1. Introducción

El desafío de la Ingeniería del Software

Ingeniería del Software: Se define como la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software,

Enfoque sistemático: Metodologías y practicas existentes para solucionar un problema dentro de un dominio determinado

▼ Desafíos principales

1. **Escala,** queremos software tenga la capacidad de escalar para arriba y para abajo.
2. **Consistencia** y **Repetibilidad**, queremos que los procesos definidos por la ingeniería del software sean repetibles, y consistentes, es decir que se pueda aplicar el mismo proceso y obtener resultados parecidos.
3. **Cambios,** queremos que el software sea adaptable tanto a cambios de personal de desarrollo (programadores) como a cambios de hardware/entorno.
4. P**roductividad,** tanto una solución que demora mucho tiempo como un software barato y de baja calidad son inaceptables.
5. **Calidad**

   Para medir calidad se utilizan los siguientes parámetros

   1. **Funcionalidad**, que provea las funciones que fueron establecidas o especificadas por el cliente.
   2. **Confiabilidad**, que no haga más cosas de las que queremos y que se hagan en tiempo y forma.
   3. **Usabilidad**, capacidad de ser comprendido, utilizado o aprendido por el usuario.
   4. **Eficiencia**, que el uso de hardware/recursos sea acorde.
   5. **Mantenibilidad**, capacidad de ser modificado con el propósito de corregir, mejorar o adaptar.

6. **Portabilidad**, capacidad de ser adaptado a distintos entornos sin aplicar mas acciones que las especificadas por el producto.

Cabe aclarar que no se puede garantizar todo, siempre se pone el foco en algunos parámetros.

Enfoque de la Ingeniería del Software: Para alta C&P necesitamos buena **tecnología**, buenos **procesos** o **métodos** y **gente** que haga bien su trabajo. El **proceso** es lo que menos varia en el tiempo, queremos la sistematización de los procesos.

▼ Fases del proceso de desarrollo

Es importante separar en fases para la separación de incumbencias (echar culpas). Cada grupo tiene una tarea específica. Cuando la tarea general falla, es importante saber qué parte falló.

En general los modelos de procesos consisten de:

1. **Análisis de requisitos y especificación:** qué quiere el cliente
2. **Arquitectura y Diseño:** el sistema se separa en módulos
3. **Codificación**
4. **Testing**
5. **Entrega e instalación:** entregar, instalar y mantener

Queremos que cada etapa tenga una entrada y una salida definidas para poder testear

2. Analisis y especificacion

▼ SRS

Debe dar una clara comprensión de lo que se espera de un determinado software.

La SRS es el medio para **reconciliar las diferencias** y **especificar las necesidades** del cliente/usuario de manera que todos entiendan.

Hay abogados y escribanos involucrados. Es un **contrato** o acuerdo entre cliente/usuario y quién suministrará el software. No debe ser ambigua.

▼ ¿Por qué la SRS es necesaria?

1. **Una SRS establece el contrato entre el cliente y el proveedor respecto a lo que el software va a hacer.**
2. **La SRS provee un referencia para la validación del producto final.**
3. **Una SRS de alta calidad es esencial para obtener un software de calidad.**
4. **Una buena SRS reduce los costos de desarrollo.**

▼ Actividades básicas

Dividimos esta etapa en sub-etapas:

▼ 1. Análisis del problema

El objetivo es lograr una buena comprensión de las necesidades, requerimientos y restricciones del software.

El análisis incluye:

1. Entrevistas con el cliente y usuarios.
2. Lectura de manuales.
3. Estudio del sistema actual.
4. Ayudar al cliente/usuario a comprender las nuevas posibilidades.

Prototipado

Un prototipo de software se puede definir como una **implementación parcial** de un sistema cuyo propósito es **aprender** algo del problema a resolver o del enfoque de la solución.

Existen enfoques de prototipado:

- **Descartable**: Se construye con la idea de que va a descartarse luego de que se complete el análisis.
- **Evolutivo**: Se construye con la idea de que eventualmente se va a convertir en el sistema final.

Existen dos tipos de prototipo:

- **Vertical**: Una parte elegida del sistema, que no ha sido bien comprendida, se construye completamente.
- **Horizontal**: El Sistema es organizado como una serie de capas y alguna capa es el foco del prototipo.

▼ 2. Especificación de los requerimientos

Requerimientos: Una condición o capacidad necesaria de un usuario para solucionar un problema o alcanzar sus objetivos.

Características de la SRS

1. **Correcta**

   Cada requerimiento representa alguna característica deseada por el cliente.

2. **Completa**:

   Todas las características deseadas por el cliente están descritas.

3. **No ambigua:**

   Cada requerimiento tiene exactamente un significado

4. **Consistente:**

   Que no se contradiga.

5. **Verificable:**

   Existe algún proceso efectivo que puede verificar que el software final satisface el requerimiento.

6. **Rastreable:**

Se debe poder determinar el origen de cada requerimiento y cómo éste se relaciona a los elementos del software.

7. **Modificable:**

   La estructura y estilo de la SRS es tal que permite incorporar cambios fácilmente preservando completitud y consistencia.

8. **Ordenada en aspectos de Importancia y Estabilidad:**

   Los requerimientos puede 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.

▼ 3. Validación

Proceso

Hay muchas posibilidades de malentendidos en el proceso de análisis y especificación.

La SRS se revisa por un grupo de personas, conformado por autor, cliente, representantes de **usuarios** y de desarrolladores.

Métricas

Para poder **estimar costos y tiempos**, y planear el proyecto se necesita "medir" el esfuerzo que demandará. Una métrica es importante solo si es útil para el seguimiento o control de costos, calendario o calidad

Punto función

Se determina a partir de la SRS. Define el tamaño en términos de Funcionalidad y se obtiene el Tamaño estimado del proyecto en LOC

Se define en términos de la funcionalidad, definimos el siguiente cuadro

Tipo de función	Descripción
Entradas externas	Tipo de Entrada (Dato/Control) externa a la aplicación
Salidas externas	Tipo de salida que deja el Sistema
Archivos lógicos	Grupo lógico de dato/control de información generado/usado/manipulado
Archivos de interfaz externa	Archivos pasados/compartidos entre aplicaciones

Tipo de función	Descripción
Transacciones externas	Input/output inmediatos (Queries)

Se categorizan en **Simple**, **Promedia**, **Compleja**

Luego se define c_{ij} como la cantidad de funciones del tipo i con complejidad j y se calcula el punto función no ajustada como $\mathrm{UFP} = \sum_{i=1}^{5} \sum_{j=1}^{3} (c_{ij} w_{ij})$.

Luego, vamos a ajustar esta función según el entorno, definimos entonces las características del entorno y los criterios de caracterización:

Características de entorno: (Laura dijo que nos acordemos 4)

- Reusabilidad
- Múltiples sitios
- Ingreso de datos online
- Actualizacion online
- Procesamiento distribuido
- Facilidad para la Instalación
- Facilidad para la Operación

Criterios:

1. no presente (p_0)
2. influencia insignificante (p_1)
3. influencia moderada (p_2)
4. influencia promedio (p_3)
5. influencia significativa (p_4)
6. influencia fuerte(p_5)

Se calcula entonces $\mathrm{CAF} = 0.65 + .01 \sum_{i=0}^{14} p_i$

Por último se calculan los punto función como $\mathrm{UPF} \times \mathrm{CAF}$

Cabe aclarar que cada punto función se corresponde con x LOC en el lenguaje y para la compañía z, es decir, no es universal, cada compañía

tiene sus estimaciones dependiendo de muchos factores.

3. Arquitectura del software

Da una visión de las partes del sistema y de las relaciones entre ellos. La arquitectura permite que los cambios sean simples de hacer en sistemas grandes y complejos. Permite information hiding y encapsulación.

Definición: La Arquitectura de Software de un sistema es la arquitectura del sistema que comprende a los elementos del software, las propiedades externamente visibles de tales elementos, y la relación entre ellas.

▼ ¿Por qué es importante?

- **Comprensión y comunicación**: Muestra la estructura de alto nivel del sistema ocultando la complejidad de cada una de las partes, lo cual ayuda al entendimiento entre las partes.
- **Reuso**: Se elige una arquitectura tal que las componentes existentes encajen adecuadamente con otras componentes a desarrollar.
- **Construcción y evolución**: La división sirve para guiar el desarrollo y asignar equipos de trabajo.
- **Análisis**: La arquitectura permite predecir parcialmente propiedades de confiabilidad y desempeño.

▼ Vistas de la arquitectura

Una vista consiste de **elementos y relaciones** entre ellos lo cual describe una estructura, la cual destaca un aspecto determinado, entendemos que no hay una única vista de un sistema.

La mayoría de las vistas propuestas pertenece a alguno de estos tipos:

1. **Vista de módulos**: estructuras de código, planeamiento.
2. **Vista de Componentes y conectores**: estructuras de ejecución, análisis de desempeño.
3. **Vista de Asignación de recursos**: co-estructuras de software y entorno.

▼ Vista de módulos

Un sistema es una colección de **unidades de código** las cuales no representan entidades en ejecución. La **relación** entre ellos está basada en

el código.

Ej. unidades de código: clases, paquetes, funciones, procedimientos, métodos, etc.

Ej. relaciones: "parte de", "usa a", "depende de", llamadas, generalización o especialización, etc.

▼ Vista de Asignación de Recursos

Se enfoca en cómo las unidades de software se asignan a recursos como hardware, sistemas de archivos, personas, etc.
Exponen propiedades estructurales como qué proceso ejecuta en qué procesador, qué archivo reside dónde, etc.

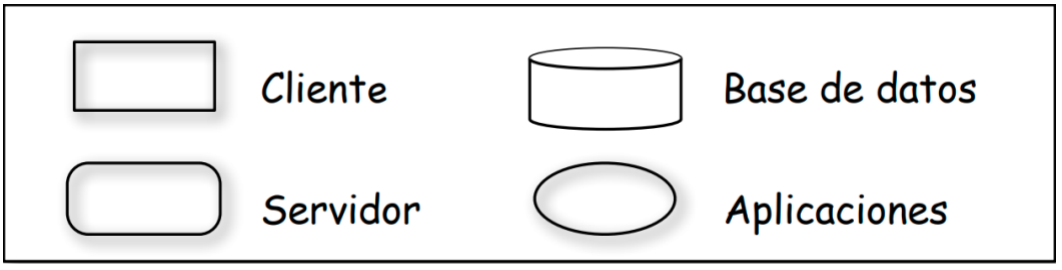
▼ Vista de Componentes y conectores

Los elementos son entidades de ejecución denominados **componentes**. Esta vista define los componentes y cómo se conectan entre ellos a través de conectores. Describe un estructura en ejecución del sistema, qué componentes existen y cómo interactúan entre ellos en tiempo de ejecución.

▼ Componentes:

Son elementos computacionales o de almacenamiento de datos. Los componentes utilizan interfaces o puertos para comunicarse con otros componentes.

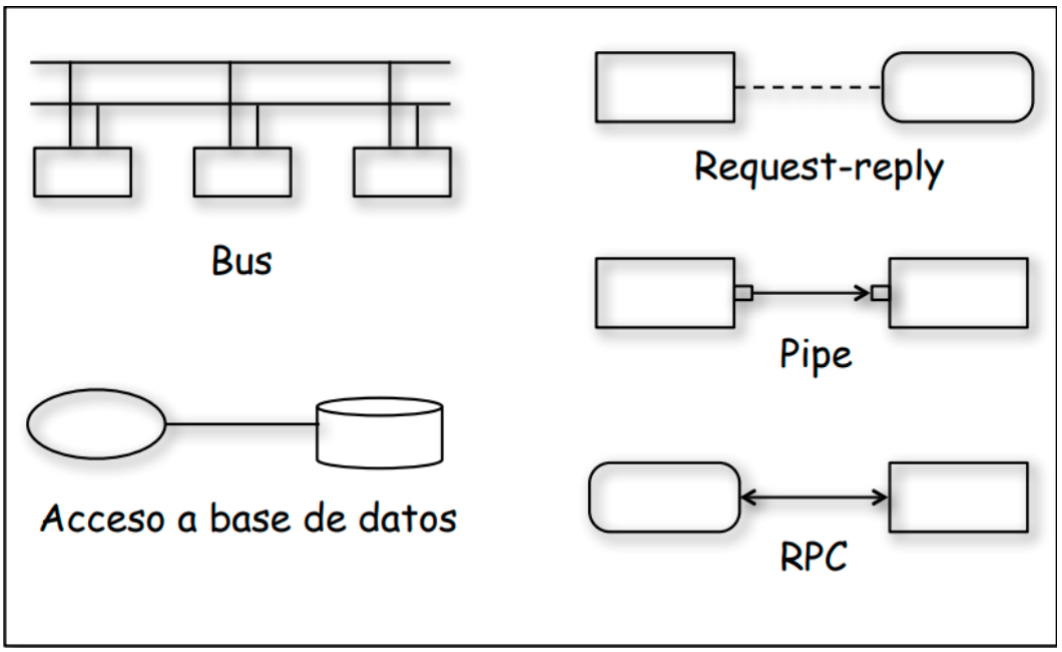
Ejemplo: objetos, procesos, .exe , .dll , etc.



▼ Conectores:

Son mecanismos de interacción entre los componentes. Los conectores describen el medio en el cual la interacción entre componentes toma lugar. Los conectores no necesariamente son binarios. Tienen nombre y tipo.

Ejemplo: pipes, sockets, protocolos,llamada a procedimiento/función, puertos TCP/IP, etc.



▼ Estilos arquitectónicos para la vista de C&C

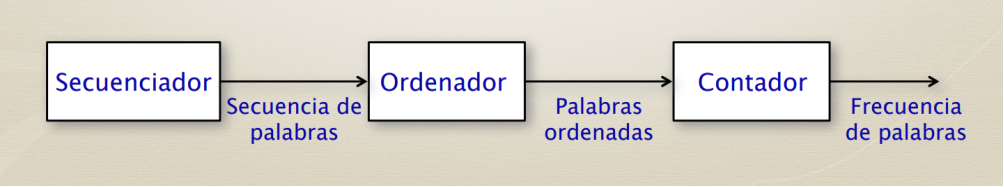
Define una familia de arquitecturas que satisfacen las restricciones de ese estilo. Distintos estilos pueden combinarse.

▼ Pipe and Filter:

Adecuado para sistemas que fundamentalmente realizan transformaciones de datos. Un filtro realiza transformaciones y le pasa los datos a otro filtro a través de un tubo.

- **Filtro (Componente):** es independiente y asíncrona, se limita a consumir y producir datos..
- **Pipe (Conector):** Unidireccional y unarios.

Ejemplo:



▼ Datos Compartidos:

Está conformado por:

- **Repositorio (Conector):** provee almacenamiento permanente y confiable.

- **Usuario (Componente):** acceden a los datos en el repositorio, realizan cálculos y ponen los resultados otra vez en el repositorio.
- **Conector:** Existen solo de un tipo, se limitan a lectura/escritura

La comunicación entre los usuarios sólo se hace a través del repositorio, los componentes no actúan directamente entre ellos.
Hay dos estilos:

- **Estilo pizarra**, cuando se modifica el repositorio, se informa a todos los usuarios.
- **Estilo repositorio**, no se informa de cambios.

▼ **Cliente Servidor**

La comunicación es siempre iniciada por el cliente y usualmente sincrónica. Usualmente el cliente y el servidor residen en distintas máquinas.

Está conformado por:

- **Clientes:** sólo se comunican con el servidor.
- **Servidores:** solo responden a los clientes.
- **Conector**: solicitud/respuesta, es asimétrico y asincrónico.

Otros estilos son:

- **Publicar-suscribir:** Componentes publican eventos y cuando estos son enviados se invocan a los componentes que están suscritos a dichos eventos.
- **Peer-to-peer**: Único componente Peer que puede pedir servicios a otros Peer.
- **Procesos que se comunican**: Procesos que se comunican a través de mensajes.

▼ ATAM (Architecture Tradeoff Analysis Method)

Analiza las propiedades y las concesiones entre ellas.

Pasos principales

1. **Recolectar escenarios**:
   Se eligen los escenarios (interacciones con el sistema) de interés para el

análisis.

2. **Recolectar requerimientos y/o restricciones**:
   Definir lo que se espera del sistema en tales escenarios. Junto a los atributos de interés
3. **Describir las vistas arquitectónicas**
   Las vistas del sistema que serán evaluadas son recolectadas.
4. **Análisis específicos a cada atributo**.
   Se analizan las vistas bajo distintos escenarios de forma separada para cada atributo de interés. Esto forma la base para la elección entre arquitecturas.
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.

4. Diseño

- Es una actividad **creativa**
- Tiene un gran impacto en el **testing** y **mantenimiento**
- Es la vista de los módulos del sistema

▼ Criterios para Evaluar el Diseño

1. **Corrección**
    - ¿El diseño implementa todos los requerimientos?
    - ¿Es factible el diseño dada las restricciones?
2. **Eficiencia**
    - Apropiado uso de los recursos del sistema (principalmente CPU y memoria).
3. **Simplicidad**
    - Facilita la comprensión del sistema.
    - Facilita el testing, modificación de código, mantenimiento, descubrimiento y corrección de bugs.

▼ Principios de diseño

- **Partición y jerarquía**:

  Consiste en dividir el problema en pequeñas partes, simplificando el diseño y facilitando el
  mantenimiento. Cada parte debe poder **solucionarse y modificarse independientemente.** Aunque no son totalmente independientes: deben comunicarse para solucionar el problema mayor.

- **Abstracción**

  La abstracción de una componente describe el comportamiento externo sin dar detalles internos de cómo se produce dicho comportamiento. Es decir, trata de ocultar detalles de lo que pasa en niveles más bajos.

  Abstracción orientada a funciones

  Abstracción orientada a objetos

- **Modularidad**

  Un sistema se dice modular si consiste de componentes discretas que se pueden implementar separadamente; donde un cambio a una de ellas tenga mínimo impacto sobre las otras. Resulta de la **conjunción** entre **Particion y jerarquía** y **Abstracción**

▼ Diseño orientado a funciones

Un módulo es una parte lógicamente separable del programa, es una unidad discreta e identificable respecto a la carga y compilación

▼ Acoplamiento

El acoplamiento es un concepto **inter-modular**, que determina la forma y nivel de **dependencia** entre módulos.

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

Siempre buscamos poco acoplamiento.

▼ Factores que influyen en el acoplamiento

1. **Tipo de conexiones entre módulos**:

   La complejidad y oscuridad de las interfaces.

   Ejemplo: ¿Utilizo solo las interfaces especificadas o también uso datos compartidos?

2. **Complejidad de las interfaces**:

   ¿Estoy pasando solo los parámetros necesarios?

   Cierto nivel de complejidad en las interfaces es necesario para soportar la comunicación requerida con el módulo.

3. **Tipo de flujo de información entre módulos**:

   ¿Estoy pasando parámetros de control (flags)? Si se pasan flags que determinan un caso de la función, es probable que se pueda dividir en varias funciones.

   Transferencia de información de control permite que las acciones de los módulos dependan de la información; y hace que los módulos sean más difíciles de comprender.

El acoplamiento disminuye si:

- Solo las entradas definidas en un módulo son utilizadas por los otros.
- La información se pasa exclusivamente a través de parámetros.
- Sólo se pasa la información estrictamente necesaria.
- Las interfaces solo contienen comunicación de datos.

El acoplamiento se incrementa si:

- Se utilizan interfaces indirectas y oscuras.
- Se usan directamente operaciones y atributos internos al módulo.
- Se utilizan variables compartidas.
- Se pasan parámetros que contienen más información de la necesaria.
- Las interfaces contienen comunicación de información híbrida (datos+control).

▼ Cohesión

Es **intra-modular**. Tiene que ver con la relación de las componentes del mismo módulo. Y determina cuán fuertemente vinculados están los elementos de un módulo.

- Minimizar relaciones entre elementos de módulos distintos.
- Maximizar relaciones entre elementos del mismo módulo.

Siempre buscamos alta cohesión.

▼ Tipos de cohesión

1. **Casual**, la relación entre los elementos del módulo no tiene significado.
2. **Lógica**, existe alguna relación lógica entre los elementos del módulo.
3. **Temporal**, los elementos están relacionados en el tiempo y se ejecutan juntos.
4. **Procedural**, contiene elementos que pertenecen a una misma unidad procedural.
5. **Comunicacional**, tiene elementos que están relacionados por una referencia al mismo dato.
6. **Secuencial**, los elementos están juntos porque la salida de uno corresponde a la entrada del otro.

7. **Funcional**, todos los elementos del módulo están relacionados para llevar a cabo una sola función.

▼ Diseño orientado a objetos

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

▼ Acoplamiento

Tipos de acoplamiento

- **Interacción**
  - Métodos de una clase invocan a métodos de otra clase.
  - No se puede eliminar del todo.
  - Menor acoplamiento: si se comunican a través de la menor cantidad de parámetros pasando menos información y nada de control.
  - Mayor acoplamiento: Los métodos acceden a partes internas de otros métodos o variables. O la información se pasa a través de variables temporales.
- **Componentes**
  - Una clase A tiene variables de otra clase C. Si A tiene variables de instancia, parámetros o métodos con variables locales de tipo C.
  - Cuando A está acoplada con C, también está acoplada con todas sus subclase.
  - Mejor Si las variables de la clase C en A son, o bien atributos o parámetros en un método. Es decir, son visibles.
- **Herencia**
  - Si una es subclase de otra
  - Lo malo es s modifican la signatura o elimina un método (**Herencia no estricta**)
  - Menor acoplamiento si la subclase solo agrega variables de instancia y métodos pero no modifica los existentes en la superclase

▼ Cohesión

- **Cohesión de método**

    - ¿Por qué los elementos están juntos en el mismo método?
    - La cohesión es mayor si cada método implementa una única función claramente definida.
- **Cohesión de Clase**
    - Una clase debería representar un único concepto con todos sus elementos
      contribuyendo a este concepto.
    - La cohesión es más alta mientras menos clases están encapsuladas en una.
- **Cohesión de Herencia**
    - ¿Por qué distintas clases están juntas en la misma jerarquía?
    - La cohesión es más alta si la jerarquía se produce como consecuencia de la
      generalización-especialización.

▼ Principio Abierto-Cerrado

- El comportamiento puede extenderse para adaptar el sistema a nuevos requerimientos, pero el código existente no debería modificarse. Es decir, permitir agregar código pero no modificar el existente.
- Este principio se satisface si se usa apropiadamente la herencia y el polimorfismo. La herencia permite crear una nueva subclase para extender el comportamiento sin modificar la clase original.
- **Si se cumple el principio de Sustitución de Liskov generalmente se cumple el principio abierto-cerrado.**
- $\text{LISKOV} \Rightarrow \text{ABIERTO-CERRADO}$

Principio de sustitución de Liskov

Un programa que utiliza un objeto A con clase B debería permanecer inalterado si A se reemplaza por cualquier objeto de una subclase de B.