Parcial 1 ING

Introducción:

Las **fallas de software** son distintas de las fallas mecánicas o eléctricas.
• En software, en general, las fallas no son consecuencia del uso y el deterioro.
• Las fallas ocurren como consecuencia de errores (o “bugs”) introducidos durante el desarrollo.
• La falla que causa el problema existe desde el comienzo, sólo que se manifiesta tarde.

¿Por qué es necesario el mantenimiento si el software no se deteriora con el uso?
Para corregir errores residuales (updates)
=> mantenimiento correctivo
Para mejorar funcionalmente el software (upgrades) y adaptarlo a los cambios de entorno
=> mantenimiento adaptativo
Incluye la comprensión del software existente (código y documentación), comprensión de los efectos del cambio, realización de los cambios (código y doc.), testear lo nuevo y re-testear lo viejo.

Ingeniería de Software: aplicación de un enfoque sistemático, disciplinado(manera precisa de hacerlo), y cuantificable(que se puede dividir para comparar y para poder criticar) al desarrollo, operación, y mantenimiento del software.
• Enfoque sistemático: metodología y prácticas existentes para solucionar un problema dentro de un dominio determinado.
Esto permite repetir el proceso y da la posibilidad de predecirlo (independientemente del grupo de personas que lo lleva a cabo).

DESAFÍOS DE LA IS:

El problema de producir software para satisfacer las necesidades del cliente/
usuario guía el enfoque usado en IS.
• Pero hay otros factores que tienen impacto en la elección del enfoque:
Escala, Calidad, Productividad, Consistencia, Cambios

ESCALA:

IS debe considerar la escala del sistema a desarrollar.
Los métodos utilizados para desarrollar pequeños problemas no siempre escalan a grandes problemas.
• Los métodos de IS deben tener *la capacidad de adaptación y respuesta de un sistema con respecto al rendimiento del mismo a medida que aumentan o disminuyen de forma significativa el número de usuarios o requerimientos del mismo.*

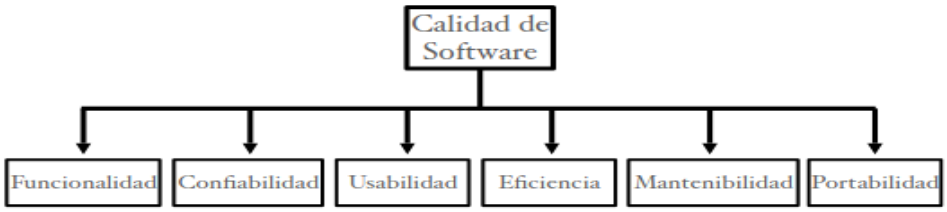
• Dos claras dimensiones a considerar:
Métodos de ingeniería. / Administración del proyecto.
• Pequeños sistemas: ambos pueden ser informales/ad-hoc.
• Grandes sistemas: ambos deben ser formalizados.
Pequeño: < 10 KLOC
Mediano: 10 a 100 KLOC
Grande: 100 a 1000 KLOC
Muy grande: > 1000 KLOC

PRODUCTIVIDAD:

IS está motivada por el costo y el cronograma.
Tanto una solución que demora mucho tiempo como una que entrega un software barato y de baja calidad son inaceptables. • El costo del software es principalmente el costo de la mano de obra, por lo que se mide en Persona/Mes (PM).
• El cronograma es muy importante en el contexto de negocios. Reducir “time to market”.
• La productividad (en términos de KLOC(linea de codigo) / PM(persona en un mes))captura ambos conceptos
Si es más alta => menor costo y/o menor tiempo.
• Los enfoques de IS deben generar alta productividad

CALIDAD

Desarrollar software de alta calidad es un objetivo fundamental. La calidad del software es difícil de definir.
• El enfoque utilizado en la IS debe producir software de alta calidad.



• Funcionalidad: Capacidad de proveer funciones que cumplen las necesidades establecidas o implicadas.
• Confiabilidad: Capacidad de realizar las funciones requeridas bajo las condiciones establecidas durante un tiempo específico.
• Usabilidad: Capacidad de ser comprendido, aprendido y usado.
• Eficiencia: Capacidad de proveer desempeño apropiado relativo a la cantidad de recursos usados.
• Mantenibilidad: Capacidad de ser modificado con el propósito de corregir, mejorar, o adaptar.
• Portabilidad: Capacidad de ser adaptado a distintos entornos sin aplicar otras acciones que las provistas a este propósito en el producto.

Confiabilidad es usualmente el principal criterio de calidad.
Confiabilidad inversamente relacionada a la probabilidad de falla. Es difícil medir la cantidad de defectos.
+fallas => -confiable
Aproximado por el número de defectos encontrados en el software.
• Para normalizar: Calidad = densidad de defectos
= Cantidad defectos en software entregado / tamaño
• En las prácticas habituales: < 1 defecto / KLOC
• Pero: ¿qué es un defecto? Depende del software.

CONSISTENCIA Y REPETITIVIDAD

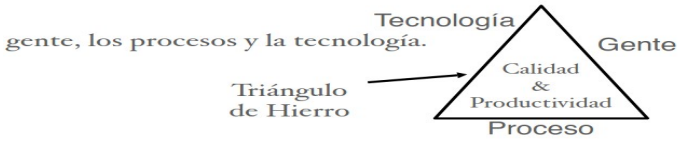
Un objetivo de la IS es la sucesiva producción de sistemas de alta calidad y con alta productividad.
• La consistencia permite predecir el resultado del proyecto con certeza razonable.
Sin consistencia sería difícil estimar costos.

CAMBIO

Cambios en las empresas/instituciones es lo habitual. Cambio el software en si. • El software debe cambiar para adaptarse a los cambios de dicha institución. • Las prácticas de IS deben preparar al software para que éste sea fácilmente modificable. Los métodos que no permiten cambios, aún si producen alta calidad y productividad, son poco útiles.

Enfoque de la IS

Ya comprendemos el dominio del problema y los factores que motivan la IS: Consistentemente desarrollar software de alta calidad y con alta productividad
(C&P) para problemas de gran escala que se adapten a los cambios.
• C&P son los objetivos básicos a perseguir bajo gran escala y tolerancia a cambios.
• C&P son consecuencia de la gente, los procesos y la tecnología.
La IS se enfoca mayormente en el proceso para conseguir los objetivos de calidad y productividad.
• El enfoque sistemático es realmente el proceso que se utiliza.
• La IS separa el proceso para desarrollar software del producto desarrollado (i.e. el software)



Premisa: El proceso es quien determina, en buena medida, la C&P => un proceso adecuado permitirá obtener gran C&P. • Diseñar el proceso apropiado y su control es el desafío clave de la IS.

Enfoque de la IS: El proceso de desarrollo en fases

El proceso de desarrollo consiste de varias fases,el **motivo de separar en fases es la separación de incumbencias**: cada fase manipula distintos aspectos del desarrollo de software.
• El proceso en fases permite verificar la calidad y progreso en momentos definidos del desarrollo (al final de la fase)

FASES:

Análisis de requisitos y especificación / Arquitectura/ Diseño/ Codificación/ Testing/ Entrega e instalación
Los enfoques sistemáticos requieren que cada etapa se realice rigurosa y formalmente.

Análisis y especificación de los requisitos del software (Capítulo 3)

Requerimientos del software
Por qué la SRS es necesaria? La SRS establece las bases para el acuerdo entre el cliente/usuario y quien suministrará el software.

Hay 3 partes involucradas:
necesidades del cliente
consideraciones del usuario
} deben comunicarse al desarrollador

Brecha comunicacional entre las partes:
Cliente: no comprende el proceso de desarrollo de software.
Desarrollador: no conoce el problema del cliente ni su área de aplicación. La SRS es el medio para reconciliar las diferencias y especificar las necesidades del cliente/usuario de manera que todos entiendan

Requerimientos (IEEE) del Software:

Una condición o capacidad necesaria de un usuario para solucionar un problema o alcanzar los objetivos.
Entender/discernir los requerimientos es complejo:
• Visualizar un futuro sistema es difícil.
• Las capacidades del futuro sistema no están claras, por lo tanto, sus necesidades tampoco.
• Los requerimientos cambian con el tiempo.
Es necesario realizar apropiadamente el análisis y especificación de los requerimientos (SRS: qué y no cómo)

Por qué la SRS es necesaria?

• Ayuda al usuario a comprender sus necesidades.
• Los usuarios no siempre saben lo que quieren o necesitan. Debe analizar y comprender el potencial.
• El proceso de requerimientos ayuda a aclarar las necesidades.
• La SRS provee una referencia para la validación del producto final.
• Debería dar una clara comprensión de lo que se espera.

Proceso de requerimientos

Secuencia de pasos que se necesita realizar para convertir las necesidades del usuario en la SRS.
• El proceso tiene que recolectar las necesidades y los requerimientos y especificarlos claramente.
El Proceso no es Lineal; es Iterativo y Paralelo. Existe superpocisión entre las fases.
• *Actividades básicas:*
1) Análisis del problema o requerimientos:Recoleccion/Extracción. Es la parte en la que se busca comprender las necesidades del cliente/usuario. ◦ Además de eso, asesorarlo y acordar ofrecimientos.
◦ *Estrategia Dividir y Conquistar:* Descomponer el problema en pequeñas partes, comprender cada una de estas partes y las relaciones entre ellas.

● *objetivo:* comprender la estructura del problema y su dominio:componentes, entradas, salidas.
Objetivo: lograr una buena comprensión de las necesidades, requerimientos, y restricciones del software.
El análisis incluye: • entrevistas con el cliente y usuarios, • lectura de manuales, • estudio del sistema actual, • ayudar al cliente/usuario a comprender nuevas posibilidades.
El analista no solo recolecta y organiza la información, i.e. rol pasivo, sino también actúa como consultor, i.e. rol activo.

2) Especificación de los requerimientos:Es la producción de la SRS.
◦ Debe ser comprensible para todas las partes involucradas. ◦ Debe seguir una estructura ordenada.
◦ Se enfoca en el comportamiento externo del sistema (Dominio de la Solución). (plasmarlo en un documento).

3) Validación:constatar que las dos partes están conformes.
Revisión y corrección de errores. ◦ El autor de la SRS, el cliente y los usuarios revisan la SRS en busca de defectos a corregir. ● Se suelen usar herramientas como "Listas de Control".
● En esta fase se utilizan las métricas para estimar costos, tiempos y esfuerzos que demandará la fabricación del producto final

Principio básico del Análisis del problema : particionar el problema.
Luego comprender cada subproblema y la relación entre ellos, pero... ¿con respecto a qué?
• Funciones: análisis estructural
• Objetos: análisis OO
• Eventos del sistema: particionado de eventos

Método del Análisis Estructurado
Se usa para automatizar el sistema ya existente.
Pasos principales:
1. Dibujar el diagrama de contexto: Ve al sistema completo como un transformador e identifica el contexto.
• Es un DFD con un único transformador (el sistema), con entradas, salidas, fuentes, y sumideros del sistema identificado

2. Dibujar el DFD del sistema existente:• El sistema actual se modela tal como es con un DFD con el fin de comprender el funcionamiento. • Se refina el diagrama de contexto.
• Cada burbuja representa una transformación lógica de algunos datos.
• Pueden usarse DFD en niveles jerárquicos.
• Para obtenerlo se debe interactuar intensamente con el usuario.
• El DFD obtenido se valida junto a los usuarios, haciendo una “caminata” a través del DFD.

3. Dibujar el DFD del sistema propuesto e identificar la frontera hombre-máquina.
a. Se hace el DFD completo:ya sean procesos automatizados o manuales.
• Validar con el usuario, también esta parte
b. Establecer qué procesos se automatizarán y cuáles permanecerán manuales.

Prototipado del Análisis del problema
• Se construye un sistema parcial prototípico.
• Cliente, usuarios y desarrolladores lo utilizan para comprender mejor el problema y las necesidades.
• Ayuda a visualizar cómo será el sistema final.
• Dos enfoques:
Descartable*:* el prototipo se construye con la idea de desecharlo luego de culminada la fase de requerimientos.
Evolucionario*:* se construye con la idea de que evolucionará al sistema final.
• *El descartable es más adecuado para esta fase del problema.*

Especificación de los requerimientos

Características de una SRS

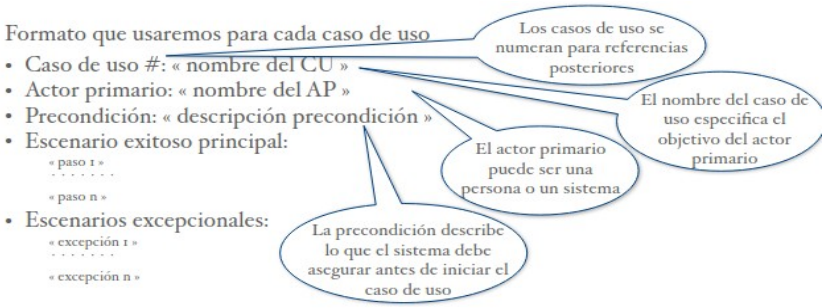
• Correcta : Cada requerimiento representa precisamente alguna característica deseada por el cliente en el sistema final.
• Completa: Todas las características deseadas por el cliente están descriptas.
La característica más difícil de lograr, para conseguirla uno debe detectar las ausencias en la especificación.
Corrección y completitud están fuertemente relacionadas
• No ambigua: Si para cada requerimiento existe un solo significado.
Si es ambigua los errores se colarán fácilmente. La no ambigüedad es esencial para verificabilidad.
Como la verificación es usualmente hecha a través de revisiones, la SRS debe ser comprensible, al menos por el desarrollador, el usuario y el cliente. Particular atención si se usa lenguaje natural.
Los lenguajes formales ayudan a "desambiguar"
• Consistente: Ningún requerimiento contradice a otro
• Verificable: Si existe para cada requerimiento algún proceso efectivo que puede asegurar que el software final satisface el requerimiento
• Rastreable (Traceable): Se debe poder determinar el origen de cada requerimiento y cómo éste se relaciona a los elementos del software.
Hacia adelante: dado un requerimiento se debe poder detectar en qué elementos de diseño o código tiene impacto.
Hacia atrás: dado un elemento de diseño o código se debe poder rastrear que requerimientos está atendiendo
• Modificable:Si la estructura y estilo de la SRS es tal que permite incorporar cambios fácilmente preservando completitud y consistencia.
La redundancia es un gran estorbo para modificabilidad, puede resultar en inconsistencia
• Ordenada en aspectos de importancia y estabilidad:Los requerimientos pueden ser críticos, importantes pero no críticos, deseables pero no importantes.
Algunos requerimientos son esenciales y difícilmente cambien con el tiempo. Otros son propensos a cambiar.
=> Se necesita definir un orden de prioridades en la construcción para reducir riesgos debido a cambios de requerimientos.

Componentes de una SRS: ¿Qué debe contener una SRS?
• Tener lineamientos sobre qué se debe especificar en una SRS ayudará a conseguir completitud.

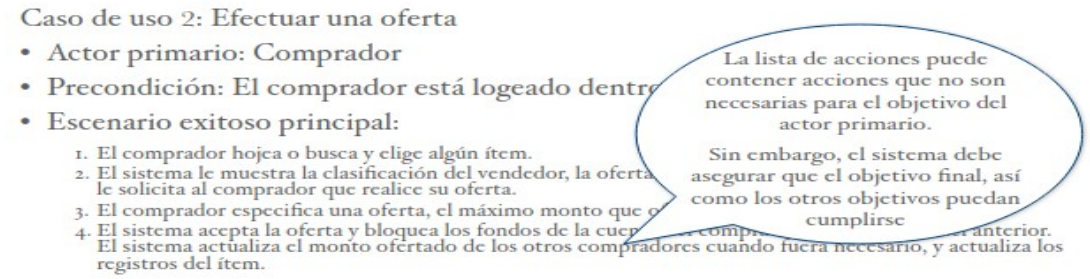
Una SRS debe especificar requerimientos sobre:
• *Funcionalidad.*
• *Desempeño (performance)*
• *Restricciones de diseño.*
• *Interfaces externas*

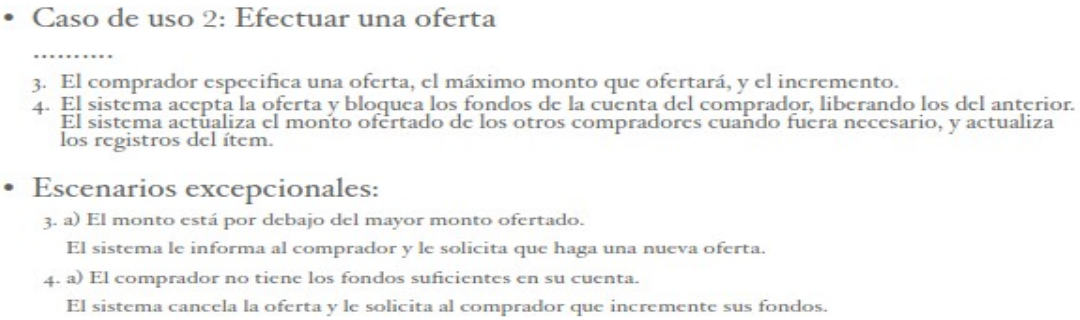
Todas las restricciones en el desempeño del sistema de software.
• **Requerimientos Dinámicos,** especifican restricciones sobre la ejecución:
Tiempo de respuesta.
Tiempo esperado de terminación de una operación dada.
Tasa de transferencia o rendimiento.
Cantidad de operaciones realizadas por unidad de tiempo.
En general se especifican los rangos aceptables de los distintos parámetros, en casos normales y extremos.
• **Requerimientos Estáticos** o de capacidad, no imponen restricción en la ejecución:
Cantidad de terminales admitidas.
Cantidad de usuarios admitidos simultáneamente.
Cantidad de archivos a procesar y sus tamaños.
• Todos los requisitos se especifican en términos medibles => verificable.

Especificación funcional: Casos de Uso: (Práctico)
"Captura el comportamiento del sistema como interacción de los usuarios con el sistema."
Formato



ejemplos:





validación de los requerimientos:

• Errores más comunes:

Omisión	30%
Inconsistencia	10-30%
Hechos incorrectos	10-30%
Ambigüedad	5-20%

Aparte de los errores de "papeleo"

Métricas

Para poder estimar costos y tiempos y planear el proyecto se necesita "medir" el esfuerzo que demandará.
• El esfuerzo del proyecto depende de muchos factores.
• El tamaño es el principal factor, validado por muchos experimentos y datos de análisis.
• Al principio el tamaño sólo puede ser estimado.
• Conseguir buenos estimadores es muy difícil.
• *Una métrica es importante sólo si es útil para el seguimiento o control de costos, calendario o calidad.*
• Se necesita una unidad de tamaño que se pueda computar a partir de los requerimientos:
¿Tamaño de la SRS? => dependen mucho del autor

Punto función

- Es una estimación similar a la métrica LOC.
- Se determina sólo con la SRS.
- Define el tamaño en términos de la "funcionalidad".

Tipo de funciones:
Entradas externas
Salidas externas
Archivos lógicos internos
Archivos de interfaz externa
Transacciones externas

Tipo de entrada (dato/ control) externa a la aplicación

Tipo de salida que deja el sistema

Punto función

- Es una estimación similar a la métrica LOC.
- Se determina sólo con la SRS.
- Define el tamaño en términos de la "funcionalidad".

Tipo de funciones:	Simp.	Prom.	Comp.
Entradas externas	3	4	6
Salidas externas	4	5	7
Archivos lógicos internos	7	10	15
Archivos de interfaz externa	5	7	10
Transacciones externas	3	4	6

Pesos (w_{ij})

Punto función

- Contar cada tipo de función diferenciando según sea compleja, promedio o simple.
- C_{ij} denota la cantidad de funciones tipo "i" con complejidad "j".
- Punto función no ajustado (UFP):

$$\sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} C_{ij}$$

Punto función

- Ajustar el UFP de acuerdo a la complejidad del entorno.
  Se evalúa según las siguientes características:

1. comunicación de datos
2. procesamiento distribuido
3. objetivos de desempeño
4. carga en la configuración de operación
5. tasa de transacción
6. ingreso de datos online
7. eficiencia del usuario final
8. actualización online
9. complejidad del procesamiento lógico
10. reusabilidad
11. facilidad para la instalación
12. facilidad para la operación
13. múltiples sitios
14. intención de facilitar cambios

- Factor de ajuste de complejidad (CAF):
- Puntos función = CAF · UFP

1 punto función =
- 125 LOC en C
- 50 LOC en C++ o Java

Cada uno de estos ítems debe evaluarse como: p_i

no presente	0
influencia insignificante	1
influencia moderada	2
influencia promedio	3
influencia significativa	4
influencia fuerte	5

$$0.65 + 0.01 \sum_{i=1}^{14} p_i$$

La calidad de la SRS tiene impacto directo en los costos del proyecto.
=> se necesitan buenas métricas de calidad para evaluar la calidad de la SRS.
• ***Métricas de calidad directa***: evalúan la calidad del documento estimando el valor de los atributos de calidad de la SRS.
• ***Métricas de calidad indirecta:*** evalúan la efectividad de las métricas del control de calidad usadas en el proceso en la fase de requerimientos.
Ej.: Número de errores encontrados. Frecuencia de cambios de requerimientos.

Arquitectura del Software (Capítulo 4)

El rol de la arquitectura de software
Definición: La arquitectura de SW de un sistema es la estructura del sistema que comprende los elementos del SW, las propiedades externamente visibles de tales elementos, y la relación entre ellas.
• Por cada elemento sólo interesan las propiedades externas necesarias para especificar las relaciones.
• Para la arquitectura no son importantes los detalles de cómo se aseguran dichas propiedades.
• La definición no habla de la bondad de la arquitectura; para ello se necesita analizarla.

En general, se tienen varias estructuras, de aspectos ortogonales. Una descripción arquitectónica del sistema describe las distintas estructuras del sistema.

La “Arquitectura” es el Diseño de más alto nivel:
● Divide al sistema en partes lógicas tal que cada una puede ser comprendida independientemente y describe las relaciones entre ellos.
o Un enfoque para diseñar sistemas es identificar Subsistemas y la forma que interactúan entre ellos.

¿Por qué?

- **Comprensión y comunicación:**

• Al mostrar la estructura de alto nivel del sistema ocultando la complejidad de sus partes, la descripción arquitectónica facilita la comunicación.
• Define un marco de comprensión común entre los distintos interesados: usuarios, cliente, arquitecto, diseñador, etcétera. • Útil para negociación y acuerdos. • También puede ayudar a comprender un sistema existente

- **Reuso:**

• El reuso es considerado una de las principales técnicas para incrementar la productividad.
• Una forma de reuso es componer el sistema con partes existentes, reusadas, otras nuevas.
• Se facilita si a un alto nivel se reusan componentes que proveen un servicio completo.
=> Se elige una arquitectura tal que las componentes existentes encajen adecuadamente con otras componentes a desarrollar.
=> Las decisiones sobre el uso de componentes existentes se toman en el momento de diseñar la arquitectura.

- Construcción y evolución:

• La división provista por la arquitectura servirá para guiar el desarrollo del sistema:
• Cuáles partes son necesarias construir primero, cuáles partes ya están construidas.
• Ayuda a asignar equipos de trabajo: las distintas partes pueden construirse por distintos grupos, si las partes son relativamente independientes entre sí.
• Durante la evolución del SW, la arquitectura ayuda a decidir cuáles partes necesitan cambiarse para incorporar nuevas características o cambios, y a decidir cuál es el impacto de tales cambios en las otras componentes

- Análisis:

• Es deseable que propiedades de confiabilidad y desempeño puedan determinarse en el diseño de alto nivel; esto permite considerar distintas alternativas de diseño hasta encontrar los niveles de satisfacción deseados.
• Requerirá descripción precisa de la arquitectura así como de las propiedades de las componentes.

VISTA DE LA ARQUITECTURA: Se refiere a una flia de arquitectura que cumplen las mismas restricciones.
Vistas de la arquitectura tres tipos: Módulo / Componentes y conectores / Asignación de recursos

Vistas de Módulos:
● Un sistema es una colección de unidades de código (ellas no representan entidades en ejecución).
o La relación entre ellos está basada en el código.
o Los elementos son módulos: Clases, paquetes, funciones, métodos, etc.

-Vista de asignación de recursos:
● Se enfoca en cómo las unidades de software se asignan a recursos como el Hardware, sistemas de archivos, gente, etc.
o Especifica la relación entre los elementos del Software y las unidades de ejecución en el entorno.
● Exponen propiedades estructurales como qué proceso ejecuta en qué procesador, qué archivo reside dónde, etc

Vista de componentes y conectores:
• Los elementos son entidades de ejecución denominados componentes, i.e., una componente es una unidad que tiene identidad dentro del sistema en ejecución. Ej.: objetos, procesos, .exe,
• Los conectores proveen el medio de interacción entre las componentes.
Ej.: pipes, sockets, memoria compartida, protocolos, etcétera

• Dos elementos principales: componentes y conectores.
Componentes: son elementos computacionales o de almacenamiento de datos.
Conectores: son mecanismos de interacción entre las componentes.
• Una vista C&C define las componentes y cómo se conectan entre ellas a través de conectores.
• La vista C&C describe una estructura en ejecución del sistema: qué componentes existen y cómo interactúan entre ellos en tiempo de ejecución.
• Es básicamente un grafo donde las componentes son los nodos y los conectores las aristas.

Componentes
• Son unidades de cómputo o de almacenamiento de datos. • Cada componente tiene un nombre que representa su rol y le provee una identidad. • Cada componente tiene un tipo.
Los distintos tipos se representan con distintos símbolos.
• Las componentes utilizan interfaces o puertos para comunicarse con otras componentes

Conectores
• Describen el medio en el cual la interacción entre componentes toma lugar.
• Un conector puede proveerse por medio del entorno de ejecución.
Ej.: llamada a procedimiento/función.
• Sin embargo, los conectores pueden también ser mecanismos de interacción más complejos, ej.: puertos TCP/IP, RPC, protocolos como HTTP, etcétera.
• Notar que estos mecanismos requieren una infraestructura de ejecución significativa + programación especial dentro de la componente para poder utilizarla.

La vista C&C ¿que describe?describe una estructura en ejecución del sistema: qué componentes existen y cómo interactúan entre ellos en tiempo de ejecución.
• Es básicamente un grafo donde las componentes son los nodos y los conectores las aristas.

Estilos arquitectónicos para la vista de C&C

• Sistemas distintos tienen estructuras de C&C distintas.
• Algunas estructuras son generales y son útiles para una clase de problemas => estilos arquitectónicos.
• Un estilo arquitectónico **define** una familia de arquitecturas que satisface las restricciones de ese estilo.
• Los estilos proveen ideas para crear arquitecturas de sistemas.
• Distintos estilos pueden combinarse para definir una nueva arquitectura.

1. **Tubos y Filtros (Pipe and Filter):**
   • Adecuado para sistemas que fundamentalmente realizan transformacion de datos.
   • Un sistema que usa este estilo utiliza una red de transformadores para realizar el resultado deseado.
   • Tiene un sólo *tipo de componente*: filtro.
   • Tiene un sólo *tipo de conector*: tubo.
   • Un filtro realiza transformaciones y le pasa los datos a otro filtro a través de un tubo

Restricciones 1:
• Un filtro es una entidad independiente y asíncrona (se limita a consumir y producir datos).
• Un filtro no necesita saber la identidad de los filtros que envían o reciben los datos.
• Un tubo es un canal unidireccional que transporta un flujo de datos de un filtro a otro.
• Un tubo sólo conecta 2 componentes.
• Los filtros deben hacer “buffering” y sincronización para asegurar el correcto funcionamiento como productor y consumidor.
Restricciones 2:
• Cada filtro debe trabajar sin conocer la identidad de los filtros productores o consumidores.
• Un tubo debe conectar un puerto de salida de un filtro a un puerto de entrada de otro filtro.
• Un sistema puro de tubos y filtros usualmente requiere que cada filtro tenga su propio hilo de control.

2)**Estilo de datos compartidos**
• Dos tipos de componentes: repositorio de datos y usuarios de datos.
• Repositorio de datos: provee almacenamiento permanente confiable.
• Usuarios de datos: acceden a los datos en el repositorio, realizan cálculos, y ponen los resultados otra vez en el repositorio. • La comunicación entre los usuarios de los datos sólo se hace a través del repositorio.
• En este estilo sólo hay un tipo de conector: lectura/escritura
Dos variantes principales:
• ***Estilo pizarra:***
Cuando se agregan/modifican datos en el repositorio, se informa a todos los usuarios. i.e.: la fuente de datos compartidos es una entidad activa.
• ***Estilo repositorio:***
El repositorio es pasivo. Ej.: sistemas orientados a base de datos; sistemas de web; entornos de programación; etcétera.

3)**Estilo cliente-servidor**
• Dos tipos de componentes: clientes y servidores.
• Los clientes sólo se comunican con el servidor, pero no con otros clientes.
• La comunicación siempre es iniciada por el cliente quien le envía una solicitud al servidor y espera una respuesta de éste => la comunicación es usualmente asincrónica.
• Solo un tipo de conector: solicitud/respuesta (request/reply) - es asimétrico.
• Usualmente el cliente y el servidor residen en distintas máquinas
forma de una estructura multi-nivel. El servidor también actúa como cliente.
Ej. clásico => 3 niveles: • *Nivel de cliente*: contiene a los clientes.
• *Nivel intermedio*: contiene las reglas del servicio.
• *Nivel de base de datos*: reside la información.

4)Estilo publicar-suscribir (publish-subscribe):
• Dos tipos de componentes: las que publican eventos y las que se suscriben a eventos.
• Cada vez que un evento es publicado se invoca a las componentes suscriptas a dicho evento.
5) Estilo peer-to-peer:
• Un único tipo de componente. • Cada componente le puede pedir servicios a otra ≈ modelo de computación orientado a objetos.
6) Estilo de procesos que se comunican:
• Procesos que se comunican entre sí a través de pasaje de mensajes

El método de análisis ATAM

Pasos principales:
1. Recolectar escenarios:
• Los escenarios describen las interacciones del sistema. • Elegir los escenarios de interés para el análisis.
• Incluir escenarios excepcionales sólo si son importantes.
2. Recolectar requerimientos y/o restricciones:
• Definir lo que se espera del sistema en tales escenarios.
• Deben especificar los niveles deseados para los atributos de interés (preferiblemente cuantificados.)
3. Describir las vistas arquitectónicas:
• Las vistas del sistema que serán evaluadas son recolectadas.
• Distintas vistas pueden ser necesarias para distintos análisis.
4. Análisis específicos a cada atributo:
• Se analizan las vistas bajo distintos escenarios separadamente para cada atributo de interés distinto. • Esto determina los niveles que la arquitectura puede proveer en cada atributo.
• Se comparan con los requeridos. • Esto forma la base para la elección entre una arquitectura u otra o la modificación de la arquitectura propuesta. • Puede utilizarse cualquier técnica o modelado
5. Identificar puntos sensitivos y de compromisos:
• Análisis de sensibilidad: cuál es el impacto que tiene un elemento sobre un atributo de calidad. Los elementos de mayor impacto son los puntos de sensibilidad.
• Análisis de compromiso: Los puntos de compromiso son los elementos que son puntos de sensibilidad para varios atributos.

Diseño (Capítulo 6-7)

Niveles en el proceso de diseño

• Diseño arquitectónico:

• Identifica las componentes necesarias del sistema, su comportamiento y relaciones.

• ***Diseño de alto nivel:*** • Es la vista de módulos del sistema.

• Es decir: cuáles son los módulos del sistema, qué deben hacer, y cómo se organizan/ interconectan.

• Diseño detallado o diseño lógico:

• Establece cómo se implementan las componentes/módulos de manera que satisfagan sus especificaciones.

• Incluye detalles del procesamiento lógico (i.e. algoritmos) y de las estructuras de datos.

• Muy cercano al código

Criterios para evaluar el diseño

Objetivo: encontrar el mejor diseño posible.

• Se deberán explorar diversos diseños alternativos.

• Los criterios de evaluación son usualmente subjetivos y no cuantificables.

Principales criterios para evaluar:

• Corrección

• Eficiencia

• Simplicidad

Corrección:

• Es fundamental (¡pero no el único!)

• ¿el diseño implementa los requerimientos?

• ¿es factible el diseño dada las restricciones?

• Eficiencia:

• Le compete el uso apropiado de los recursos del sistema (principalmente CPU y memoria).

• Debido al abaratamiento del hardware toma un segundo plano.

(OJO: muy importante en ciertas clases de sistemas, ej.: sistemas integrados o de tiempo real).

Simplicidad:

• Tiene impacto directo en mantenimiento.

• El mantenimiento es caro. • Un diseño simple facilita la comprensión del sistema => hace al software mantenible. • Facilita el testing.

• Facilita el descubrimiento y corrección de bugs. • Facilita la modificación del código.

Eficiencia y simplicidad no son independientes => el diseñador debe encontrar un balance

Principios fundamentales:

• Partición y jerarquía

• Abstracción

• Modularidad

particion y jerarquia

Principio básico: “divide y conquistarás”

• Dividir el problema en pequeñas partes que sean manejables:

• Cada parte debe poder solucionarse separadamente. • Cada parte debe poder modificarse separadamente.

• Las partes no son totalmente independientes entre sí: deben comunicarse/ cooperar para solucionar el problema mayor. • La comunicación agrega complejidad.

• A medida que la cantidad de componentes aumenta, el costo del particionado (incluyendo la complejidad de la comunicación) también aumenta.

• Detener el particionado cuando el costo supera al beneficio

Abstracción

• Esencial en el particionado del problema.

• Utilizado en todas las disciplinas de ingeniería.

• La abstracción de una componente describe el comportamiento externo sin dar detalles internos de cómo se produce dicho comportamiento.
• Abstracción de componentes existentes:
• Representa a las componentes como cajas negras.
• Oculta detalle, provee comportamiento externo.
• Útil para comprender sistemas existentes => tiene un rol importante en mantenimiento.
• Útil para determinar el diseño del sistema existente

Dos mecanismos comunes de abstracción:
• Abstracción funcional.
• Abstracción de datos.
• *Abstracción funcional:*
• Especifica el comportamiento funcional de un módulo.
• Los módulos se tratan como funciones de entrada/salida.
• La mayoría de los lenguajes proveen características para soportarla. Ej.: procedimientos, funciones.
• Un módulo funcional puede especificarse usando pre y postcondiciones.
• Forma la base de las metodologías orientadas a funciones.
Abstracción de datos:
• Una entidad del mundo real provee servicios al entorno.
• Es el mismo caso para las entidades de datos: se esperan ciertas operaciones de un objeto de dato.
• Los detalles internos no son relevantes.
• La abstracción de datos provee esta visión:
• Los datos se tratan como objetos junto a sus operaciones.
• Las operaciones definidas para un objeto sólo pueden realizarse sobre este objeto.
• Desde fuera, los detalles internos del objetos permanecen ocultos y sólo sus operaciones son visibles.
• Muchos lenguajes soportan abstracción de datos.
• Forma la base de las metodologías orientadas a objetos.

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

Principios de diseño
Estrategias top-down y bottom-up
Un sistema es una jerarquía de componentes.
• Dos enfoques para diseñar tal jerarquía:
• Top-down:
• comienza en la componente de más alto nivel, la más abstracta;
• prosigue construyendo las componentes de niveles más bajos descendiendo en la jerarquía.
• Bottom-up:
• comienza por las componentes de más bajo nivel en la jerarquía, las más simples;
• prosigue hacia los niveles más altos hasta construir la componente más alta

4 – Diseño orientado a función

Criterios utilizados para seleccionar módulos que soporten abstracciones bien definidas y solucionables/modificables separadamente:
• Acoplamiento • Cohesión

Acoplamiento
Dos módulos son independientes si cada uno puede funcionar completamente sin la presencia del otro.

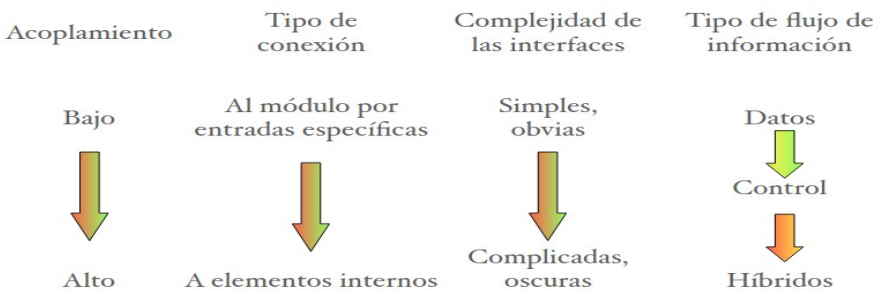
• La independencia entre módulos es deseable: • Los módulos se pueden modificar separadamente.
• Se pueden implementar y testear independientemente. • El costo de programación decrece.
• En un sistema no existe la independencia entre todos los módulos.
• Los módulos deben cooperar entre sí.
• Cuanto más conexiones hay entre dos módulos, más dependientes son uno del otro, i.e. se requiere más conocimiento de un módulo para comprender el otro módulo.
• El acoplamiento captura la noción de dependencia

Factores más importantes que influyen en el acoplamiento:
• Tipo de conexiones entre módulos.
• Complejidad de las interfaces.
• Tipo de flujo de información entre módulos.

Acoplamiento
La complejidad y oscuridad de las interfaces de un módulo, i.e. el tipo de conexión incrementan el acoplamiento.
• Minimizar la cantidad de interfaces por módulo.
• Minimizar la complejidad de cada interfaz.
• **El acoplamiento disminuye si:**
• Sólo las entradas definidas en un módulo son utilizadas por otros.
• La información se pasa exclusivamente a través de parámetros.
• **El acoplamiento se incrementa si:**
• Se utilizan interfaces indirectas y oscuras.
• Se usan directamente operaciones y atributos internos al módulo.
• Se utilizan variables compartidas.

El acoplamiento se incrementa con la complejidad de cada interfaz, ej.: cantidad y complejidad de los parámetros.
• Usualmente se usa más de lo necesario, ej.: pasar un registro completo cuando sólo un campo es necesario.
• Cierto nivel de complejidad en las interfaces es necesario para soportar la comunicación requerida con el módulo. => Encontrar balance.
• Mantener las interfaces de los módulos lo más simple que sea posible.

El acoplamiento depende del tipo del flujo de información.
• Dos tipos de información: **control o dato**.
• **Transferencia de información de control:**
• Las acciones de los módulos dependen de la información.
• Hace que los módulos sean más difíciles de comprender.
• **Transferencia de información de datos:**
• Los módulos se pueden ver simplemente como funciones de entrada/salida.
• **Bajo acoplamiento**: las interfaces sólo contienen comunicación de datos.
• **Alto acoplamiento**: las interfaces contienen comunicación de información híbrida: datos+control

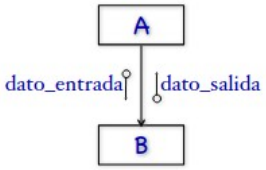


cohesion:El acoplamiento caracteriza el vínculo inter-modular.
• Se reduce minimizando las relaciones entre los elementos de los distintos módulos.
Otra forma de lograr un efecto similar es maximizando las relaciones entre los elementos del mismo módulo.
• La cohesión considera esta relación.

• La cohesión caracteriza el vínculo intra-modular.
• Con la cohesión intentamos capturar cuan cercanamente están relacionados los elementos de un módulo entre sí.
Buscamos: **menor acoplamiento y mayor cohesión.**

Niveles de Cohesión (de menor a mayor):
● **Casual**: La relación entre los elementos del módulo no tiene significado.
● **Lógica**:Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.
● **Temporal:** Parecido a cohesión lógica pero los elementos están relacionados en el tiempo y se ejecutan juntos.
● **Procedural:** Contiene elementos que pertenecen a una misma unidad procedural.
● **Comunicacional:** Tiene elementos que están relacionados por una referencia al mismo dato, i.e. los elementos están juntos porque operan sobre el mismo dato.
● **Secuencial:** Los elementos están juntos porque la salida de uno corresponde a la entrada de otro. Puede contener varias funciones o parte de una.
● **Funcional:** Es la más fuerte de todas las cohesiones: Todos los elementos del módulo están relacionados para llevar a cabo una sola función

Diagrama de estructura
Todo programa tiene estructura.
• Diagrama de estructura: • Presenta una notación gráfica para tal estructura estática del software. • Representa módulos y sus interconexiones. • La invocación de A a B se representa con una flecha. • Cada flecha se etiqueta con los ítems que se pasan.



Pasos de la Metodología SDM:
1. Reformular el problema como un DFD.
● comienza con un DFD Que capture el flujo de datos del sistema propuesto. Provee una visión de alto nivel del sistema. Enfatiza el flujo de datos a través del sistema.• Identificar las entradas, salidas, fuentes, sumideros del sistema.● Ignora aspectos procedurales.
2. Identificar las entradas y salidas más abstractas.
● Primero la entrada debe convertirse en un formato adecuado.
● Las salidas producidas por los transformadores principales deben transformarse a salidas físicas adecuadas. Se requieren varios transformadores para procesar las entradas y las salidas.
Objetivo de este paso: separar tales transformadores de los que realizan las transformaciones reales
● **Entradas Más Abstractas (MAI):** Elementos de datos en el DFD que están más distantes de la entrada real, pero que aún puede considerarse como entrada.
o Para encontrar la MAI: ▪ Ir desde la entrada física en dirección de la salida hasta que los datos no puedan considerarse entrantes. ▪ Ir lo más lejos posible sin perder la naturaleza entrante.
o Es similar para obtener las **Salidas Más Abstractas (MAO).**

3. Realizar el primer nivel de factorización.
● Primer paso para obtener el diagrama de estructura.
Especificar el módulo principal.
Especificar un módulo de entrada subordinado por cada ítem de dato de la MAI.
Especificar un módulo de salida subordinado por cada ítem de dato de la MAO.
Especificar un módulo transformador subordinado por cada transformador central.
Las entradas y salidas de estos módulos transformadores están especificadas en el DFD.
4. Factorizar los módulos de entrada, de salida, y transformadores.
Se repite el proceso del primer nivel de factorización considerando al módulo de entrada como si fuera el módulo principal.
• Se crea un módulo subordinado por cada ítem de dato que llega a este nuevo transformador central.
• Se crea un módulo subordinado para el nuevo transformador central.

La **factorización de la salida** es simétrica.
• Módulos subordinados: un transformador y módulos de salida.

Factorizar los transformadores centrales

La factorización de los módulos de entrada y salida es simple si el DFD es detallado.
• No hay reglas para factorizar los módulos transformadores.
• Utilizar proceso de refinamiento top-down.
• Objetivo: determinar los subtransformadores que compuestos conforman el transformador.
• Repetir el proceso para los nuevos transformadores encontrados. Tratar al transformador como un nuevo problema en sí mismo. • Graficar DFD. • Luego repetir el proceso de factorización.
• Repetir hasta alcanzar los módulos atómicos.

5. Mejorar la estructura (heurísticas, análisis de transacciones).
● SDM apunta a acercarse a la factorización completa

Heurística de diseño: "Un conjunto de "rules of thumb" que generalmente son útiles".
La estructura obtenida podría modificarse si fuera necesario (si uno lo considera así).
• El objetivo, siempre, es lograr bajo acoplamiento y alta cohesión. • Utilizar heurísticas de diseño para modificar el diseño inicial. "Ver el resultado total como un todo y evaluar el resultado general"

Métricas (~)
"Busca proveer una evaluación cuantitativa del diseño (así el producto final puede mejorarse)."
Métricas : • Tamaño
• Complejidad
• De red
• De estabilidad
• Flujo de la información

El **tamaño y la complejidad** suelen ser métricas de interés.
Métricas de Red: Se enfoca en la estructura del diagrama de estructuras. Cuanto más se desvíe de la "forma de árbol", más impuro es el diagrama:
o Impureza del grafo = n - e – 1 EN DONDE ▪ n = nodos del grafo Y ▪ e = aristas el grafo
o Impureza=0 => árbol
▪ A medida que este valor se hace más negativo, se incrementa la impureza.

Métricas de Estabilidad
Trata de capturar el impacto de los cambios en el diseño. Cuanta mayor estabilidad, mejor. Estabilidad de un módulo: la cantidad de suposiciones por otros módulos sobre éste. Se conocen luego del diseño.

Métricas de Flujo de información
"El acoplamiento también se incrementa con la complejidad de la interfaz.". • Las métricas de flujo de información tienen en cuenta:
• la complejidad **intra-módulo**, que se estima con el tamaño del módulo en LOC.
• la complejidad **inter-módulo** que se estima con
• **inflow**: flujo de información entrante al módulo.
• **outflow**: flujo de información saliente del módulo.
• La complejidad del diseño del módulo C se define como
• **DC = tamaño * (inflow * outflow)2**
También es importante la cantidad de módulos desde y hacia donde fluye la información. Entonces la complejidad del diseño del módulo C:
o DC = fan_in * fan_out + inflow * outflow donde:
▪ fan_in: cantidad de módulos que llaman al módulo C.
▪ fan_out: cantidad de módulos llamados por C.
Para usar la métrica : Se usa el promedio de la complejidad de los módulos y su desviación estándar para identificar los módulos complejos y los propenso a error:
o Propenso a error si: ▪ **DC > complej_media + desv_std**
o Complejo si: ▪ **complej_media < DC < complej_media + desv_std**
o Normal en caso contrario.

4 – Diseño orientado a objetos

El propósito del diseño OO es el de definir las clases del sistema a construir y las relaciones entre éstas.

objetos :es el encapsulado:
• Encapsula datos e información (i.e. el estado) y provee interfaces para accederlos y modificarlos. • Brinda abstracción y ocultado de información.
• Los objetos tienen estado persistente: • Las funciones y procesos no retienen el estado.
• Un objeto sabe de su pasado y mantiene el estado. • Los objetos tienen identidad: cada objeto puede ser identificado y tratado como una entidad distinta. • El comportamiento del objeto queda definido conjuntamente por los servicios y el estado: • La respuesta de un objeto depende del estado en que se encuentra.

Clases
• Una clase es una plantilla del cual se crean los objetos; define la estructura y los servicios.
• Una clase tiene: • una interfaz que define cuales partes de un objeto pueden accederse desde el exterior,
• un cuerpo que implementa las operaciones, • variables de instancias para retener el estado del objeto.
• Las **operaciones de una clase** pueden ser:
• **públicas**: accesibles del exterior,
• **privadas**: accesibles sólo desde dentro de la clase,
• **protegidas:** accesibles desde dentro de la clase y desde sus subclases.
• Existen operaciones constructoras y destructoras.

Acoplamiento
"El acoplamiento es un concepto inter-modular, captura la fuerza de interconexión entre módulos"
Objetivo: Bajo Acoplamiento.
Tipos de Acoplamiento:
Acoplamiento por Interacción:
o Ocurre cuando: Métodos de una clase invocan a métodos de otra clase.
o **Mayor acoplamiento** cuando: ▪ Métodos acceden partes internas de otros métodos.
▪ Métodos manipulan directamente variables de otras clases.
o **Menor acoplamiento** si: ▪ Métodos se comunican directamente a través de los parámetros.
▪ Pasando la menor cantidad de información posible.
Acoplamiento de Componentes:
o Ocurre cuando: Una clase A tiene variables de otra clase C.
o Mayor acoplamiento si: ▪ Cuando A está acoplado con C, también está acoplado con todas sus subclases.
o **Menor acoplamiento si:** ▪ Las variables de clase C en A son atributos o parámetros en un método son visibles.
Acoplamiento de Herencia:
o Ocurre cuando: Si una clase es subclase de otra.
o **Mayor acoplamiento** si: Las subclases modifican la signatura de un método o eliminan un método.
▪ Si modifica el comportamiento de un método.
o **Menor acoplamiento** si:▪ La subclase solo agrega variables de instancia y métodos, no modifica los existentes en la superclase.

Cohesión:
"La cohesión es un concepto intra-modular; captura cuán relacionado están los elementos de un módulo."
Objetivo: **alta cohesión.**
Tipos de Cohesión:
Cohesión de método:
o Ocurre cuando: Los elementos están juntos en el mismo método.
o **Mayor cohesión** si: ▪ Si cada método implementa una única función claramente definida con todos sus elementos contribuyendo a implementar esta función.

Cohesión de clase:
o Ocurre cuando: Los distintos atributos y métodos están en la misma clase. Una clase debería representar un único concepto con todos sus elementos contribuyendo a este concepto.
• Si una clase encapsula múltiples conceptos, la clase pierde cohesión
o **Mayor cohesión** si: ▪ Una clase debería representar un único concepto con todos sus elementos contribuyendo a este concepto.

Cohesión de la herencia:
o Ocurre cuando: Distintas clases están juntas en la misma jerarquía.
o **Mayor cohesión** si: Si la jerarquía se produce como consecuencia de la generalización-especialización.
● Definir subclases sirve para el reúso.

Principio de Responsabilidad Única (Single Responsibility)
Principio abierto-cerrado (Open-Closed)
Principio de sustitución de Liskov (Liskov Substitution)
Principio de segregación de interfaces (Interface Segregation)
Principio de inversión de dependencia (Dependency Inversion)

Principio Abierto-Cerrado:
"Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas"
Permitir agregar código sin modificar el existente.
o Minimiza el riesgo de "dañar" la funcionalidad existente al ingresar cambios.
o Programadores prefieren escribir código nuevo en lugar de cambiar el existente.
En OO este principio es satisfecho si se usa apropiadamente la herencia y el polimorfismo.

Principio de sustitución de Liskov:
"Un programa que utiliza un objeto O con clase C debería permanecer inalterado si O se reemplaza por cualquier objeto de una subclase de C". Si las jerarquías de un programa siguen este principio, entonces el programa responde al principio abierto-cerrado.

Principio de Responsabilidad Única (RSP)
Una clase debe tener sólo una razón para cambiar, lo que significa que sólo debe tener un trabajo o responsabilidad

Principio de segregación de interfaces (ISP)
Las interfaces deben ser específicas y enfocarse en los requerimientos de los clientes que las utilizan

Principio de inversión de dependencia (DIP)
Las clases deben depender de las abstracciones/interfaces y no de implementaciones concretas.
Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones. Además, las abstracciones no deberían depender de los detalles, pero los detalles deberían depender de las abstracciones.

Metodología de Diseño: Object Modeling Technique (OMT)
Usando este modelo se debe producir un modelo detallado final.
Se utiliza en la fase de ???.
Pasos:
1. Producir el diagrama de clases:diagrama obtenido en análisis
2. Producir el modelo dinámico y usarlo para definir operaciones de las clases:
El modelo dinámico apunta a especificar cómo cambia el estado de los distintos objetos cuando ocurre un evento.
• Evento: desde el punto de vista de un objeto, es una solicitud de operación.
• Escenario: secuencia de eventos que ocurre en una ejecución particular del sistema, recordar casos de uso.
• Los escenarios permiten identificar los eventos que realizan los objetos, i.e., los servicios de los objetos
3. Producir el modelo funcional y usarlo para definir operaciones de las clases: El modelo funcional describe las operaciones que toman lugar en el sistema.

• Especifica cómo computar los valores de salida a partir de los valores de la entrada.
• No considerar los aspectos de control, usar DFD.
• En OO las operaciones se realizan sobre objetos.
• Los transformadores del DFD representan esas operaciones.
4. Identificar las clases internas y sus operaciones:Evaluar críticamente cada clase para ver si es necesaria en su forma actual, puede que algunas clases innecesarias para la implementación se descarten.
• Considerar luego las implementaciones de las operaciones de cada clase.
• Puede que necesiten operaciones de más bajo nivel sobre clases auxiliares más simples. Ejemplos: árboles,
• Estas clases se denominan clases contenedoras.
5. Optimizar y empaquetar: Agregar asociaciones redundantes, con el fin de optimizar acceso a datos.
• Guardar atributos derivados, con el fin de evitar cálculos complejos repetidos, asegurar consistencia.
• Usar tipos genéricos, permite reusabilidad de código. Ej.: lista de ... , BTree de ...
• Ajustar la herencia,

Métricas (~)

Las Métricas son distintas formas con las que se puede evaluar los diseños, sirve para proveer una evaluación cuantitativa de diseño.

Métodos Pesados por Clase (WMC):

- La complejidad de la clase depende de la **cantidad de métodos en la misma y su complejidad**.
- Sean M_1 ... M_n los métodos de la clase C en consideración.
- Sea $C(M_i)$ la complejidad del método i. Ejemplo: la longitud estimada, complejidad de la interfaz, complejidad del flujo de datos, etcétera.
- Luego: $WMC = \sum_{i=1}^{n} C(M_i)$
- Si WMC es alto => la clase es más propensa a errores.

Profundidad del árbol de herencia (DIT)

Mide el nivel de profundidad de una clase en la jerarquía de herencia.
"Es la longitud del camino de la raíz a la clase C" "Si está muy debajo en la jerarquía dificulta la predicción de su comportamiento" Significativo en detección de clases propensas a errores:
Mayor DIT => Mayor probabilidad de error en esa clase.

Cantidad de hijos (NOC)

Mide la cantidad de subclases inmediatas de C. Mayor NOC => Mayor Reúso
También da la idea de la influencia directa de la clase C sobre otros elementos de diseño:
Mayor influencia => Mayor importancia en la corrección del diseño de esta clase.+

Acoplamiento entre clases (CBC)

Mide la cantidad de clases a las cuales una clase está acoplada.
"Dos clases están acopladas si los métodos de una usan métodos o atributos de la otra."
Menor acoplamiento de una clase => Mayor independencia de la clase => Más fácil de modificar.
Mayor CBC => Mayor probabilidad de error en esa clase.

Respuesta para una clase (RFC)

Mide el grado de conexión de los métodos de una clase con otras clases.
RFC = Cantidad de métodos que pueden ser invocados como respuesta de un mensaje recibido por un objeto de la clase. RFC es muy significativo en la predicción de clases propensas a errores.

5. Diseño Detallado (~)

"Se encarga de especificar la lógica de los distintos módulos especificados en el diseño del sistema."
No existen procedimientos claros para hacerlo: **Heurísticas y métodos.**
El método más común es el refinamiento paso a paso. Para eso necesitamos un lenguaje que sea detallado (como un Lenguaje Formal) pero también llevadero y comprensible (como un Lenguaje Normal).
En PDL, el diseño puede expresarse en el nivel de detalle más adecuado para el problema.

Process Design Language (PDL)

"Captura la lógica completa del procedimiento pero revela pocos detalles de implementación."
Tiene la sintaxis externa de un lenguaje de programación estructurado.
Es posible hacer cierto procesamiento automatizado sobre PDL. Pero el vocabulario es de lenguaje natural.
Utilidad: Puede usarse para especificar el diseño completo (desde la arquitectura hasta la lógica).
Permite un enfoque de refinamientos sucesivos.