

INGENIERÍA DEL SOFTWARE

PARCIAL I

INTRODUCCIÓN (cap. 1)

Dominio del problema

"Software de nivel industrial", es decir, aquel que no solo resuelve problemas a ciertos usuarios, sino también del que podrían depender grandes sistemas o negocios y cuya falla podría significar pérdida directa o indirectamente.

Software de nivel industrial (industrial strength software)

actividades importantes requieren de su correcto funcionamiento

debe ser de alta calidad respecto a propiedades como confiabilidad, usabilidad, portabilidad...

debe ser fuertemente testeado

su desarrollo debe dividirse en fases, cada una con su respectiva salidas

documentación, estándares, procesos...

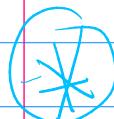
requerimientos detallados de lo que debe hacer el software

más grande y complejo

requiere más esfuerzo (10 veces más aproximadamente)

! La ingeniería del software se enfoca en cómo construir estos sistemas.

no solo en el desarrollo de los programas, sino también con todo lo otro que constituye el software.



El software es CARO.

es laboralmente intensivo

Su mayor costo es la mano de obra
gralmente medida en persona-mes y
Su productividad en KLOC por persona-mes

Optimizar el proceso de desarrollo para abaratar este costo.

A pesar del avance de la IS, sigue siendo un "área débil"

- Muchos proyectos desbocados (presupuesto y costo fuera de control)
- Mucho software poco confiable

Resistir a las

comportamientos inesperados (fallas)

mecánicas ó eléctricas, sino que son consecuencia de errores (bugs) introducidos durante el desarrollo. Esta falla siempre existió, solo que se manifiesta tarde.

Este tipo de software, una vez entregado, requiere mantenimiento (puede costar más que el desarrollo)

correchivo

corregir errores residuales
(update)

adaptativo

mejorar el funcionamiento y adaptarlo a los cambios del entorno.
(upgrades)

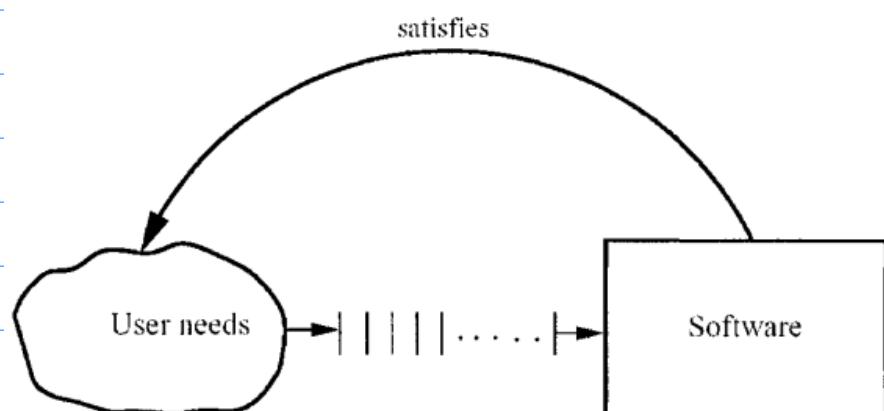
esto requiere comprensión del software existente, de los efectos del cambio, realizar los cambios y testearlos

Desafíos de la ingeniería del Software

aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

* Permite regular el proceso y la posibilidad de predecirlo

El problema fundamental con el que lida la IS es la satisfacción de las necesidades del usuario a través del software



Los desafíos de la IS son los factores que afectan al enfoque elegido para resolver este problema. Son las fuerzas principales que manejan el progreso y desarrollo en ese campo.

ESCALA:

- Debe considerarse el tamaño del Sistema a desarrollar
- en proyectos pequeños podemos usar métodos informales para el desarrollo y mantenimiento, en proyectos grandes hay que formalizar sí o sí
- los métodos de la IS deben ser adaptables respecto a respuesta y rendimiento del sistema a medida que aumentan o disminuyen los usuarios o requerimientos del mismo

CALIDAD Y PRODUCTIVIDAD (C&P):

La IS es impulsada por tres grandes factores: costo, tiempo (schedule) y calidad.

dominado por la mano de obra medida en RH

Se busca reducir el "time to market"

Productividad capta ambos
mayor productividad \Rightarrow menor costo y/o menor tiempo

Por otro lado, generar software de alta calidad es un objetivo fundamental, sin embargo, este concepto es un poco difícil de definir.

Según el estándar ISO, la calidad del software se compone de seis principales atributos

funcionalidad: capacidad de proveer funciones que cumplen las necesidades establecidas o implicadas.

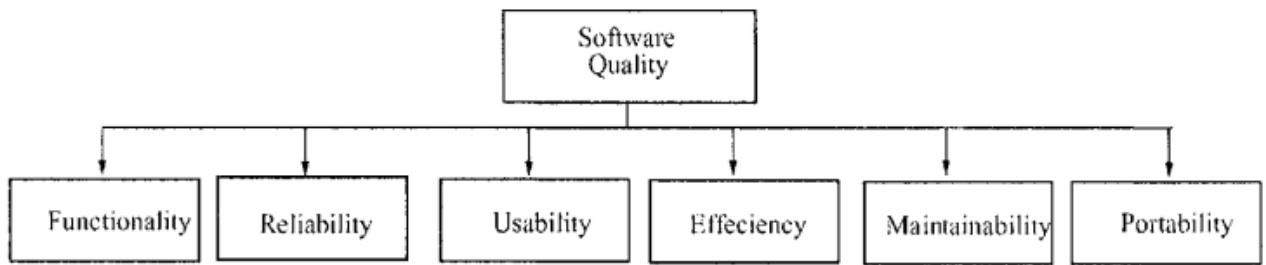
confiabilidad: capacidad de realizar funciones requeridas bajo las condiciones establecidas durante un tiempo específico.

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 ese propósito en el producto.



Dos importantes consecuencias de tener muchas dimensiones de calidad:

- El software no puede reducirse a un solo número.
- El concepto de calidad es específico al proyecto. El objetivo de calidad debe especificarse de antemano y el desarrollo deberá revisar por este objetivo.

Confiabilidad

suele ser el criterio



+ fallas \Rightarrow - confiable

principal, está inversamente relacionada con la probabilidad de falla

- ¿Qué es un defecto dependiente del software!

CONSISTENCIA y REPETIBILIDAD:

Un desafío clave de la IS es como garantizar que los resultados exitosos quedan rellenarse para mantener cierto grado de consistencia en la C&P.

Por lo tanto, un objetivo de la IS es la sucesiva producción de sistemas de alta C&P.

La consistencia permite predecir el resultado de un proyecto con razonable certeza. Sin ella es difícil estimar costos.

CAMBIO:

La IS debe preparar el software para que este sea fácilmente modificable, para así adaptarse a los habituales cambios de las empresas/instituciones.

Alta C&P pero baja adaptabilidad es poco útil

entonces...

El desafío de la IS es consistentemente desarrollar software de alta calidad y productividad para problemas de gran escala que se adapten a los cambios.

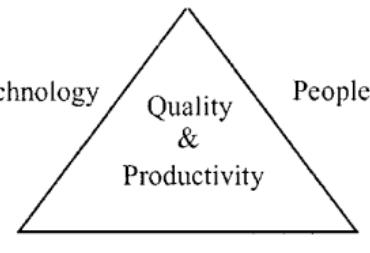
Enfoque de la ingeniería del Software

Como vimos, el principal objetivo de los IS es desarrollar software de alta calidad y productividad.

Las tres principales fuerzas que gobernan la C&P son

- las personas
- los procesos
- las tecnologías

el foco, en IS,
está acá.



triángulo de hierro

El enfoque de los IS está en seguir el proceso de desarrollo del producto desarrollado (final). Esto lo distingue de otras disciplinas.

Proceso adecuado \Rightarrow gran C&P

diseñarlo y controlarlo es el desafío clave de la IS //

Proceso de desarrollo en fases

El proceso de desarrollo consiste en varias **fases**

- cada una tiene una salida definida
- cada una maneja un aspecto del desarrollo
- permiten verificar calidad y progreso en momentos definidos del desarrollo (al final de c/ fase)

En general las fases consisten en:

- 1- Análisis de requisitos y especificación
- 2- Arquitectura
- 3- Diseño
- 4- Codificación
- 5- Testing
- 6- Entrega e instalación

Los enfoques sistemáticos requieren que cada etapa se realice rigurosamente

* Si este proceso en fases no es bien administrado, el objetivo de C&P no podrá ser alcanzado. Para esto es necesario un planeamiento y para ello son importantes las métricas y las medidas.

Resumiendo

- Dominio del problema de la IS es software industrial (no solo los programas) sino tmb su documentación
- Esse software es curto, complejo, requiere mantenimiento y ser de alta calidad.
- La IS es la disciplina q se encarga de proveer métodos y procedimientos para el desarrollo sistemático de software industrial.
Los desafios con los q se enfrenta a la hora de hacerlo son la escala del software, la calidad y productividad, la consistencia y reutilización y el cambio.
- El enfoque q toma la IS a la hora de cumplir sus objetivos es la separación del proceso de desarrollo y el producto desarrollado. La IS se enfoca en los procesos, si estos son de alta C&P, entonces el producto tmb lo será.
- Para afrontar estos desafios la IS trabaja con un proceso dividido en fases.
- Otro enfoque clave es la buena administración de este proceso y sus fases usando métricas.

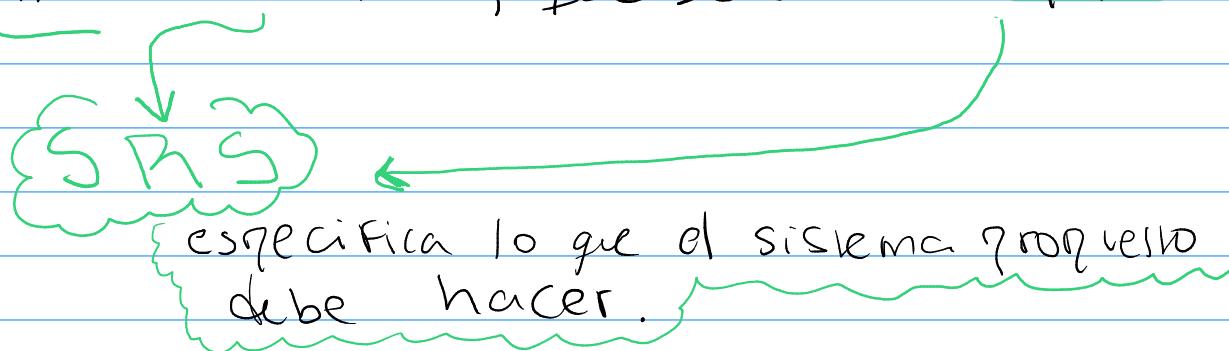
ANÁLISIS Y ESPECIFICACIÓN DE REQUERIMIENTOS (cap. 3)

Mientras más complejo es el sistema, más difícil es comprender y especificar sus requerimientos.

Requiere interacción con los clientes, la entrada de esta fase del proceso son las ideas que tienen en su cabeza.

Por esto otra fase no es automatizable.

Esta fase transforma esta entrada (que es informal, imprecisa e incompleta) en un documento formal, que será la salida.



Requerimientos del software

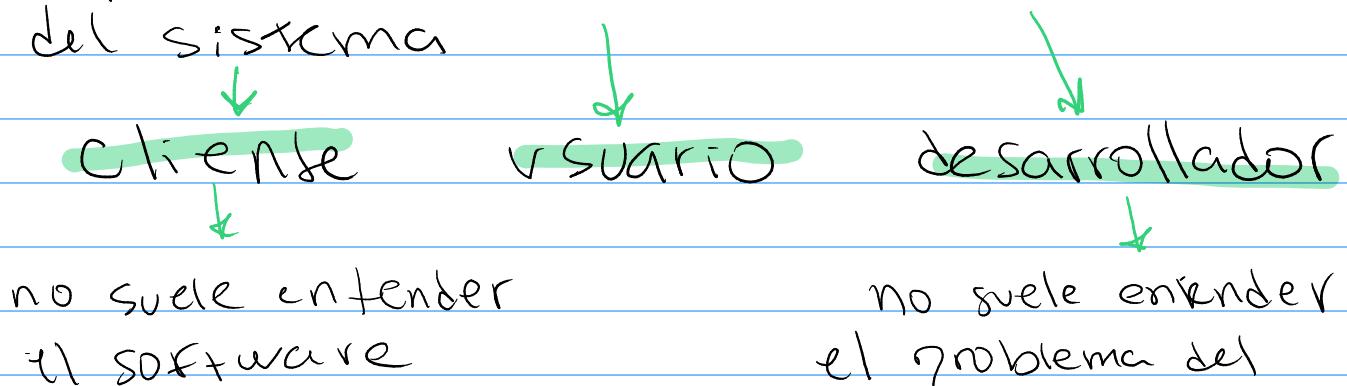
Características y funcionalidades que el sistema debe tener.

La SRS (Software Requirements Specification) describe qué debe tener el sistema progresivo sin describir cómo lo hará. Establece las bases para el acuerdo entre cliente/usuario y quien suministra el software.

Algunas limitaciones a la hora de producir la SRS

- Visualizar algo que no existe (futuro sistema)
- No conocemos sus capacidades, menos sus necesidades
- Las necesidades de los usuarios cambian constantemente

Partes involucradas en la creación del sistema



brecha comunicacional

la SRS busca reducirla

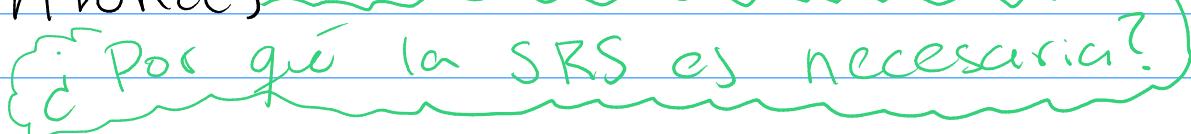
(mediante el cual las necesidades del cliente y usuario son especificadas al desarrollador)

La SRS provee una referencia para la validación del producto final, ayudar al cliente a determinar si el software cumple con los requisitos.

SRS de alta calidad es esencial para obtener un software de alta calidad.

Buena SRS reduce los costos de desarrollo, pues los errores que surgen de aquí son más caros de corregir a medida que progresa el proyecto.

Entonces,

 Por qué la SRS es necesaria?

- Reduce la brecha comunicacional entre cliente y desarrollador
- Asegura calidad en el producto final
- Reduce costos de desarrollo

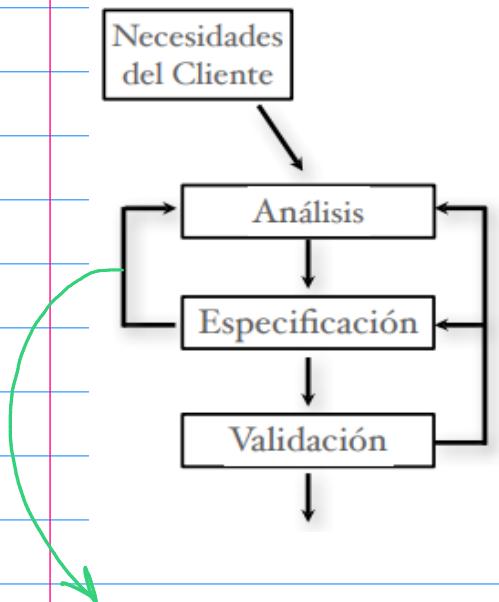
Proceso de requerimientos:

Secuencia de pasos que se necesita realizar para convertir las necesidades del cliente en SRS.

3 tareas básicas:

- 1- Análisis del problema: entender qué necesita proveer el sistema.
- 2- Especificación de los requerimientos: especificar los requerimientos en un documento.
- 3- Validación: asegurarse de lo especificado son, efectivamente, todos los requerimientos del software

Este proceso no es lineal



esta transición
es complicada

- hay superposición
- especificar puede ayudar al análisis
- validación puede conducir a más análisis y especificación
- diferentes objetivos
- especificación se fija en comportamiento externo
- el análisis recopila más info de la necesaria en la especificación.
- el análisis ayuda a comprender en lugar de asistir la especificación.

Análisis del Problema

Objetivo: lograr buena comprensión de las necesidades, requerimientos y restricciones del software

Para ello el analista no solo recopila info y la organiza sino que también actúa como consultor

rol pasivo

debe

comprender el funcionamiento de la organización, el cliente y los usuarios.

rol activo

El principio básico para hacerlo es particionar el problema (divide y vencerás)

Concejos de estudio y proyección son usados efectivamente

Se hace análisis detallado para el estudio posible del sistema.

Se analiza el sistema desde distintos puntos de vista.

Métodos para el análisis del problema:

Enfoque informal

- No hay metodología definida.
- Info obtenida a través del análisis, observación, interacción, etc.
- NO se construye modelo formal del sistema.
- Info directamente traducida de las ideas a la SRS.
- Útil en muchos contextos.

Modelado de Flujo de datos

(Se le conoce a menudo como técnica de análisis estructurado)

Se enfoca en las **funciones** que realiza el sistema, lo ve como una red de **transformadores de datos** por los cuales fluye la información

Para el modelado utiliza DFD's
(diagrama de flujo de datos)

Un DFD es una representación gráfica del flujo de datos a través del sistema.

lo ve como una función que transforma las entradas en las salidas deseadas

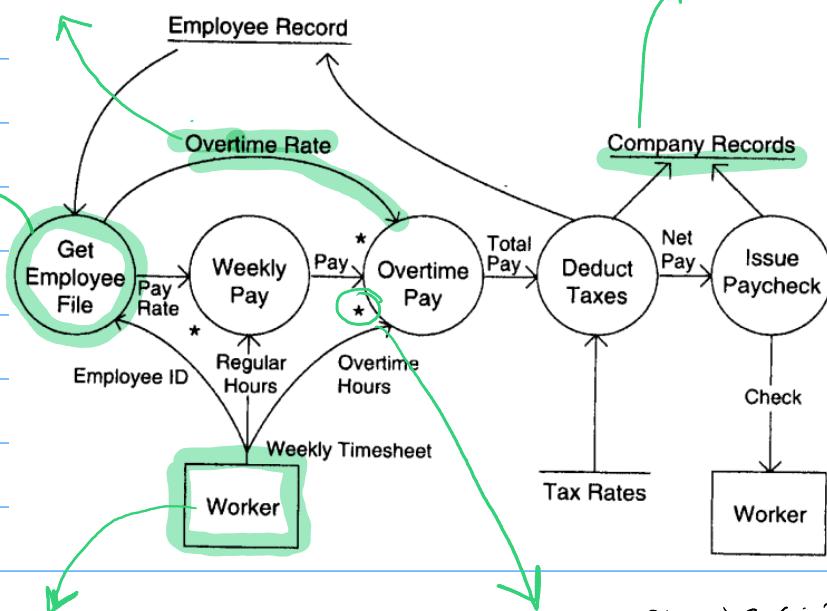
a través de procesos (burbujas, transformadores)

Captura cómo ocurre esa transformación a medida que los datos se mueven.

dato que fluye
de un proceso a otro
(sustitutivos)

almacén/base de
datos, archivos
externos

procesos/
transformadores
(representativo)
verboso



fuent/sumidero

originador/consumidor
de datos, está por
fuera del sistema

representación de
la necesidad de
múltiples entradas,
tmb existe el +

- Se enfoca en qué hacen los transformadores, no en cómo.
- Muestran solo entradas y salidas más significativas.
- No hay loops, ni condicionales, ni secuencialidad.
- No es un diagrama de control

Pueden existir DFD's en niveles, cuando un sistema es muy grande

Podemos definir con mayor precisión los datos fluyendo en el DFD usando diccionarios de datos, siempre con el DFD anterior

```

weekly timesheet =
    Employee.name +
    Employee.Id +
    [Regular.hours + Overtime.hours] *

pay_rate =
    [Hourly | daily | weekly] +
    Dollar.amount

Employee.name =
    Last + First + Middle.initial

Employee.Id =
    digit + digit + digit + digit
  
```



errores comunes

→ no etiquetar todo

los flujos de datos

→ flujos irrelevantes

omisiones e irrelevantes no

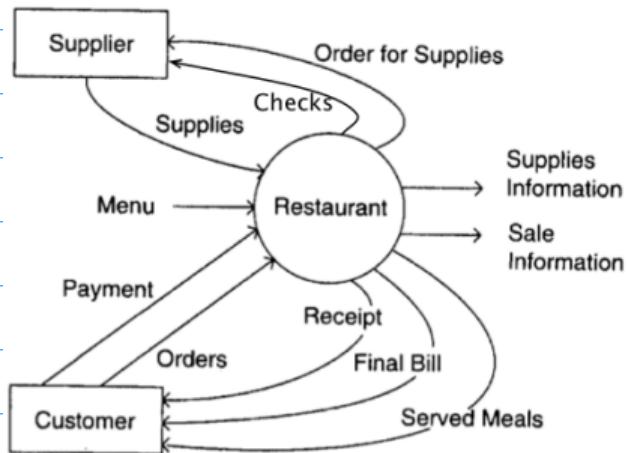
→ omisión de procesos

→ info de control

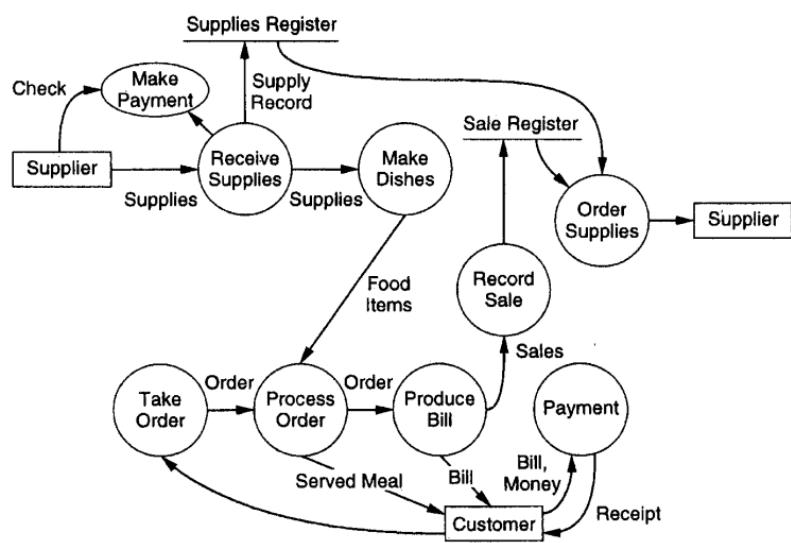
Veamos ahora ese modelo de flujo de datos como un método de análisis estructurado.

Pasos principales

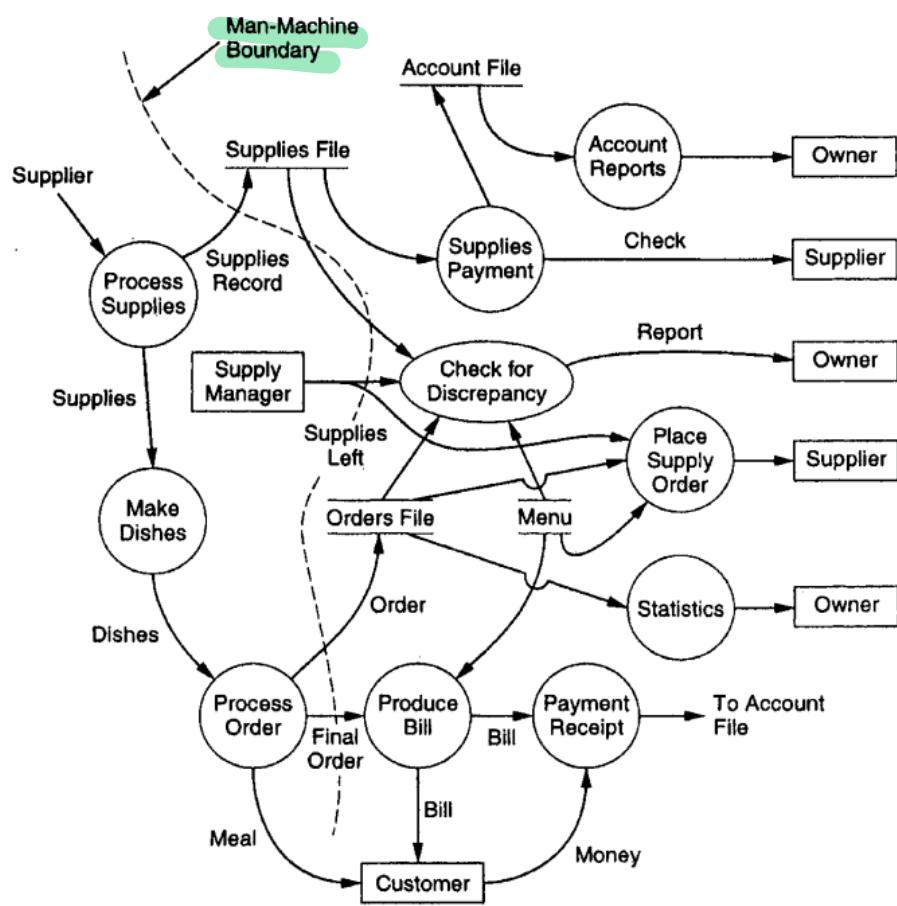
1-Dibujar diagrama de contexto:
 el sistema es tratado como un solo proceso, con sus respectivas entradas, salidas, fuentes, etc.



2-Dibujar DFD del sist. existente:
 Refinamiento del diagrama de contexto, DFD tal como lo vimos antes.
 Debe validarse con los usuarios haciendo una caminata a través de él.



3- Dibujar DFD sistema que se va a implementar en la frontera hombre-máquina
 se agreguen todos los procesos, ya sean automatizados o no
 Se establece cuáles procesos se automatizan o no
 Se muestra la interacción entre ambos.



Modelado orientado a objetos

Ventajas:

- Más fácil de construir y mantener
- Pasar de él al diseño orientado a objetos es simple
- Resistente / adaptable a cambios
(los objetos son más escasos que las funciones)

El sistema es visto como objetos que interactúan entre sí o con el usuario a través de los servicios que ceder uno provee.

Objeto => instancia de una clase

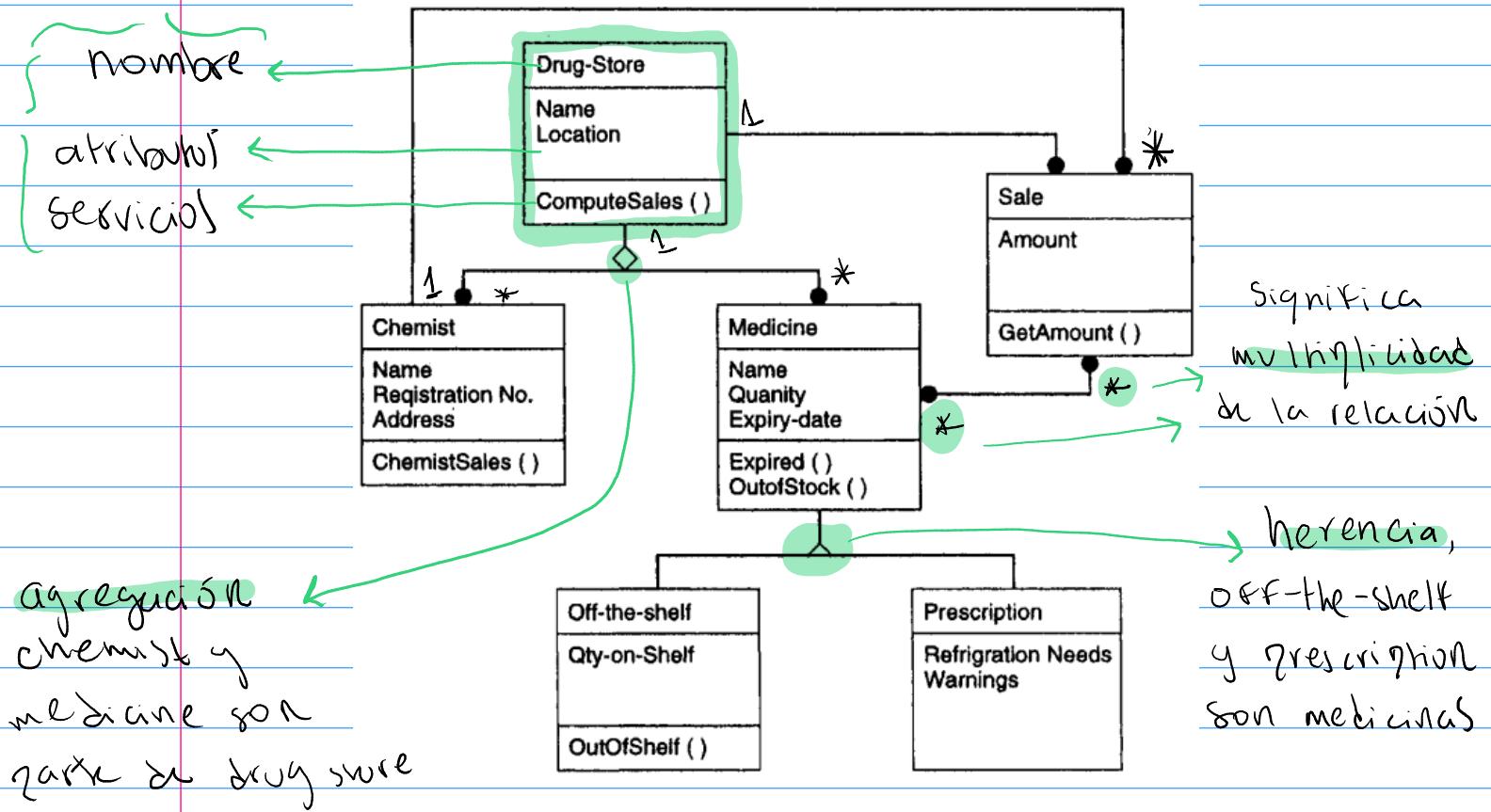
- Tendrán atributos (definen el estado)
- Se agrupan en su respectiva clase
- Provee servicios / realiza operaciones que son quienes permitirán ver o modificar el estado del objeto (están definidos por las clases)

Este modelo utiliza diagrama de clases

Representados por rectángulos



CLASE



Otras relaciones entre clases:

asociación: las clases sólo se relacionan entre sí A ————— B (A se relaciona con B)

composición: una clase es parte de otra y no puede vivir sin ella.

A ————— B (A es parte de B y no puede vivir sin B)

Prototipado

Se construye un sistema parcial prototípico que ayudar a visualizar cómo será el sistema final.

Ayuden a comprender mejor el problema y las necesidades.

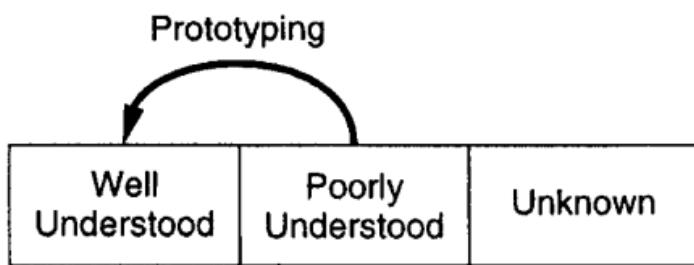
Dos enfoques:

Descartable: el prototipo se descarta luego de terminar la fase de requerimientos (mas adecuado para este tipo)

Evolucionario: se busca que el prototipo evolucione al sistema final.

Hay 3 tipos de requerimientos:

- Bien entendidos
 - Pobremente entendidos
 - No entendidos
- estos deben incorporarse en el prototipo



- críticos para diseñar) en estos debe enfocarse el prototipo
- no críticos para diseñar

Especificación de los requerimientos:

Salida: la SRS

Pequeños problemas \Rightarrow se hace una vez el análisis se completa.

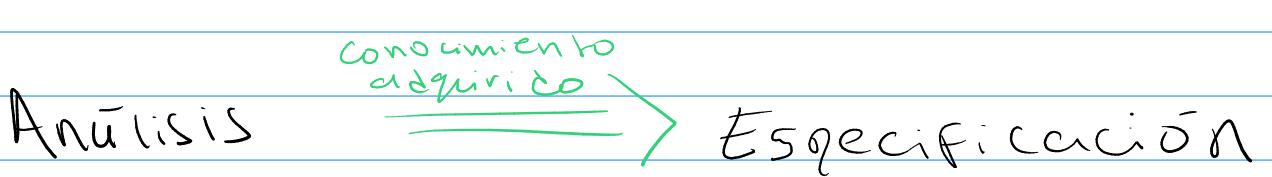
Grandes problemas \Rightarrow se hace concurrentemente con el análisis.

Modelado \rightarrow se enfoca en la estructura
SRS \rightarrow se enfoca en el comportamiento externo

interfaces de usuario, tratamiento de errores, reglamentos de desempeño, restricciones de diseño, estándares, etc. DEBEN estar en la SRS (no así en el modelado) 

\Rightarrow transición del modelo a SRS no es directa 

(no es una formalización del modelo)



Características de una SRS

Fuertemente
relacionadas

- correcta: todos los requerimientos en la SRS son requerimientos deseados por el cliente en el sistema final
- completa: todos los requerimientos deseados por el cliente están descritos (más difícil de lograr)
- no ambigua: para cada requerimiento existe un solo significado (esencial para verificabilidad, lenguajes formales ayudan a desambiguar)
- consistente: ningún requerimiento contradice a otro.
- verificable: existe, para cada requerimiento, un proceso efectivo que puede asegurar que el software final lo satisface.
- rastreable: se puede determinar el origen de cada requerimiento y cómo se relaciona con el software
- hacia adelante: qué elementos del software satisfacen el requerimiento
- hacia atrás: qué requerimiento satisface el elemento del software
- modificable: permite incorporar cambios fácilmente manteniendo complejidad y consistencia. (evitar redundancia)
- ordenada en aspectos de importancia y estabilidad: tener bien definido el orden de prioridades en la construcción de la SRS.

Componentes de una SRS

Una SRS debe especificar requerimientos sobre

funcionalidad: mayor parte de la especificación

- toda funcionalidad del sistema
- qué salidas producen las entradas y las reacciones entre ellas
- todas las operaciones que el syst. debe realizar

→ entradas válidas, verificaciones de entradas y salidas, comportamiento ante situaciones anormales y normales pero con imposibilidad de operar

Desempeño: restricciones en el desempeño del sistema

→ Req. dinámicos: restricciones sobre la ejecución (tiempo de ejec., tiempo esperado de terminación de una operación, operaciones por unidad de tiempo, etc.). Se especifican (rangos aceptable), en caso) normal) y extremo)

→ Req. Estáticos: o de capacidad. No restringen la ejecución (cant. usuarios admitido), cant. archivos o procesos, etc.)

Todos se especifican en términos medibles verificables

Diseño: Queden existir factores en el contexto del cliente que restrinjan las elecciones de diseño (ajustarse a estándares, limitaciones de hardware, etc).

Interfaces externas: interacciones del software con gente, hardware y otros software.

El lenguaje de especificación debe facilitar la producción de la SRS con las características deseadas, debe ser fácil de aprender.

Puede ser **natural** (es ambiguo, se apoya en documentos estructurados para reducir error) ó **formal** (preciso pero difícil de aprender)

Se usa en **proyectos específicos o sistemas críticos**

El alcance de la especificación implica qué abarca y qué no abarca el proyecto, prioridades, tiempos de entrega, criterios de aceptación, límites presupuestarios, entre otros).

La SRS debe ser **estructurada** y **organizada** para asegurar la inclusión de todos los requerimientos del sistema. Para ello es necesario organizarla en **secciones** y **subsecciones**.

Usar un estándar a la hora de estructurarla ayuda a la comprendión por parte de otros y también a la completitud.

Especificación funcional con casos de uso.

Los casos de uso especifican la funcionalidad del sistema capturando su comportamiento en forma de interacciones del usuario con el sistema.

Se enfocan en especificación de la funcionalidad, pero quedan usarse durante el análisis.

Son adecuados para sistemas interactivos.

Conceptos básicos:

- Forma básica \Rightarrow textual.
- actor: persona o sistema que interactúa con el sistema propuesto.
- actor primario: actor principal que inicia el caso de uso.
 → debe satisfacer su objetivo
La ejecución real puede llevar a cabo otra persona o sistema en su representación.
- escenario: conjunto de acciones que se llevan a cabo para alcanzar el objetivo.

- **escenario exitoso principal:** todo funciona normalmente y se alcanza el objetivo.
- **escenario alternativo:** el objetivo es alcanzado pero no "normalmente"
- **escenario excepcional:** algo sale mal y no puede alcanzarse el objetivo.

El caso de uso será una colección de muchos escenarios.

Pueden organizarse jerárquicamente (un caso de uso usa otro caso de uso)

Formato:

Formato que usaremos para cada caso de uso

- Caso de uso #: « nombre del CU »
- Actor primario: « nombre del AP »
- Precondición: « descripción precondición »
- Escenario exitoso principal:

 « paso 1 »

 « paso n »

- Escenarios excepcionales:

 « excepción 1 »

 « excepción n »

Los casos de uso se numeran para referencias posteriores

El nombre del caso de uso especifica el objetivo del actor primario

El actor primario puede ser una persona o un sistema

La precondición describe lo que el sistema debe asegurar antes de iniciar el caso de uso

Validación de los requerimientos:

Es importante detectar errores en esta etapa ya que, mientras mas tardemos en encontrarlos, mas nos costará arreglarlos.

El objetivo básico de validar la SRS es asegurar que essa refleja de forma clara y precisa las necesidades del cliente.

Los errores más comunes a la hora de especificar los requisitos son:

→ omisión: algún requerimiento no figura en la SRS

→ inconsistencia: hay requerimientos que se contradicen

→ hechos incorrectos: algún requerimiento no es correcto

→ ambigüedad: hay requerimientos con múltiples significados.

La validación debe encargarse de cubrir estos errores.

Se hace una review de la SRS entre autor, cliente, usuario y desarrollador.

Las checklists son muy útiles, ejemplo

- ¿Se definieron todos los recursos de hardware?
- ¿Se especificaron los tiempos de respuesta de las funciones?
- ¿Se definió todo el hardware, el software externo y las interfaces de datos?
- ¿Se especificaron todas las funciones requeridas por el cliente?
- ¿Son testeables todos los requerimientos?
- ¿Se definió el estado inicial del sistema?
- ¿Se especificaron todas las respuestas a las condiciones excepcionales?
- ¿Los requerimientos contienen restricciones que pueda controlar el diseñador?
- ¿Se especifican modificaciones futuras posibles?

Existen herramientas para el modelado y análisis que deben ser verificadas por herramientas específicas.

↳ Se enfocan en chequear consistencia y complejidad.

Métricas

Proveen información cuantitativa para la administración del proceso. Esta info podrá ser usada para controlar el proceso de desarrollo.

Para poder estimar costos y tiempo necesitamos de estas métricas para "medir" el esfuerzo que demandaría el proyecto.

Principal factor → el tamaño

Una métrica es importante solo si es útil para el seguimiento de costos, tiempo y calidad.

algunas métricas:

Punto Función: Similar a LOC, define el tamaño en términos de funcionalidad, este se calcula con el número de funciones implementadas, de inputs, de outputs, etc. Solo depende de las capacidades del sistema.

Según el enfoque del punto función, hay cinco parámetros que logran capturar la funcionalidad total del sistema:

- Entradas externas control de info
- Salidas externas ↗ usuarios
- Archivos lógicos internos

compartidos entre archivos de interfaz externa
aplicaciones • transacciones externas (querier)

Una vez que cada función se la clasifica con alguno de estos parámetros, se la vuelve a clasificar en

- simple
- promedio
- compleja

La fórmula para calcular el punto función (no ajustado) es:

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

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

con w_{ij} el peso de cada función (cuadro siguiente) y C_{ij} la cantidad de funciones tipo i con complejidad j

Una vez que se obtiene el UPF, se lo ajusta a la complejidad del entorno. Se evalúan 14 características en una escala

del 0 al 5 y luego se calcula el factor de ajuste de complejidad

$$CAF = 0,65 + 0,01 \sum_{i=1}^{14} q_i$$

con q_i la evaluación en

Ahora sí,

$$PF = CAF * UFP$$

dicha escala de c1 característica

Métricas de calidad:

- directas: evalúan la calidad del documento estimando el valor de los atributos de calidad de la SRS.

- indirectas: evalúan la efectividad de las métricas del control de calidad usadas en el proceso en la fase de requerimientos

c_i : n de errores encontrados

ARQUITECTURA DEL (cap. 4) SOFTWARE

Un sistema complejo está compuesto por **subsistemas** que interactúan entre sí.

Cuando diseñamos un sistema, el enfoque lógico es **identificar** estos **subsistemas**, sus **interfaces** y **cómo interactúan** entre sí. Ese es el **objetivo** de la arquitectura.

Definición: estructura o estructuras del (más aceptada) sistema, compuesto por los elementos del software, las propiedades visibles externas de esos elementos y las relaciones entre ellos.

- Da una visión de muy alto nivel de las partes y sus relaciones.
- Cada parte puede comprenderse de forma independiente.
- El sistema se describe en términos de ellas (y sus relaciones).
- Solo nos interesan las propiedades de otros elementos, asumen que existir y son necesarias para establecer relaciones. Podrían ser propiedades de funcionalidad o servicios de el componente brindar.
- El cómo de estas propiedades no interesa.
- Para saber si la arquitectura es buena o no es necesario analizarla

¿Para qué es buena la arquitectura?
(rol)

- **Entendimiento y comunicación:** Presentar el sistema a un nivel de abstracción alto, compuesto por subsistemas y relaciones, esconde complejidades y facilita su entendimiento. Esto a su vez, facilita la comunicación entre todos los actores involucrados (cliente, usuario), desarrolladores y autores. Facilita, por lo tanto, la negociación y acuerdos.

Puede usarse para comprender sistemas existentes.

- **Reuso:** Principal técnica para incrementar la productividad.

La arquitectura debe elegirse de forma que los componentes se reusen, encajen y, junto a otros componentes, provean funcionalidades necesarias.

La decisión de usar estos componentes se toma a la hora de diseñar la arquitectura.

- **Construcción y evolución:** La división provista por la arquitectura servirá para guiar el desarrollo del sistema (qué partes hay que construir? Cuáles ya están construidas?)

Como las partes son independientes entre sí (relativamente) ayuda a asignar grupos de trabajo. A medida que el sistema evoluciona, la

arquitectura ayuda a decidir qué partes necesitan cambiarse, cómo impactará esto en los otros componentes.

- **Análisis:** la arquitectura permite el análisis del sistema a otro nivel. Propiedades como desempeño y confiabilidad quedan determinadas aquí, permitiendo considerar alternativas para encontrar los niveles de satisfacción deseados.

Requiere descripción precisa de la arqu y de las propiedades de los componentes.

Vistas de la arquitectura

- No hay una única arquitectura de un sistema.
- Una vista representa el sistema como compuesto por elementos y relaciones entre ellos.
- Hay distintas vistas para un sistema, que exponen diferentes propiedades y atributos.
- Qué elementos son usados por una vista dependen de lo qe la vista quiera resaltar.
- Reducen la complejidad a la qe se enfrenta el "lector", ya qe se enfocan solo en algunos aspectos del sistema.
- La mayoría de las vistas que vemos pertenecen a alguna de estos tres tipos:
 - Módulos
 - Componentes y conectores
 - Asignación de Recursos

+ Vista de módulos: colección de unidades de código, cada una implementa una funcionalidad del sistema.

elemento principal => módulos (clases, paquetes, funciones, procedimientos, etc.)

(clases, paquetes, funciones, procedimientos, etc.)

Las relaciones entre módulos son también code-based.
("parte de", "usa a", "depende de", etc.)

+ Vista de componentes y conectores:

elemento principal => unidades de ejecución

(objetos, colecciones de objetos, procesos, etc.)

Los componentes (unidades de ejecución) necesitan interactuar con otros => conectores.

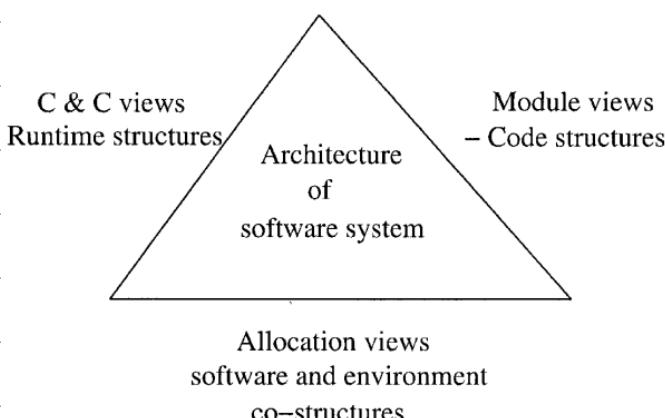
+ Vista de asignación de recursos:

cómo las distintas unidades del sistema asignan recursos.

especifica las relaciones entre los elementos del software y elementos del entorno.

(propiedades estructurales como qué proceso corre en qué procesador, etc.)

Todas estas vistas se relacionan entre sí.



Las vistas necesarias para un proyecto dependen de él.

Largo y complejo => distintas

vistas

Nos enfocamos en la vista de

Componentes y conectores

elementos computacionales

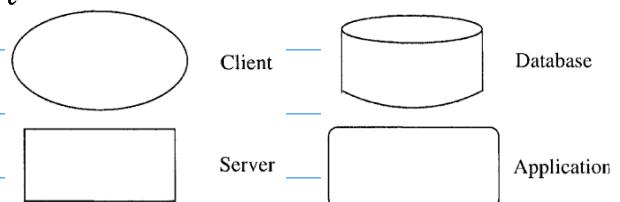
o de almacenamiento de datos.

Cada uno tiene un nombre que representa su rol y le provee una identidad.

Tmb tienen su tipo,

se representan con símbolo).

Usan interfaces o protocolos para comunicarse con otras.



mecanismos de interacción

entre las componentes

Describen el medio en el cual la interacción toma lugar.

Pueden proveerse por medio del entorno de ejecución (llamadas a procedimientos)

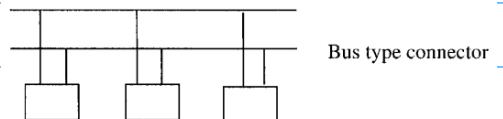
o ser mecanismos más complejos (que los, protocolos)

Tienen nombre (naturaleza de la interacción), tipo (binario, n-ario, unidireccional)

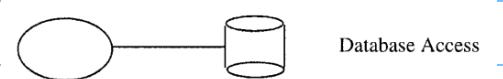
Esta vista define las componentes y cómo se conectan a través de conectores.

Describe una estructura en ejecución del sistema:

qué componentes existen y cómo interactúan entre ellos en tiempo de ejecución.



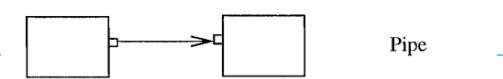
Bus type connector



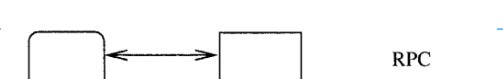
Database Access



Request – Reply



Pipe

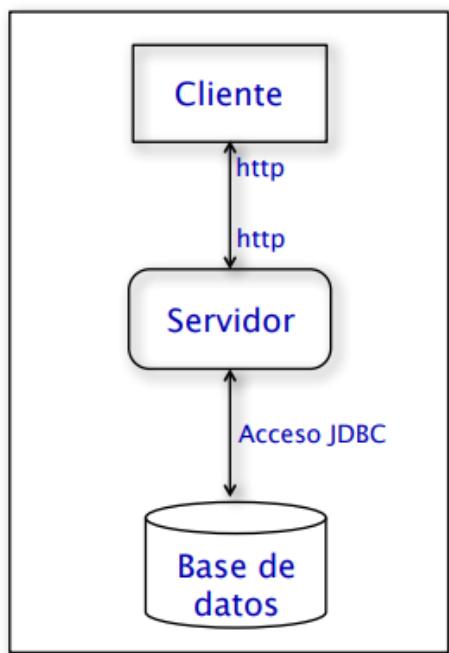


RPC

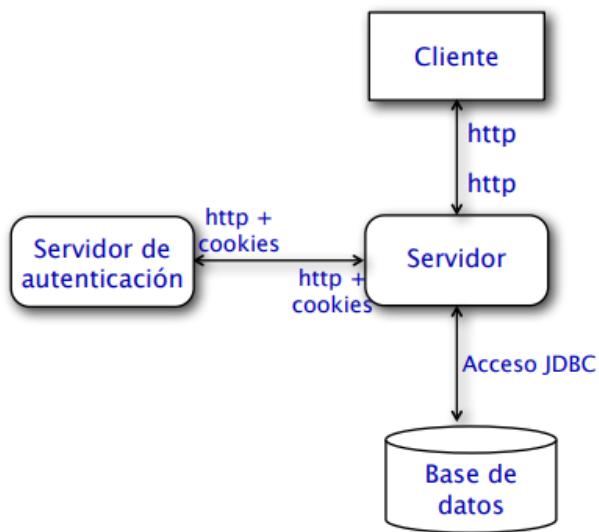
Ejemplo: sistema de encuestas a alumnos de la facultad.

multiple-choice, se muestran los resultados al terminarla.

Se construye sobre la web

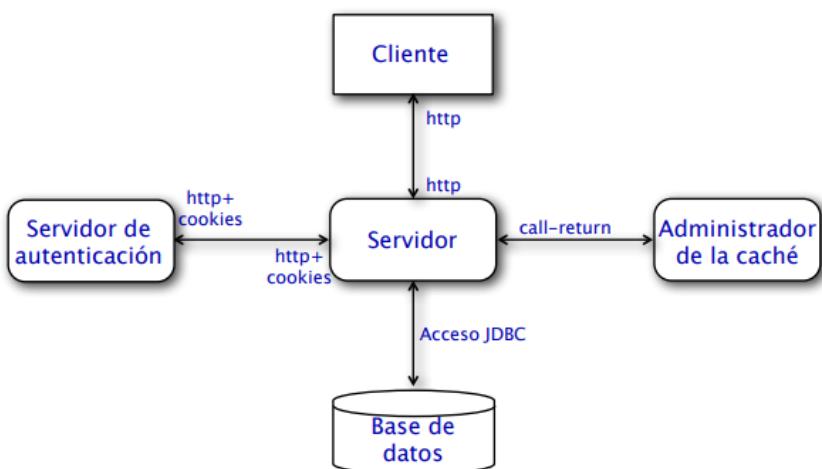


Versión 0; debería ofrecer seguridad, que no todos pueden acceder a la encuesta.



La base de datos se cae con frecuencia, los resultados quedan ser parciales, un poco desactualizados (5 encuestas)

extensión I (agrega seguridad)



extensión II (incrementa desempeño y configurabilidad)

Elegir la arquitectura correcta ayuda a construir buenos sistemas.

Estilos arquitectónicos para la Vista C&C:

Diferentes sistemas tienen diferentes estructuras (aún si son la misma vista).

Hay estructuras generales que son útiles para la arquitectura de cierta clase de problemas

→ estilos arquitectónicos

Un estilo define una familia de estructuras que satisfacen las restricciones de ese estilo.

Proveen ideas para crear vistas.

Pueden combinarse para encontrar la arquitectura deseada.

Tubos y Filtros (hose and filter) → adecuado para sistemas que realizan fundamentalmente transformaciones de datos.

- Se recibe el input y el objetivo del sistema es producir un output, habiendo transformado ese input.
- Esta transformación deseada se alcanza usando una red de transformaciones más pequeñas, organizándolas de manera que se obtengan.
- Sólo un tipo de componente => Filtros
- Sólo un tipo de conector => Tubos

Un filtro transforma y pasa la info a otro filtro a través de un tubo.

Restricciones del estilo (limitaciones/reglas):

Filtro

→ entidad independiente y asíncrona
→ se limita a producir y consumir datos.

→ debe hacer "buffering" y sincronización para asegurar su correcto funcionamiento.

→ debe trabajar sin conocer la identidad de los otros filtros.

→ Si el sistema es puramente de tubos y filtros, cada filtro debe tener su propio hijo de control

Tubo

→ canal unidireccional

→ sólo conecta dos componentes

→ conecta un par de salidas de un filtro a un par de entradas de otro.

Ejemplo de datos:

compartidos

Repositorio de datos

→ provee almacenamiento permanente confiable

Dos tipos de componentes

Usuario de datos

→ acceden a los datos en el repositorio, realizan acciones y

luego vuelven a colocar los datos en el repositorio

la comunicación entre ellos solo se hace a través del repositorio

Sólo un tipo de conexión => lectura/escritura

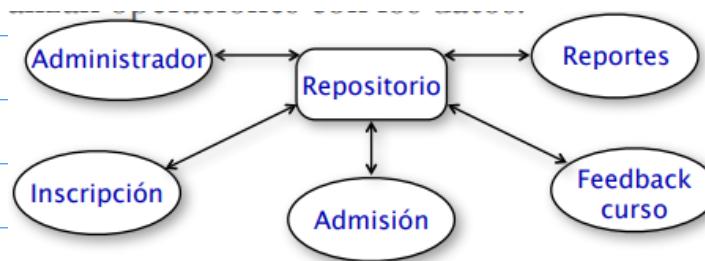
- Dos variantes → estilo zigzag

estilo repositorio

el re~~go~~o es pasivo

cuando se agregan/modifican datos en el re~~go~~, se le avisa a todos los usuarios
(re~~go~~o activo)

ejemplo: sistema de inscripción de alumnos



Ciente - Servidor :

- Dos hijos de componentes

Ciente

Servidor

- Ciente sólo se comunican con el servidor, no con otros clientes.

Comunicación siempre iniciada por el cliente



- Sólo un hilo de conexión => solicitud/resposta (request/reply)

- Cliente y servidor residen en distintas máquinas

• Este estilo en general tiene forma de estructura multi-nivel: cliente solicita, servidor solicita a otro servidor, servidor 2 responde, etc. El servidor tmb actua como cliente.

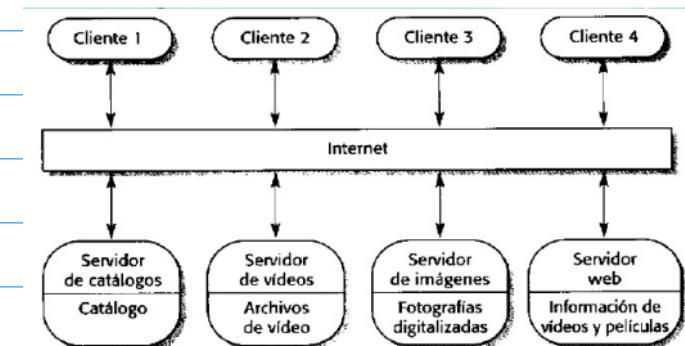
Ejemplo clásico: 3 niveles

→ Nivel de cliente: clientes

→ Nivel intermedio: lógica de negocios del servidor

→ Nivel de base de datos: reside la info

Ejemplo: sistema de usuarios web de una biblioteca de imágenes



Otros estilos

publish-subscribe

dos componentes

los que los que se
publican suscriben

el conector es

publicar/suscribir

cada vez que se publica un

evento, se invoca las com-
ponentes suscritas a ese evento.

peer-to-peer
(object-oriented)

solo un tipo de
componente

el componente 1 e

pede pedir servicio

a otra

procesos que se
comunican

los componentes son
los procesos

y comunican entre

ellos a través de

mensajes

Documentación del diseño arquitectónico:

Si bien los diagramas son un medio adecuado para discutir y crear diseños, no son suficientes para documentarlos. Esta documentación debe especificar precisamente las vistas y las relaciones entre ellas.

- Organización del documento:
1. Contexto del sistema y la arquitectura
 2. Descripción de las vistas de la arqu.
 - a - Presentación principal de la vista.
 - b - Catálogo de elementos
 - c - Fundamento de la arquitectura
 - d - Compartimiento
 - e - Otra info
 - 3 - Documentación transversal a las vistas

Arquitectura y diseño:

Ambos dividen el sistema en partes y especifican cómo se organizan. ¿Cuál es la diferencia?

La arquitectura es un diseño de muy alto nivel que se enfoca en las componentes principales.

"Diseño" se enfoca en los módulos que se transformarán en el código de esos componentes.

Podemos decir que el diseño provee la visión de módulos del sistema.

Arquitecto y diseñador son quienes deciden donde acaba e inicia cada tarea.

Arquitectura solo necesita identificar los papeles necesarios para evaluar las propiedades deseables; no considera la estructura interna de los papeles; impone restricciones sobre elecciones que pueden realizarse en el diseño.

Preservación de la integridad de la arquitectura:

La arquitectura impone restricciones que deben preservarse en la implementación, inclusive. Para que esto tenga sentido debe acompañar el diseño y desarrollo del sistema.

Ejemplo Falote pag. 185

Evaluación de las arquitecturas:

La arquitectura tiene impacto en los atributos no funcionales del sistema (atributos de calidad). Por eso deben evaluarse en la arquitectura 7 razones.

- **¿Cómo?** • Con técnicas formales (redes de colas, model checkers, lenguajes de especificación, etc.)
- metodologías rigurosas

Una técnica más elaborada y formal para la evaluación de la arquitectura es **ATAM** (architectural tradeoff analysis method).

ATAM (architectural tradeoff analysis method)

Además de analizar la arquitectura para ciertas propiedades, también ayuda a identificar dependencias entre propiedades y hacer un análisis de compromisos.

Nos enfocaremos en el análisis

● Pasos de ATAM básicos:

1 - Recolección de escenarios:

Los escenarios describen interacciones del sistema.

Elegir sólo los de interés para el análisis, incluir escenarios excepcionales solo si son importantes.

2 - Recolección de requerimientos y/o restricciones:

definir lo que se espera del sistema en dichos escenarios

se deben especificar los niveles deseados para el atributo de interés

3 - Describir vistas arquitectónicas:

las vistas se serán evaluadas son recolectadas

diferentes vistas pueden usarse para distintos análisis.

4 - Análisis específico a cada atributo:

se analizan las vistas bajo distintos escenarios separadamente para cada atributo de interés.

se comparan con los requeridos base para la elección de una arquitectura u otra, o la modificación de la propuesta.

Se puede usar cualquier técnica o modo de lenguaje

5 - Identificar quénos Sensibles y de Compromiso:

• Sensibilidad: impacto que tiene un elemento en un atributo, elementos de mayor impacto => quénos de sensibilidad

● Compromiso: elementos que son
junto de sensibilidad para varios
atributos.

DISEÑO DEL SOFTWARE

(cap 6 y 7)

- Comienza una vez que los requerimientos están definidos
- Se realiza antes de la implementación
- Lenguaje intermedio entre requerimiento y el código
- Desgloseamiento desde el dominio del problema al dominio de la solución
 - El resultado es el que se usa para implementar el sistema
 - Actividad creativa
 - Objetivo: crear un "plano del sistema"
 - Determina las mayores características del sistema
 - Gran impacto en testing y mantenimiento
 - Su documentación forma referencia para casos posteriores

Niveles en el proceso de diseño:

→ diseño arquitectónico (arquitectura):
identifica componentes necesarios del sistema, su comportamiento y relaciones

→ diseño de alto nivel:

Vista de módulos del sistema → cuáles son, qué deben hacer, cómo se organizar e interconectar

→ diseño detallado:

Cómo se implementan las componentes/módulos, incluye detalles de procesamiento lógico y estructuras de datos muy cercano al código.

Criterios para evaluar el diseño:

El **objetivo** es encontrar el mejor diseño posible dentro de las limitaciones (requerimientos, entorno físico y social)

Para ello debemos especificar **propiedades** y **criterios**, que suelen ser **subjetivos** y poco cuantificables.

Los principales criterios son:

→ **Corrección**: fundamental. Un diseño es **correcto** si implementa **todos** los **requerimientos** y es **fácil** de detectar las **restricciones**.

→ **Eficiencia**: uso apropiado de los **recursos** del sistema. Toma un segundo plano debido al abaratamiento del hardware (en algunos sistemas, no así en sistemas integrados o de tiempo real)

→ **Simplicidad**: Un diseño simple será **fácil** de comprender. Esto tiene un **impacto directo** en la **mantenibilidad** del sistema. Tmb facilita el **testing**, descubrimiento y corrección de **bugs** y modificar el código.

Estos dos últimos criterios **no son independientes** (están comprometidos) y es tarea del diseñador encontrar un equilibrio.

Principios fundamentales del diseño:

Dado que el diseño es un proceso creativo, no existe una serie de pasos a seguir para crearlo, pero sí hay principios que guían el proceso.

Principios fundamentales:

Partición y jerarquía

- Principio básico => "divide y vencerás"
- dividir el problema en pequeñas partes que sean manejables
 - cada una debe poder solucionarse y modificarse separadamente
- las partes no son totalmente independientes, se comunican y cooperan entre sí para solucionar el problema mayor
- esta comunicación agrega complejidad
- más componentes => más cuesta el particionado
deben cuando sugiere el beneficio
- tratar de mantener la mayor independencia posible entre las partes (simplificar el diseño y facilitar el mantenimiento)
- el particionamiento genera una jerarquía de componentes, este, usualmente, se forma a partir de la relación "es parte de"

abstracción

- describe el comportamiento externo sin dar detalles internos de cómo se da ese comportamiento en el componente
- esencial en el particionado

abstracción de componentes existentes

- componentes como cajas negras
- oculta detalle, provee comportamiento externo
- útil para comprender sistemas existentes
- útil para determinar el diseño del sistema existente

abstracción en el proceso de diseño

- componentes no existen (todavía)
- para decidir cómo interactúan solo se releva el comportamiento externo
- permite concentrarse en un componente a la vez, sin preocuparse por las otras
- permite el control de la complejidad
- permite una transición gradual de abstracto a concreto
- necesaria para solucionar los problemas separadamente

dos mecanismos comunes de abstracción

- **funcional:**

- especificar el módulo según la función que este realice, se tratan como funciones de entrada/salida
 - se usan pre y post-condiciones
 - base de las metodologías orientadas a funciones

de objetos:

- el sistema es visto como un conjunto de objetos que proveen servicios.
- Solo estos servicios son visibles por fuera del objeto
- base de metodologías orientadas a objetos

modularidad

Un sistema se considera modular si consiste en componentes discretos tal que quedar implementarse separadamente y los cambios en cada uno no tienen gran impacto en los demás.

- Ayudar en el debugging, en la reparación y en la construcción del sistema
- es un objetivo para crear sistemas entendibles y manejables
- resulta de la conjunción de la abstracción y el particionado.

top-down → del componente de más alto nivel (más abstracto) a los de niveles más bajos

Estrategias para diseñar la jerarquía de los componentes

bottom-up → del componente de más bajo nivel (más simple) al de más alto nivel

bottom-up

DISEÑO ORIENTADO A FUNCIONES

El sistema es visto como una función que transforma las entradas en salidas deseadas. Se especifican los módulos también como funciones o procesos de entradas/salidas.

conceptos a nivel módulo

módulo: parte lógicamente separable, unidad discreta e identificable respecto a carga y compilación.

se usan ciertos criterios para que los módulos soporten abstracciones bien definidas y sean solucionables y modificables por separado y así hacer que el diseño sea modular.

Acoplamiento: grado de dependencia entre los módulos.

- los módulos deben estar tan débilmente acoplados como sea posible (que sean independientes en caso de que se queden)
- es un concepto inter-modular
- factores que incluyen en el acoplamiento:

(figos) → tipo de conexión: complejidad y escavidad de las interfaces de un módulo

disminuye si:

- + solo las entradas definidas en un

minimizadas

módulos son usados por otros, ejemplo, la info de módulo a módulo sólo se pasa a través de parámetros

anterior si:

usamos interfaces indirectas y oscuras, ejemplo, se usan atributos y operaciones internas al módulo, se usan variables compartidas

→ complejidad de cada interfaz: cantidad y complejidad de los parámetros mientras más compleja la interfaz más incrementar su acoplamiento.

cuando se usa más de lo necesario

cierto nivel es necesario para la comunicación, encontrar balance mediante las simples

→ tipo de flujo de información: dos tipos de info

+ control: la acción del módulo depende de esta info

+ datos: el módulo se comporta como una función & entradas/salidas bajo acoplamiento:

+ las interfaces solo reciben y envían info dentro (más bajo) ó sólo info de control (más que si sólo de datos, gen. bajo)

alto acoplamiento:

+ las interfaces reciben y envían info híbrida, es decir, tanto info de datos como info de control

Cohesión: relación entre elementos del mismo módulo, qué tan cercanos son.

- intra-modular

- debe ser alta.

- varios niveles de cohesión:

1 casual: la relación no tiene significado

2 lógico: relación lógica

3 temporal: relación en el tiempo, se ejecutan juntos

4 procedural: elementos en una misma unidad procedural

5 comunicacional: elementos operan sobre el mismo dato

6 secuencial: salida de uno es la entrada de otro

7 funcional: todos se relacionan para llevar a cabo una sola función

acabando de menor cohesión a mayor cohesión

Los módulos funcionalmente cohesivos pueden describirse con una oración simple

Diagramas de estructura

Es una notación gráfica para especificar la estructura estática del software

Representa los módulos, su jerarquía y sus interconexiones.

Sirve para que el diseñador pueda representar de forma rápida sus decisiones de una forma compacta y así poder evaluarlas y modificarlas.

Metodología de diseño estructurado

Como todo diseño, esta provee **guías** para ayudar al **diseñador** en el proceso de diseño.

Ve al software como una función de transformación (convierte una entrada en una salida esperada).

Su objetivo es especificar módulos de funciones y sus conexiones siguiendo una estructura jerárquica con bajo acoplamiento y alta cohesión.

Pasos:

1- Reformular el problema como un DFD: se captura el flujo de datos del sistema que se pregunta.

2- Identificar entradas y salidas más abstractas:

- **MAI**: elementos de datos en el DFD que están más distantes de las entradas reales que queden considerarse entradas

- **MAO**: ídem pero desde la salida hacia la entrada

Esto separa las distintas funciones del sistema

- Subsistema que realiza principalmente la entrada

- " " " "

las transformaciones

- " " " "

la presentación de la salida.

3- Realizar primer nivel de factorización: divide el problema en tres problemas separados

- Módulos de entrada (uno x cada MAI)
- Módulos transformadores (uno x el transformador central)
- módulos de salida (uno x cada MAO)

Estos tres se conectan con un módulo controlador (principal)

4- Factorizar los módulos de entrada, salida y de transformadores

- Módulos de entrada y salida: los transformadores que producen las MAI y MAO son tratados ahora como transformadores centrales y sobre cada uno se requiere el proceso del primer nivel de factorización.
- Módulos transformadores: se deben determinar los subtransformadores que compuestos forman el transformador.

5- Mejorar la estructura usando heurísticas y verificación

Heurísticas de diseño

Conjunto de reglas generales / principios básicos para mejorar la estructura del sistema.

Tienen que ver con el tamaño del módulo, con la cantidad de flechas de entrada y salida, alcance del efecto de un módulo, alcance del control de un módulo.

Rule of thumb: por cada módulo, el alcance del efecto tiene que ser un subconjunto del de control.

Verificación del diseño

Objetivo: asegurar que el diseño implemente los requerimientos

Métricas

Proveer una evaluación cuantitativa del diseño para luego mejorarlo

- **Tamaño:** cantidad módulos y sus tamaños estimados
- **Complejidad:** módulos simples es mejor
- **Métricas de red:** se enfoca en el diagrama de estructura, es bueno si c/u módulo tiene un sólo módulo invocador
- **Métricas de estabilidad:** estabilidad de un módulo tiene que ver con la cantidad de módulos dependientes/exequentes de él que tiene + estabilidad mejor
- **Métricas de flujo de información:** Se puede calcular la Complejidad de un módulo teniendo en cuenta la info que recibe (inflow), la info que sale (outflow) y la cantidad de módulos a los que llama (fan-out) y por los que es llamado (fan-in)

$$DC = \text{fan-in} * \text{fan-out} + \text{inflow} * \text{outflow}$$

Se calcula el promedio y desv. estándar de todos los módulos $\Rightarrow DC > \text{media} + \text{estándar} \Rightarrow$ riesgo de error
 $\text{media} < DC < \text{media} + \text{estándar} \Rightarrow$ complejidad normal en caso contrario.

DISEÑO ORIENTADO A OBJETOS

Ve datos y funciones juntas. Su **objetivo** es **definir las clases del sistema** a definir y **las relaciones** entre estos.

Hace **incapacitación** en el **comportamiento dinámico** del sistema.

Algunos conceptos

+ **clases**: plantilla de la cual se crean los objetos; define la estructura y los servicios tiene:

- **atributos** que definen las partes accesibles desde el exterior
- **metodo** que implementa las operaciones
- **variables** de instancia para resaltar el estado

+ **objetos**: instancia de una clase encapsulado es su propiedad básica.
tienen estado persistente

[Objeto vinculado con otro durante un tiempo] \Rightarrow Asociación entre ellos

[Objeto es parte de otra clase] \Rightarrow Agregación

[Objeto compuesto por otros] \Rightarrow Composición

[Objeto definido por otro] \Rightarrow Herencia

Conceptos de diseño

El diseño se puede evaluar usando

Aglomeramiento

Tres tipos

- por interacción: se den cuando métodos de una clase invocan a otros de otra clase

Menor aglomeramiento

- + métodos acceden a partes internas de otros métodos, manipulan variables de otras clases, etc.

Menor aglomeramiento

- + los métodos se comunican directamente a través de los parámetros (menor n.º de paráms. posibles)

- de componentes: incrementar cuando una clase tiene variables de otra clase (variables de instancia, parámetros, método con variables locales)

decrementar cuando las variables de otra clase en una clase son atributos, o parámetros de un método

- de herencia: implica aglomeramiento aumentar

+ las subclases modifican o eliminan la declaración de un método (que)

+ mantienen la declaración pero modifican su comportamiento.

disminuye

+ solo agregar variables de instancia y métodos

Cohesión

tres tipos

• de método: por qué los elementos de un método están juntos en ese método

Otra cohesión:

+ cada método implementa una única función claramente definida con todos sus elementos trabajando para llevarla a cabo

• de clase: por qué distintos atributos y métodos están en la misma clase

Otra cohesión:

+ la clase representa un único concepto

Baja cohesión:

+ la clase encapsula múltiples conceptos

• de la herencia: por qué distintas clases están juntas en la misma jerarquía

Otra cohesión:

+ las clases están jerarquizadas en consecuencia de la generalización-especialización

Baja cohesión

las clases jerarquizadas en consecuencia del reuso?

? No decir..

(OCP) {Principio abierto-cerrado}

"Las entidades de software deberían ser abiertas para extenderse y cerradas para modificarse"

B. Meyer

El comportamiento debe extenderse para adquirir el sistema a nuevos regerimientos, pero el código existente no debe verificarse.

Minimiza el riesgo de "dejar" lo existente

En OO: se satisface al user correcrumente herencia y polimorfismo

permite crear nuevas
(sub)clases para extender
el comportamiento del
Sistema

Principio de Sustitución de Liskov (LSP)

"Un programa que utiliza un objeto de clase C debería permanecer inalterado si ese objeto se reemplaza por otro de una subclase de C"

B. Huberman/Liskov

Si las jerarquías de un programa siguen este principio, entonces el programa cumple con el principio abierto-cerrado

Otros principios

Principio de Responsabilidad Única: (SRP)

Una clase debe tener sólo una razón para cambiar, o sea, sólo debe tener una responsabilidad.

Principio de Segregación de Interfaces: (ISP)

Las interfaces deben ser específicas y enfocarse en los requerimientos de los clientes que las utilizarán.

Principio de Inversión de Dependencia: (DIP)

Las clases deben depender de las abstracciones y no de implementaciones concretas.

OMT (Object Modeling Technique)

Pasos:

1 - Producir el diagrama de clases: básicamente el Objetivo durante el análisis

2 - Producir el modelo dinámico: ayuda a especificar cómo cambia el estado de los objetos cuando ocurre un evento (solicitud de operación). Sirve para definir qué operaciones tendrá la clase.

3 - Producir el modelo funcional: describe las operaciones que tienen lugar en el sistema, especificar cómo computar los valores de salida a partir de los de entrada.

4- Definir las clases y operaciones

internas: se evalúa críticamente cada clase para ver si es necesaria en su forma actual, considerando versiones de implementación.

5- Optimizar: agregar asociaciones redundantes, guardar atributos derivados, usar tipos genéricos, ajustar la herencia.

Métricas

WMC

• Métodos pedidos por clase: mide la complejidad de la clase, esto depende de la cantidad de métodos en la misma y su complejidad.

DIT

• Profundidad del árbol de herencia: las clases que están muy por debajo de la jerarquía de clases heredan muchos métodos, dificultando la predicción de su comportamiento.

NOC

• Cantidad de hijos: cantidad de subclases inmediatas de una clase. Identifica el reuso (menor cant. hijos, menor reuso) y da un orden de las clases más importantes de corregir (más h.ijo, más corrección)

CBC

- Acoplamiento entre clases: cantidad de clases a las que una clase está acoplada. Cuantifica la modificabilidad y la propensión a errores.

RFC

- Resqueste para una clase: cantidad de métodos que pueden ser invocados como resquistas de un mensaje recibido por un objeto de esa clase. Predice propensión a errores).

DISEÑO DETALLADO (ap. 8)

Nivel del proceso de diseño que se encarga de especificar la lógica que implementará cada uno de los módulos del diseño de nivel intermedio

PDL (Process Design Language)

Es una forma de comunicar el diseño a cualquier nivel de detalle que nos convenga. En este caso lo usamos para extender el diseño incluyendo la lógica.

Este PDL apunta a expresar el diseño en un lenguaje que sea preciso y no ambiguo, sin mucho detalle y que pueda ser fácilmente convertido a implementación.

Tiene la **sintaxis** de un lenguaje de programación estructurado pero el **vocabulario** de un lenguaje natural

Ventajas:

- fácil de integrar con código fuente, lo hace fácil de mantener
- permite la declaración de datos y del procedimiento
- forma más barata y efectiva de cambiar la argi del programa

desventajas:

- no explica funcionalidad de una manera comprendible
- la notación es comprendible para personas con manejo del PDL

constructores básicos

- IF-THEN-ELSE
- CASE
- DO

? no necesitan escribirse
} formalmente

El método más común para diseñar los algoritmos es el refinamiento paso a paso

Verificación

Mostrar que el diseño de software cumple con las especificaciones del diseño

Tres métodos:

- Recorridos del diseño: reunión informal
- Revisión crítica del diseño: solo si se usó PDL.
- Verificadores de consistencia: revisar estándar.

Métricas

Se usan muchas destinadas al código.

Otras:

- Complejidad ciclomática: más sentencias y condiciones => más complejidad
- Volumen de datos: cómo se comunican los módulos a través de las invocaciones que se hacen. Mide acoplamiento
- Métrica de cohesión: más cohesión si la ejecución del módulo usa todos los recursos del mismo.