceso de identificación y corrección de errores.

No es la prueba en sí, sino una consecuencia de una prueba efectiva (o sea que encontró algo).

Puede:

1. Hallarse la causa, corregirse y eliminar el problema.
2. No se encuentra la causa, y por tanto debe diseñarse un caso de prueba para investigarla.

Características de los errores:

1. Síntoma lejano (geográficamente) de la causa
2. Síntoma desaparece temporalmente al corregir otro error
3. Síntoma producido por error
4. Síntoma causado por error humano
5. Síntoma causado por problemas de tiempo
6. Condiciones de entrada difíciles de reproducir
7. Síntoma intermitente (especialmente en desarrollos hardware-software)
8. El síntoma se debe a causas distribuidas entre varias tareas que se ejecutan en diferentes procesadores

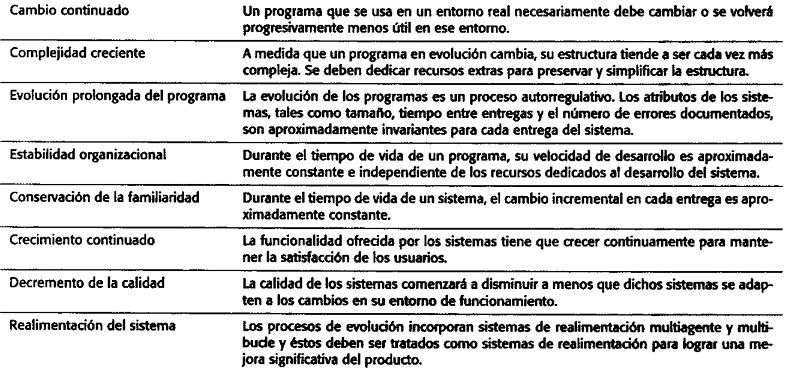
Enfoques de depuración:

* Repetir la falla con programas de prueba específicos para rastrearla.
* Rastrear el programa manualmente y simular la ejecución.
* Usar las herramientas interactivas.
* Una vez corregido el error debe reevaluarse el sistema: volver a hacer las inspecciones y repetir las pruebas (pruebas de regresión)

# Teoría 10: Mantenimiento

* Surge porque los **sistemas no son perfectos** y al lanzarse pueden tener errores no arreglados que el público masivo puede llegar a encontrar, o se quiere adaptar a las nuevas tecnologías.
* Estos **sistemas** normalmente son **viejos**, **sin** metodología o **documentación** y **nada modulares**.
* Definimos mantenimiento de software como el proceso de modificar, actualizar y cambiar el software para adaptarlo a las necesidades cambiantes del cliente. Específicamente: **Corregir errores**, **mejorar** capacidades, eliminar código muerto, y **optimizarlo**.
* Es la etapa de “Evolución del sistema”.
* A veces hay que reventar un sistema y cambiarlo por otro. Esto se decide fijándonos en el costo del ciclo de vida y la estimación del nuevo proyecto.

## Leyes de Lehman



## Características del mantenimiento

1. Otros desarrollos disminuyen.
2. Efectos secundarios sobre código, datos y documentación.
3. Puede disminuir la calidad total del producto.
4. Generalmente se reinicia las fases de análisis, diseño e implementación.
5. 40% a 70% del costo total de desarrollo.

**Problemático**: No es atractivo, el diseño puede no prever cambios y es difícil comprender código ajeno sin documentación.

## Ciclo de mantenimiento

* **Análisis**
  + **Facilidades:** Señalar principios generales, armar planes temporales, especificar controles de calidad, identificar posibles mejoras, estimar recursos para mantenimiento.
  + **Ciclo de mantenimiento:** Comprender el alcance y el efecto de la modificación.
* **Diseño**
  + **Facilidades:** Diseñar una arquitectura clara, modular y modificable, utilizando notaciones estandarizadas. Incluir notaciones para algoritmos y estructuras de datos, especificación de interfaces, manejo de excepciones y efectos colaterales.
  + **Ciclo de mantenimiento:** Rediseñar para incorporar los cambios necesarios.
* **Implementación**
  + **Facilidades:** Usar indentación adecuada, comentarios de prólogo e internos, y codificación simple y clara.
  + **Ciclo de mantenimiento:** Recodificar y actualizar la documentación interna del código.
* **Prueba**
  + **Facilidades:** Realizar lotes de prueba y documentar los resultados.
  + **Ciclo de mantenimiento:** Revalidar el software.
* **Documentación y Distribución**
  + **Facilidades:** Mantener la documentación clara y actualizada.
  + **Ciclo de mantenimiento:** Actualizar la documentación de apoyo y distribuir e instalar las nuevas versiones.



A veces el mantenimiento lo hace otro equipo distinto al que lo hizo. Involucra usuarios, operadores o representantes del cliente.

Tareas del equipo de mantenimiento:

1. Comprender el sistema
2. Localizar información en la documentación del sistema
3. Mantener actualizada la documentación del sistema
4. Ampliar las funciones existentes para adaptarse a requisitos nuevos o cambiantes
5. Agregar nuevas funciones al sistema
6. Encontrar la fuente de fallas o problemas del sistema
7. Localizar y corregir fallos
8. Responder preguntas sobre la forma en que funciona el sistema
9. Reestructuración del diseño y los componentes del código
10. Reescribir componentes de diseño y código
11. Eliminar componentes de diseño y código que ya no son útiles
12. Gestionar los cambios en el sistema a medida que se realizan

## Tipo de mantenimiento: Mantenimiento correctivo

* Acción reactiva realizada al detectar un fallo o error en una pieza de software.
* Su objetivo es restablecer el funcionamiento óptimo del sistema y minimizar el impacto en la producción.
* Se hace lo más rápidamente posible, implicando la detección de la causa del fallo, la decisión sobre su reparabilidad y la aplicación de la solución correspondiente.

## Tipo de mantenimiento: Mantenimiento Adaptativo

* Se refiere a la adaptación del software ante cambios en tecnologías, políticas y reglas. Esto incluye modificaciones en el sistema operativo, almacenamiento en la nube, hardware, entre otros.
* El software debe ajustarse para cumplir con los nuevos requisitos y seguir funcionando correctamente.

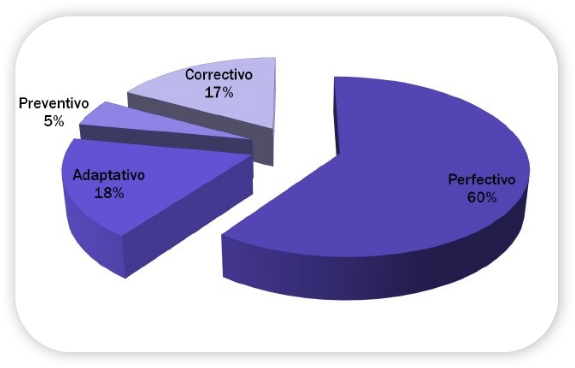
## Tipo de mantenimiento: Mantenimiento Perfectivo

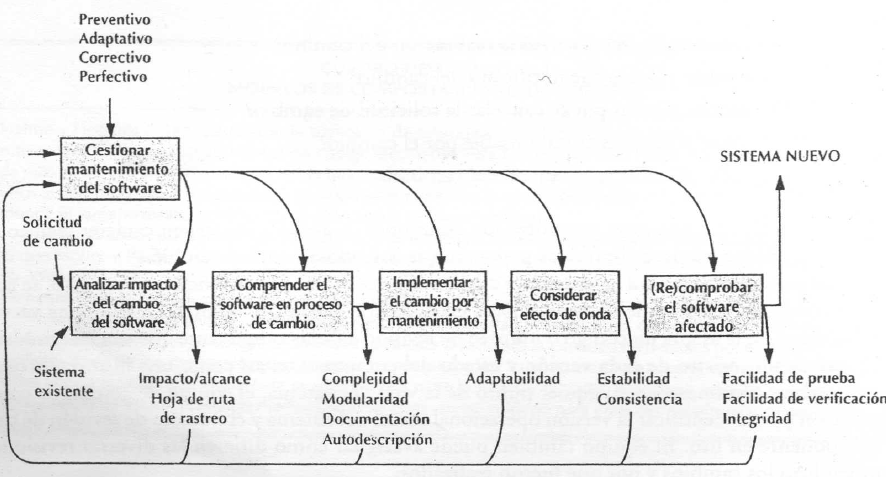
* Se enfoca en mejorar la eficiencia o rendimiento del sistema y en optimizar la experiencia del usuario mediante mejoras funcionales, basadas en comentarios de los clientes.
* Los usuarios pueden identificar nuevas características o requisitos que les gustaría ver en el software para adecuarlo mejor a sus necesidades.
* Este tipo de mantenimiento busca ajustar el software añadiendo características nuevas según sea necesario y eliminando aquellas que son irrelevantes o ineficaces.

## Tipo de mantenimiento: Mantenimiento Preventivo

Conjunto de actividades planificadas y realizadas regularmente para prevenir posibles fallos futuros.Por ejemplo:

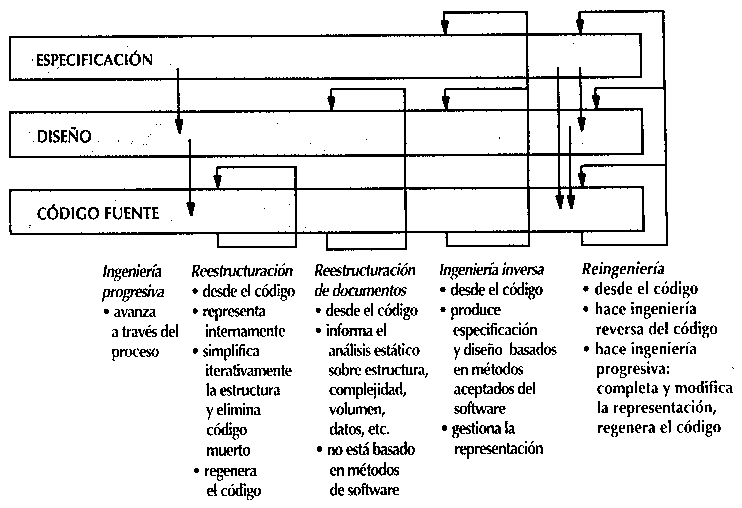
* Realizar cambios necesarios
* Actualizaciones
* Prevenir fallos en el sistema operativo, antivirus o programas ofimáticos
* Instalar y configurar un cortafuego para impedir accesos no autorizados
* Implementar programas antimalware para evitar infecciones de malware
* Activar y configurar puntos de restauración del sistema para poder revertir a un estado anterior en caso de desastre





## Rejuvenecimiento de software

Aumentar la calidad global de un sistema existente.



### Reestructuración

Se reestructura el software para hacerlo más fácil de entender y de cambiar.

1. Interpretamos el código fuente y lo representamos internamente como una red semántica o grafo.
2. Se utilizan reglas de transformación para simplificar la representación interna.
3. Los resultados se reformulan como código estructurado, donde se espera que la complejidad por ejemplo ciclomática fue reducida.

### Re-documentación

Analizamos el código estadísticamente para generar documentación, la información puede ser gráfica o textual.

Genera:

* Relaciones de llamada de componentes.
* Jerarquías de clases.
* Tablas de interfaz de datos.
* Información del diccionario de datos.
* Tablas o diagramas de flujo de datos.
* Tablas o diagramas de flujo de control.
* Pseudocódigo.
* Caminos de prueba.
* Referencias cruzadas de componentes y variables.

## Ingeniería Inversa

La ingeniería inversa es analizar un producto para entender su diseño y funcionamiento interno, incluso sin documentación. Partimos del codigo fuente.

Herramientas como Ghidra (disponible en<https://ghidra-sre.org/>) se usan para desensamblar, descompilar y analizar código binario. Ghidra convierte código ensamblador en lenguajes como C o Java, facilitando la comprensión de archivos binarios.

Como resultado, se obtienen múltiples formatos de salida que finalmente se ajustan a la especificación del sistema requerido.

## Re-ingeniería

Extensión de la ingeniería inversa. Produce código fuente nuevo que hace lo mismo pero mejor.

1. El sistema tiene ingeniería inversa y se representa internamente para humanos y modificaciones de código basadas en métodos actuales para especificar y diseñar software.
2. Se corrige o completa el modelo del sistema de software.
3. Se genera nuevo sistema a partir de esta nueva especificación o diseño.

# Teoría 11: Auditoría Informática

Es una tarea preventiva hecha para salvaguardar los activos de los sistemas informáticos y mantener la integridad de los datos además de lograr los objetivos de la organización.

El auditor sugiere, no dice, y debe evaluar todo. Objetivos:

* Salvaguardar los activos.
* Integridad de datos.
* Efectividad de sistemas.
* Eficiencia de los sistemas.
* Seguridad y confidencialidad.

Entonces decimos que la auditoría es un examen crítico para evaluar las cosas ya mencionadas y determinar cursos alternativos para lograr los objetivos.

Evalúa:

1. Los controles, sistemas y procedimientos de la informática;
2. Los equipos de cómputo;
3. La organización que participa en el procesamiento de la información.

Factores que pueden influir en la organización a través del control y la auditoría en informática:

* Controlar el uso de la computadora.
* Pérdida de capacidades de procesamiento de datos.
* Necesidad de mantener la privacidad individual.
* Posibilidad de pérdida de información o mal uso de la misma.
* Toma de decisiones incorrectas.
* Necesidad de mantener la privacidad de la organización.

Campo de acción:

1. Evaluación administrativa del área de informática.
2. Evaluación de los sistemas y procedimientos, y de la eficiencia que se tiene en el uso de la información.
3. Evaluación del proceso de datos, de los sistemas y de los equipos de cómputo (software, hardware, redes, bases de datos, comunicaciones).
4. Seguridad y confidencialidad.
5. Aspectos legales de los sistemas y de la información.

## Auditor informático

Un auditor informático verifica y evalúa los sistemas y procesos tecnológicos de una empresa para garantizar su adecuación a las necesidades de la organización y normativas, proponiendo soluciones ante posibles deficiencias detectadas.

Funciones:

* **Identificar optimizaciones** de procesos informáticos con un análisis de debilidades y errores.
* **Seguir** los **estándares** de **auditoría** establecidos por cada empresa y la normativa vigente.
* **Analizar** la **situación** actual de los **sistemas** informáticos y procesos de la empresa.
* **Controlar** y **verificar** el **funcionamiento** de los **sistemas** informáticos internos y de cualquier otro servicio que dependa de la infraestructura tecnológica de la empresa.
* **Detectar riesgos** para prevenir posibles ciberataques y desvíos de datos.
* **Elaborar informes** con los resultados de cada auditoría realizada. **Diseñar soluciones** para los problemas o errores detectados en la auditoría. **Proponer estrategias** para mejorar los sistemas informáticos.

#### Donde trabaja un auditor

Puede encontrar empleo en grandes empresas con departamentos internos de informática y ciberseguridad. También puede ser contratado externamente como auditor informático por empresas que necesiten sus servicios, o trabajar de manera independiente.

## Etapas de la auditoría informática

### Planeación

Se identifican los objetivos y define el alcance considerando los recursos y las áreas a examinar.

Se elabora un plan donde están los métodos y herramientas a usar, y un cronograma.

### Ejecución

Se recopilan los datos con entrevistas, revisión de documentos y herramientas tecnológicas.

### Análisis de los datos recabados y condiciones observadas

Evalúan los controles y procesos para determinar su eficacia y eficiencia, así como su alineación a los objetivos de la empresa. Además se identificaron deficiencias y riesgos.

### Elaboración de un informe escrito y emisión de opinión

Redacción de un informe con conclusiones, áreas de mejora y recomendaciones. Se presenta a otros organizados de la organización.

## Finalización de auditoría

Se recomienda implementar mejoras post-auditoría para fortalecer los sistemas de información de la empresa y realizar un seguimiento continuo para asegurar su efectividad y adaptabilidad al entorno tecnológico y empresarial cambiante, sugiriendo una auditoría informática anual dependiendo del tamaño y la actividad de la organización.

## Tipos de auditoria

### Interna

Usa recursos y personas de la empresa auditada. Los empleados son remunerados económicamente. Existe por decisión de la empresa.

### Externa

Realizada por personas afines a la empresa auditada, siempre remunerada. Supuestamente son más objetivas por el distanciamiento.

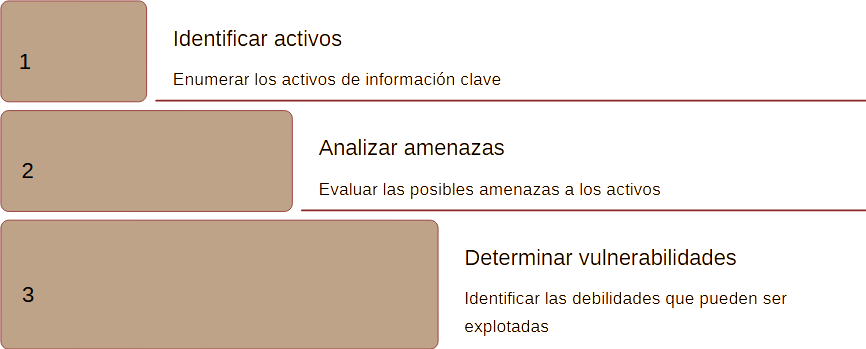
## Principios de la auditoría

1. **Principio de Beneficio del Auditado:** Maximizar la eficacia y rentabilidad de los sistemas de la empresa auditada.
2. **Principio de Calidad:** Ofrecer sus servicios utilizando los recursos científicos y técnicos disponibles de manera libre y adecuada.
3. **Principio de Confianza:** Incrementar la confianza del auditado actuando transparente y profesional, evitar hacerse el piola.
4. **Principio de Capacidad:** Debe estar capacitado para hacer la auditoría.
5. **Principio de Comportamiento Profesional:** Actuar conforme a las normas de dignidad profesional y trato personal correcto, con el auditado y terceros.
6. **Principio de Criterio Propio:** Debe actuar con criterio propio y que no influyan las opiniones de los demás si no coinciden con el suyo propio.
7. **Principio de Concentración en el Trabajo:** Evitar hacer tantas cosas a la vez que lo distraigan de la auditoría.
8. **Principio de Discreción:** Actitud de prudencia al divulgar datos.
9. **Principio de Economía:** Proteger los derechos económicos del auditado evitando gastos innecesarios.
10. **Principio de Formación Continuada:** Los auditores tienen la responsabilidad de actualizar constantemente sus conocimientos y métodos para adaptarlos a las demandas del mercado y a las exigencias competitivas.
11. **Principio de Fortalecimiento y Respeto de la Profesión:** Se enfoca en fortalecer la profesión de los auditores informáticos mediante un respeto global por su ejercicio profesional y un comportamiento que cumpla con los estándares necesarios para el éxito de las auditorías.
12. **Principio de Independencia:** Este principio, vinculado al criterio propio, requiere que los auditores mantengan total autonomía e independencia en su trabajo, ya sea que actúen como profesionales externos o estén empleados por la empresa auditada para realizar auditorías informáticas.
13. **Principio de Información Suficiente:** Proporcionar información detallada y comprensible al auditado sobre los puntos y conclusiones de la auditoría.
14. **Principio de Integridad Moral:** Actuar con honestidad, lealtad y diligencia, cumpliendo normas morales de justicia y prioridad.
15. **Principio de Legalidad:** Oponerse activamente a intentos de infringir las normas establecidas, asegurando el cumplimiento de las leyes vigentes.
16. **Principio de Libre Competencia:** Ejercer la profesión dentro del marco de la libre competencia económica, rechazando prácticas colusorias.
17. **Principio de No Discriminación:** Evitar cualquier tipo de discriminación antes, durante y después de la auditoría, actuando con igual diligencia en todos los casos.
18. **Principio de No Injerencia:** Abstenerse de interferir en el trabajo de otros profesionales y evitar aprovecharse de información obtenida de manera inapropiada.
19. **Principio de Precisión:** No concluir el trabajo hasta estar convencido, en la medida de lo posible, de la viabilidad de las propuestas.
20. **Principio de Publicidad Adecuada:** Ajustar la oferta y promoción de servicios de auditoría a las características y objetivos perseguidos, evitando la exageración o engaño.
21. **Principio de Responsabilidad:** Asumir la responsabilidad por todas las acciones, palabras y consejos en el ejercicio profesional.
22. **Principio de Secreto Profesional:** Como los psicólogos.
23. **Principio de Servicio Público:** Actuar en beneficio del interés público, sin perjudicar los intereses del cliente, para prevenir daños sociales.
24. **Principio de Veracidad:** Asegurar la veracidad en las comunicaciones con el auditado, respetando los deberes de respeto, corrección y secreto profesional

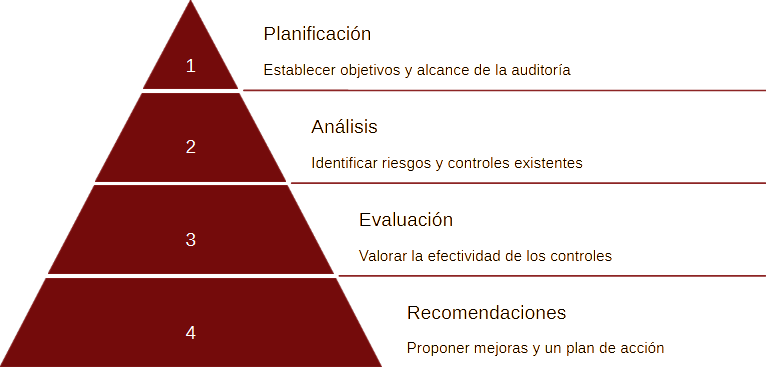
**Herramientas para realizar auditorías informáticas:** Escáneres de vulnerabilidades, analizadores de tráfico de red, herramientas de monitoreo y diagnóstico, por ejemplo Nessus, Wireshark, etc.

## Metodología Octave para auditoría informática

Serie de pasos estructurados para:



## Metodología Margerit para auditoría informática



## Controles chequeados

* Revisión de controles técnicos (como firewalls, sistemas de detección de intrusos y configuración de servidores).
* Inspección de controles físicos (como sistemas de acceso, monitoreo por cámaras y resguardo de áreas críticas).
* Evaluación de controles administrativos (como políticas de seguridad de la información, procedimientos de gestión de incidentes y concienciación de los usuarios).

## Ejemplo de reporte Recomendaciones y plan de acción

Según lo hallado se **recomienda** cosas específicas a mejorar, establece un **plan de acción** claro y estructurado, con plazos, responsables y recursos definidos, y se diseña un sistema de **seguimiento** y monitoreo para evaluar el progreso e impacto.

## Auditor vs Perito vs Consultor

**Consultor**: Asesora a las organizaciones de forma genérica con la tecnología y sistemas, realiza análisis de necesidades, propone soluciones tecnológicas y diseña estrategias de implementación, brinda recomendaciones.

**Auditor**: Lo que vimos, a diferencia del consultor, tiene acceso a los sistemas internos y controles internos de la organización y es más específico.

**Perito**: Experto para casos legales relacionados con la informática. Enfoque forense y legal, recopila pruebas digitales y analiza incidentes de seguridad. Genera análisis e informes para cosas legales.