¿Qué es la ingeniería del software? La **Ingeniería de Software** es la aplicación de un enfoque sistemático, disciplinado, y cuantificable al desarrollo, operación, y mantenimiento del software (IEEE 610.12-1990).

Escala
IS debe **considerar la escala del sistema a desarrollar** y los métodos utilizados para desarrollar **pequeños problemas no siempre escalan a grandes problemas**. **Los métodos de IS deben ser escalables** (tanto hacia arriba como hacia abajo).
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.

Productividad
IS está **motivada por el costo y el cronograma** (schedule). 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 / PM) captura ambos conceptos **Si es más alta → menor costo y/o menor tiempo**.
Los **enfoques de IS deben generar alta productividad.**

Calidad
La otra motivación detrás de IS es la calidad por lo que desarrollar software de alta calidad es un **objetivo fundamental.** Esta característica **es difícil de definir** (contrariamente al costo y al tiempo). El enfoque utilizado en la IS debe producir software de alta calidad.

Calidad - estándar ISO
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.

Consistencia y repetibilidad como un gran desafío de la IS : es muy importante utilizar métodos, que aseguren una gran probabilidad de **calidad no solo a nuestro uso sino al de los demás.**

- Algunas veces un grupo puede desarrollar un buen sistema de software.
- Desafío clave en IS: **cómo asegurar que el éxito pueda repetirse** (con el fin de mantener alguna consistencia en la calidad y la productividad).
- 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.
- Marcos como la ISO9001 o el CMM se enfocan mucho en este aspecto.

Cambios, otro desafío de la IS : las empresas e instituciones **mutan constantemente** y es muy habitual, por ello el **software debe cambiar para adaptarse** a estos nuevos cambios. Los **métodos** de la ingeniería de software **deben preparar el producto para que sea fácilmente modificable**. Por ende, los métodos que no permiten cambios son poco útiles.

Enfoque de la IS
Consistentemente **desarrollar software de alta calidad y con alta productividad para problemas de gran escala que se adapten a los cambios**. Calidad y alta productividad son los objetivos básicos a perseguir bajo gran escala y tolerancia a cambios, son consecuencia de la gente, los procesos y la tecnología.

- El enfoque sistemático es realmente el proceso que se utiliza.
- La IS **separa el proceso para desarrollar software del producto desarrollado** (lo que la diferencia de los demás campos).
- 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**.

El proceso de desarrollo en fases :

- El proceso de desarrollo consiste de **varias fases**.
- Cada fase termina con una **salida definida**.
- Las fases se realizan en el **orden especificado por el modelo de proceso que se elija** seguir.
- 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 el progreso** en momentos definidos del desarrollo.
- El proceso en fases es **central en el enfoque de la IS** para solucionar la crisis del software.

En general, está compuesto por :

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

Análisis y especificación de los requisitos del software

- Identificar y especificar los requisitos necesariamente involucra **interacción** con la gente.
- **No puede automatizarse.**
- La fase de requisitos **finaliza** produciendo el documento con la especificación de los requerimientos del software (**SRS**-System Requirements Specification).
- La **SRS especifica lo que el sistema propuesto debe hacer**.

¿Por qué la SRS es necesaria?

- La SRS establece las **bases** para el acuerdo entre el **cliente/ usuario** y quién suministrará el software. Y existen **tres partes vinculadas**, las necesidades del cliente y consideraciones del usuario deben ser comunicadas al desarrollador.
- La SRS es el **medio para reconciliar las diferencias** y **especificar las necesidades** del cliente/usuario de manera que todos entiendan.

¿Por qué la SRS es necesaria? Una SRS de alta calidad es esencial para obtener un software de calidad.

- **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
- Una buena SRS **reduce los costos de desarrollo**.

Objetivo : no sólo automatizar un sistema manual, sino también agregar valor a través de la tecnología.
Verificación: “el software satisface la SRS”.
Proceso de requerimientos : Secuencia de pasos que se necesita realizar para convertir las necesidades del usuario en la SRS y tiene que recolectar las necesidades y requerimientos para luego especificarlos.
Fases este proceso :

1. **Análisis del problema o requerimientos :** se enfoca en la comprensión del sistema deseado y sus requerimientos. El objetivo del análisis es comprender la estructura del problema y el dominio.
2. **Especificación de los requerimientos :** puede ayudar al análisis y se enfoca en el comportamiento externo. Se suele recolectar más información de la necesaria.
3. **Validación :** puede mostrar brechas que conducen a más análisis y más especificación.

- Tengamos en cuenta que este proceso no es lineal, puede ser iterativo y en paralelo. Puede existir sobreposición en las fases. Sosteniendo como estrategia básica “dividir y conquistar” atacando el problema en pequeñas partes. Conforme se van desarrollando las fases se recolecta muchísima información que es clave saber organizarla, utilizando diferentes técnicas.
- Las transiciones entre las fases puede ser complicada (análisis - especificación).
- Durante el análisis, se ayuda a comprender en vez de asistir a la especificación.
- Los métodos del análisis son similares a los del diseño. Pero, el análisis trata sobre el dominio del problema mientras que el diseño con el dominio de la solución.

Análisis del problema o requerimientos
El objetivo del análisis 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 nuevas posibilidades.

El analista no solo recolecta y organiza la información (rol pasivo), sino también actúa como consultor (rol activo). Debe comprender el funcionamiento de la organización, el cliente, y los usuarios.

Observaciones : es importante obtener la información necesaria así como también interactuar con el cliente para establecer las propiedades deseadas (brainstorming). Por otro lado, siempre ayuda tener relaciones interpersonales y habilidad para comunicarse con el cliente. Aspectos claves, pero difíciles de conseguir :

1. Asegurar completitud.
2. Asegurar consistencia.
3. Evitar diseño.

Recordar particionar el problema para luego comprender cada subproblema y su relación. Respecto a funciones, objetos y eventos del sistema. Como proyección está el hecho de obtener distintos puntos de vista.

Análisis del problema : prototipado
La idea es construir un sistema parcial, el cual es utilizado por el cliente, usuario y desarrollador para comprender mejor el problema y las necesidades. Ayuda a visualizar cómo será el sistema final. Se distinguen dos enfoques en los prototipos :

- Descartable (más adecuado): 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.

Especificación de los requerimientos : Características de la SRS
La principal salida de esta etapa es la SRS, y lo que llega a esta etapa mediante el análisis es el conocimiento adquirido sobre el sistema. La SRS debe ser :

- **Correcta :** Cada requerimiento representa precisamente alguna característica deseada por el cliente en el sistema final.
- **Completa :** Todas las características deseadas por el cliente están descritas. Hasta la característica más difícil de lograr (para conseguirla uno debe detectar las ausencias en la especificación). Corrección y completitud están fuertemente relacionadas.
- **No ambigua :** cada requerimiento tiene exactamente una interpretación, si esta es ambigua los errores se colaran fácilmente. Se presta particular atención a si se usa lenguaje natural, notar también que los lenguajes formales ayudan a desambiguar.
- **Consistente :** ningún requerimiento contradice a otro.
- **Verificable :** Si cada requerimiento es verificable, existe algún proceso efectivo que puede verificar que el software final satisface el requerimiento. 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).
- **Rastreable :** se debe poder determinar el origen de cada requerimiento y cómo se relaciona con los demás elementos del software. Debe ser trackable hacia adelante (se debe poder identificar sobre qué elementos de diseño tiene impacto) y hacia atrás (dado un elemento se debe poder rastrear a que requerimientos está atendiendo).
- **Modificable :** debe permitir incorporar cambios fácilmente preservando completitud y consistencia. La redundancia puede ser un gran estorbo a la modificabilidad.
- **Ordenada** : en aspectos de importancia y estabilidad, los requerimientos pueden ser críticos, es decir, pueden ser importantes pero no críticos o deseables pero no importantes. Algunos requerimientos son esenciales y difícilmente cambian con el tiempo. Otros son propensos a cambiar, por ende se necesita definir un orden de prioridades en la construcción para reducir riesgos debido a cambios de requerimientos.

Componentes de una SRS
La SRS está compuesta por lineamientos sobre qué se debe especificar en ella para ayudar a conseguir completitud. La misma debe especificar funcionalidad, desempeño, restricciones de diseño e interfaces externas.

Requerimientos de funcionalidad
Los requerimientos de funcionalidad son el corazón de la SRS y conforman la mayor parte de la especificación. Provee la funcionalidad completa que el sistema ofrecerá y especifica qué salidas se deben producir para cada entrada dada y las relaciones entre las mismas. Esta área describe todas las operaciones que el sistema debe realizar así como también que entradas son válidas y verificar validez en entradas/salidas. Por otro lado, describe el

funcionamiento del sistema en caso de entradas inválidas, errores de cálculos y otras situaciones inválidas.

Validación de los requerimientos
Debido a la naturaleza de esta etapa, hay muchas posibilidades de malentendidos y da lugar a muchos errores, pero es caro quizás, corregir los defectos de requerimientos más adelante por lo que esta etapa es fundamental para ello. Errores como omisión, inconsistencia, hechos incorrectos y ambigüedad son los más comunes.
La SRS es revisada por un grupo de personas, conformado por autor, cliente, representante de usuarios y desarrolladores. Se planteara un proceso de inspección estándar y en esta etapa se pueden detectar entre 40% y 80% de los errores de requerimientos. Las listas de controles son muy útiles para ello.

- Además, existen herramientas para el modelado y análisis de especificaciones.
- Se escriben en lenguajes de especificación formal.
- Hay herramientas automáticas o semiautomáticas que soportan estos lenguajes.
- Permiten verificar consistencia, dependencias circulares, o propiedades específicas.
- También permiten simular para poder comprender la completitud y corrección.

Métricas
Para poder **estimar costos y tiempos y planear el proyecto se necesita "medir" el esfuerzo que demandará, este último depende de muchos factores.** El **tamaño** es el principal factor, y al comienzo, solo puede ser estimado. Conseguir buenos estimadores es muy difícil, por lo que una métrica solo es importante solo 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.

Métricas : Punto función
Es un estimador similar a la métrica, como las LOC. Se determina sólo con la SRS, y define el tamaño en términos de funcionalidad.
Los tipos de funciones, son :

- **Entradas externas**: tipo de entrada (dato/ control) externa a la aplicación.
- **Salidas externas:** tipo de salida que deja el sistema.
- **Archivos lógicos internos:** grupo lógico de dato/ control de información generado/usado/ manipulado.
- **Archivos de interfaz externa:** archivos pasados/ compartidos entre aplicaciones.
- **Transacciones externas:** Input/output inmediatos (queries).

Contar cada tipo de función diferenciando según sea compleja, promedio o simple. Cij denota la cantidad de funciones tipo "i" con complejidad "j".
Punto función no ajustado (UFP): Σ Σ wijCij.

- Ajustar el UFP de acuerdo a la complejidad del entorno. Se evalúa según las siguientes características :
  → **Reusabilidad.**
  → **Facilidad para la instalación.**
  → **Facilidad para la operación.**
  → **Múltiples sitios.**

Factor de ajuste de complejidad (CAF) : se utiliza para acotar errores de punto función. **Es un cálculo que se le añade a punto función (de aquí se obtiene la métrica).**
→ Punto función (Ajustado) : **CAF * UFP**
Si bien el punto función es algo característico de cada empresa, **la unidad final de este cómputo equivale a una determinada cantidad de LOC en un programa determinado.**

Arquitectura del Software
Es muy importante para esta área que el sistema complejo se descomponga en subsistemas que interactúan entre sí. **Un enfoque en el diseño de sistemas es identificar subsistemas** y la forma en que estos se relacionan entre sí. Justamente, este es objetivo de la arquitectura del software, y es un área que recibe mucha información.
La arquitectura de SW de un sistema es la **estructura 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 analizar.

Una descripción arquitectónica del sistema describe las distintas estructuras del sistema.
Es el diseño del sistema al más alto nivel, donde se hacen elecciones de tecnología, productos, servidores, entre otros. No es posible diseñar los detalles del sistema y luego incorporar estas elecciones y debe ser creada de manera que se adapte a estas elecciones. Es la etapa más temprana donde puede evaluarse la confiabilidad y el desempeño y divide al sistema en partes lógicas tal que cada una puede ser comprendida independientemente. Y describe las relaciones entre ellas (que puede ser un proceso complejo).

Vistas de la arquitectura

- **No hay una única arquitectura** de un sistema.
- Hay **distintas vistas** de un sistema de software (paralelismo con ingeniería civil).
- Una vista **consiste de elementos y relaciones entre ellos**, y describe una estructura.
- Los elementos de una vista **dependen de lo que la vista quiera destacar**. Distintas vistas exponen distintas propiedades.
- Una vista que se enfoca en algún **aspecto** que reduce la complejidad con la que debe enfrentarse el lector.
- Muchos tipos de vistas fueron propuestos.
- **La mayoría pertenece a alguno de estos tres tipos:**
    - **Módulo**
    - **Componentes y conectores (C&C)**
    - **Asignación de recursos**

Las distintas vistas están **correlacionadas** (¡todas representan al mismo sistema!). Existen relaciones entre los elementos de una vista y los de otra. Tal relación puede ser muy compleja.
Una descripción arquitectónica consiste de vistas de distintos tipos, cada una mostrando una estructura diferente. Distintos sistemas necesitan distintos tipos de vistas dependiendo de las necesidades.

La vista de componentes y conectores
Existen dos elementos principales en esta vista, los componentes y los conectores.

- **Componentes:** son **elementos computacionales** o de **almacenamiento de datos**. Son unidades de cómputo o almacenamiento de datos, cada uno tiene un **nombre y un rol** los cuales le proveen de **identidad**. Cada componente tiene un **tipo**, estos se representan con distintos símbolos. Los componentes utilizan interfaces o puertos para comunicarse con otros componentes.
- **Conectores** : son **mecanismos de interacción entre las componentes**. Describen el medio en el cual la **interacción entre componentes** toma lugar, estos pueden proveerse por medio del entorno de ejecución. Sin embargo, los conectores pueden también ser **mecanismos de interacción más complejos**, llegando a requerir una **infraestructura de ejecución significativa** junto con **programación especial** dentro de la componente para poder utilizarla. Estos no necesariamente son

binarios, **tienen nombre** (identifica la naturaleza de la interacción) **y tipo** (identifica la interacción). Muchas veces estos **representan protocolos,** ya que las componentes necesitan seguir convenciones al usar el conector, los tipos se diferencian por su notación.

Define los componentes y cómo interactúan entre sí, mediante los conectores describiendo una estructura en ejecución del sistema en el cual los componentes existen y se relacionan 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 de ellas son generales y útiles para una clase de problemas, de allí derivan los estilos arquitectónicos. Un estilo arquitectónico define una familia de arquitecturas que satisface las restricciones de ese estilo, proveen ideas para crear arquitecturas de sistemas y pueden combinarse para definir una nueva arquitectura.

Estilos arquitectónicos para la vista de C&C : Tubos y Filtros

Adecuado para sistemas que fundamentalmente realizan transformaciones de datos. Un sistema que usa este estilo utiliza una red de transformadores para realizar el resultado deseado. **Tiene un sólo tipo de componente que es el filtro y un un sólo tipo de conector que es el 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 t**ubo 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.**

Estilos arquitectónicos para la vista de C&C : Estilo de datos compartidos

Compuesto por 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, y sólo hay un tipo de conector: lectura/escritura. Principalmente, se diferencian dos estilos :

- **Estilo pizarra :** cuando se agregan/modifican datos en el repositorio, se informa a todos los usuarios. La fuente de datos compartidos es una entidad activa.
- **Estilo repositorio** : el repositorio es pasivo.

Estilos arquitectónicos para la vista de C&C : Estilo cliente-servidor

Este estilo está formado por dos componentes, los clientes y los servidores. En primer lugar, los **clientes** sólo se comunican con el servidor, pero **no interactúan con otros**

clientes, la **comunicación** siempre es iniciada por el cliente quien le envía una solicitud al servidor y espera una respuesta. La comunicación es usualmente **asincrónica**, y el único tipo de conector es la solicitud y la respuesta (son asimétricos). Destacar que generalmente, el cliente y el servidor residen en distintas máquinas.
El estilo se caracteriza por una **estructura multi-nivel** en donde el servidor actúa como cliente, un ejemplo clásico podría ser la caracterización en tres niveles :

- Nivel de cliente: contiene a los clientes.
- Nivel intermedio: contiene las reglas del servicio.
- Nivel de base de datos: reside la información.

Otros estilos

- Estilo publicar-suscribir (publish-subscribe) : presenta 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 suscritas a dicho evento.
- Estilo peer-to-peer : presenta un único tipo de componente. Cada uno le puede pedir servicios a otra ≈ modelo de computación orientado a objetos.
- Estilo de procesos que se comunican : presenta procesos que se comunican entre sí a través de pasaje de mensajes.

Arquitectura y diseño
Tanto la arquitectura como el diseño dividen al sistema en partes y dicen cómo estas se organizan, **la arquitectura es un diseño de muy alto nivel que se enfoca en las componentes principales.**
Lo que usualmente llamamos diseño se enfoca en los módulos que finalmente se transformarán en el código de tales componentes, podemos considerar que este provee la vista de módulos al sistema.
El arquitecto y el diseñador definen dónde acaba una tarea y dónde empieza la otra, **en la arquitectura solo se necesita identificar las partes necesarias para evaluar las propiedades deseadas**, **no considera estructura interna e impone restricciones sobre las elecciones que pueden realizarse durante el diseño**.

Evaluación de las arquitecturas
La arquitectura tiene impacto sobre los atributos no funcionales tales como **modificabilidad, desempeño, confiabilidad, portabilidad**, etcétera. Tanto como las elecciones de diseño y codificación, se deben evaluar estas propiedades en la arquitectura propuesta. ¿Cómo hacerlo? Una posibilidad son las técnicas formales y metodologías rigurosas.

Evaluación de las arquitecturas : el método de análisis ATAM
Los pasos principales de este método son :

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 del Software

- Este proceso **comienza una vez que los requerimientos están definidos y se pone en marcha antes de la implementación.**
- Es el **lenguaje intermedio entre los requerimientos y el código**.
- Es el desplazamiento del dominio del problema al dominio de la solución.
- Se procede desde representaciones más abstractas a representaciones más concretas.
- **El resultado es el diseño que se utilizará en la implementación del sistema.**
- El diseño es una **actividad creativa**.
- El objetivo de la misma es crear un **"plano del sistema"** que satisfaga los requerimientos.
- Quizás sea la actividad **más crítica** durante el desarrollo del sistema.
- El diseño determina las mayores características de un sistema.
- Tiene un gran **impacto en testing y mantenimiento**.
- Los documentos de diseño **forman las referencias para las fases posteriores.**

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 e interconectan (orientado a funciones/objetos)
- **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 y es muy cercano al código.

Criterios para evaluar el diseño

El objetivo es encontrar el mejor diseño posible. Se deberán explorar diversos diseños alternativos, siendo los criterios de evaluación usualmente subjetivos y no cuantificables, entre ellos :

- Corrección
- Eficiencia
- Simplicidad

Corrección : es fundamental pero no única, cuestiona si el diseño implementa los requerimientos y si es factible dadas las restricciones.
Eficiencia : le compete el uso apropiado de los recursos del sistema (CPU y memoria), y debido al costo del hardware (bajo), este criterio toma un segundo plano. Pero es muy importante para ciertas clases de sistemas, como por ejemplo, sistemas integrados.
Simplicidad : tiene un impacto directo en el mantenimiento, debido a que este último es caro. Un diseño simple facilita la comprensión del sistema, haciendo el software mantenible. Facilita el testing, el descubrimiento y la corrección de bugs así como también la modificación del código.

La eficiencia y la simplicidad no son independientes, ya que el diseñador debe encontrar un balance entre estas dos.

Principios del diseño
Es un proceso creativo y no existen una serie de pasos que permitan derivar el diseño de los requerimientos. Por lo que los principios sólo guían a este proceso de diseño. Principios fundamentales :

- Partición y jerarquía.
- Abstracción.
- Modularidad.

Estos forman la base de la mayoría de las metodologías seguidas durante este proceso.
Partición y jerarquía : se basa en el principio **"divide y conquistaras",** particionando el problema en pequeñas partes que sean manejables, siendo que cada parte deba poder solucionarse separadamente así como también modificarse independientemente de las demás. Pero las partes **no son totalmente independientes entre sí**, sino que deben comunicarse y cooperar para solucionar el problema mayor.
La **comunicación entre las partes genera complejidad**, y a medida que la cantidad de componentes aumenta, el costo particionado también aumenta. Por ello, es necesario detener el particionado cuando el costo supera al beneficio. Por eso es importante intentar mantener la mayor **independencia** posible entre las distintas partes para simplificar el diseño y facilitar el mantenimiento. El particionado determina **jerarquía** entre los componentes del diseño y se forma a partir de la relación "es parte de".

Abstracción : es **esencial en el particionado del problema** y utilizado en todas las disciplinas de la 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 el mantenimiento.
- Útil para determinar el diseño del sistema existente.

Abstracción durante el proceso de diseño :

- Las componentes no existen.
- Para decidir cómo interactúan las componentes sólo el comportamiento externo es relevante.
- Permite concentrarse en una componente a la vez.
- Permite considerar una componente sin preocuparse por las otras.
- Permite que el diseñador controle la complejidad.
- Permite una transición gradual de lo más abstracto a lo más concreto.
- Necesaria solucionar las partes separadamente.

Existen dos mecanismos comunes de abstracción, la de datos y la funcional.
Abstracción funcional : específica como se comporta funcionalmente un módulo, y estos últimos son tratados como funciones de entrada/salida. La mayoría de los lenguajes proveen características para soportar. Los módulos pueden especificarse usando pre y postcondiciones. Este mecanismo forma la base de las metodologías ambientadas a las funciones.
Abstracción de datos : una entidad del mundo real que provee servicios al entorno, se aplica esa misma definición para las entidades de datos, quienes esperan ciertas operaciones de un objeto de datos. Los detalles internos no son relevantes. La abstracción de datos provee la siguiente 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 de los 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 discretos tal que puedan implementarse separadamente y un cambio a una de ellas que tenga mínimo impacto sobre otras.

- Provee 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.
- Necesita criterios de descomposición, resultando de la conjunción de la abstracción y el particionado.

Conceptos a nivel modulo (diseño orientado a funciones)
Un módulo es una parte lógicamente separable del sistema, siendo una unidad discreta e identificable respecto a la compilación y carga. Los criterios utilizados para seleccionar módulos que soportan abstracciones bien definidas y solucionables-modificables, independientemente, son :
Acoplamiento : dos módulos son independientes si cada uno puede funcionar completamente sin la presencia del otro. La **independencia entre módulos es deseable** debido a que estos pueden modificarse, implementarse y testearse independientemente, decreciendo el costo de programación. De todos modos, en un sistema, no existe la independencia entre todos los módulos. Es vital que los módulos cooperen entre sí, pero mientras más conexiones entre módulos hay, más dependencias entre ellos existen.
El acoplamiento es el encargado de capturar la noción de dependencia, siendo el objetivo diseñar módulos débilmente acoplados como sea posible. Por ende, cuando sea posible, se deben diseñar módulos independientes.

- El nivel de acoplamiento **se define a nivel de diseño arquitectónico** y de alto nivel.
- **No puede reducirse durante la implementación**.
- El acoplamiento **es un concepto intermodular**.

Los factores más importantes que influyen en el acoplamiento son **el tipo de conexiones entre los módulos, la complejidad de las interfaces y el tipo de flujo de información entre módulos.**
El acoplamiento entre módulos queda definido por la “fuerza de conexión” entre dichos módulos.

- Cuanto más necesitamos conocer de un módulo A para comprender un módulo B, mayor es la conexión de A a B.
- Los módulos fuertemente acoplados están unidos por fuertes conexiones.
- Los módulos débilmente acoplados están débilmente conectados.

La complejidad y oscuridad de un módulo incrementa el acoplamiento, por ende es importante minimizar la cantidad de interfaces por módulos y la complejidad de cada una de ellas.
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**, usualmente se usa más de lo necesario, por ende es importante encontrar un balance entre la complejidad de las mismas y mantenerlas lo más simple posible.
El acoplamiento **depende del tipo de flujo de información**, existiendo dos tipos de fuentes, dato y control.
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.

En un sistema con bajo acoplamiento, las interfaces sólo contienen comunicación de datos. Pero en uno con alto acoplamiento, las interfaces contienen comunicación de información híbrida (datos+control)

Cohesión : considera la característica de **maximizar las relaciones entre los elementos de un mismo módulo, definiendo el vínculo intramodular**. Mediante la cohesión intentamos capturar **cuan cercanamente están relacionados los elementos de un módulo entre sí.**
La cohesión da una idea de si los distintos elementos de un módulo poseen características comunes, y está muy relacionada al acoplamiento. **Generalmente, a mayor cohesión, menor acoplamiento**. Pero esta relación no es perfecta.
Existen varios niveles de cohesión, siendo una escala no lineal :

- **Casual:** relación entre elementos de módulos sin significado.
- **Lógica:** existe alguna relación lógica entre elementos módulos, los elementos realizan funciones dentro de la misma clase lógica.
- **Temporal:** elementos relacionados en el tiempo y se ejecutan juntos.
- **Procedural:** contiene elementos que pertenecen a una misma unidad procedural.
- **Comunicacional:** elementos relacionados por una referencia al mismo dato, operan sobre el mismo dato.
- **Secuencial:** los elementos están juntos por que la salida de uno corresponde con la entrada de otro.
- **Funcional:** la más fuerte de todas, todos los elementos de un módulo conectados para llevar a cabo una función.

Los módulos funcionalmente cohesivos siempre pueden describirse con una oración simple.

Metodología de diseño estructurado: la estructura se decide durante el diseño, por lo que la implementación no debe cambiarla. La estructura tiene efectos sobre el mantenimiento, **la metodología de diseño estructurado (SDM) apunta a controlar la estructura. El objetivo de las metodologías de diseño** (y en particular de la SDM) **es proveer pautas para auxiliar al diseñador en el proceso de diseño**. **No reduce al diseño a una secuencia de pasos mecánicos**. Los pasos de este proceso son :

1. **Reformular el problema como un DFD.**
2. **Identificar las entradas y salidas más abstractas.**
3. **Realizar el primer nivel de factorización.**
4. **Factorizar los módulos de entrada, de salida, y transformadores.**
5. **Mejorar la estructura (heurísticas, análisis de transacciones).**

Verificación del diseño : su principal objetivo es **asegurar que el diseño implementa los requerimientos**. También debe realizarse un análisis de desempeño, eficiencia, entre otros aspectos. La revisión del diseño es la forma más común de hacer esta verificación. La calidad del diseño se completa con una buena modularidad.

Métricas
Proveen una **evaluación cuantitativa del diseño, así el producto final puede mejorarse.**

- El **tamaño** es siempre una métrica, así el diseño puede estimarse mejor. Por ejemplo, cantidad de módulos + tamaño estimado de c/u.
- La **complejidad** es otra métrica también.

Métricas de red : Se enfoca en la composición del diagrama de estructuras, se considera un buen diagrama a aquel en el cual cada módulo tiene un solo módulo invocador, lo que reduce el acoplamiento. **Cuanto más se desvía de la forma de árbol, más impuro es el diagrama**. A medida que este valor se hace más negativo, aumenta la impureza del diagrama.

- Impureza del grafo = nodos del grafo - aristas del grafo - 1.
- Impureza = 0 → Arbol.

Métricas de estabilidad : trata de capturar el impacto de los cambios del diseño, a mayor estabilidad mejor diseño. La estabilidad de un módulo, es la cantidad de superposiciones por otros módulos sobre este. Depende de la interfaz del módulo y del uso de datos globales, se conocen luego del diseño.

Métricas de flujo de información : estas métricas, tienen en cuenta :

- Complejidad intramodulo, se estima con el tamaño del módulo (LOC).
- Complejidad intermedio, se estima con inflow y outflow. (Entrada / Salida de información al módulo).
- La complejidad del diseño de un módulo “C”, se define como **DC = tamaño * (inflow * outflow)^2.**

Esta métrica define la complejidad sólo en la cantidad de información que fluye hacia adentro y hacia fuera y el tamaño del módulo.
Ya vimos en la métrica de red que también es importante la cantidad de módulos desde y hacia donde fluye la info. En base a esto, el impacto del tamaño del módulo empieza a resultar insignificante. Por ende, la complejidad del diseño del módulo C se puede definir como:

- **DC = fan_in * fan_out + inflow * outflow**

Donde fan_in representa la cantidad de módulos que llaman al módulo C, y fan_out los llamados por C.

Diseño orientado a objetos
Orientado a objetos :

- Los sistemas procedurales tradicionales separan datos de procedimientos; modela a ambos separadamente.
- Orientación a objetos: **ve a los datos y a las funciones juntas** (abstracción de datos como base).
- El propósito del diseño OO es el de definir las clases del sistema a construir y las relaciones entre éstas.

Análisis OO y diseño OO

- Las técnicas OO pueden utilizarse tanto para el análisis del requerimiento como para el diseño.
- Los métodos y notaciones son similares.
- AOO modela el problema y DOO modela la solución.
- Existen métodos que combinan análisis y diseño (ADOO).
- ADOO sostiene que la representación creada por el AOO generalmente se subsume en la representación del dominio de la solución creada por el DOO.

En este sentido, **la línea entre AOO y DOO no está definida del todo**, sin embargo, se diferencian en el tipo de objetos que manipulan. Los objetos **AOO representan un concepto del problema** (objetos semánticos). Además de los objetos semánticos, el DOO produce otros tipos de objetos :

- Objetos de interfaces → se encargan de la interfaz con el usuario.
- Objetos de aplicaciones → especifican los mecanismos de control para la solución propuesta.
- Objetos de utilidad → necesarios para soportar los servicios de los objetos semánticos.

Además, el DOO hace mayor incapié en el comportamiento dinámico del sistema

Conceptos sobre diseño : la actividad clave de esta etapa, es la especificación de clases del sistema a construir. La corrección del diseño es fundamental, siendo necesario que este sea “bueno” → eficiente, modificable, etc. Finalmente, el diseño se puede evaluar usando acoplamiento, cohesión y principio “abierto-cerrado”.

Acoplamiento OO

Concepto inter-modular que captura la fuerza de interconexión entre módulos. El objetivo es siempre, lograr un bajo acoplamiento. Existen tres tipos de este último :

Acoplamiento por interacción : ocurre debido a **métodos de una clase que invocan a métodos de otra clase**. Similar al acoplamiento que se produce en diseño orientado a funciones, siendo mayor si :

- Los métodos acceden partes internas de otros métodos.
- Los métodos manipulan directamente variables de otras clases.
- La información se pasa a través de variables temporales.

Consecuentemente, **habrá menor acoplamiento si los métodos se comunican directamente a través de los parámetros** :

- Con el menor número de parámetros posible.
- Pasando la menor cantidad de información posible.
- Pasando sólo datos (y no control).

Acoplamiento de componentes : ocurre cuando una clase A tiene variables de otra clase C, en particular :

- Si A tiene variables de instancia de tipo C.
- Si A tiene parámetros de tipo C.
- Si A tiene un método con variables locales de tipo C.

Cuando A está acoplado con C, también está acoplado con todas sus subclases. Por ende, habrá un menor acoplamiento si las variables de clase C en A son, o bien atributos o bien parámetros en un método.

Acoplamiento de herencia : dos clases están acopladas si una es subclase de otra, la peor forma de acoplamiento es si las subclases modifican la signatura de un método o eliminan un método. También es malo si, a pesar de mantener la signatura de un método se modifica el comportamiento de este. Menor acoplamiento si la subclase sólo agrega variables de instancia y métodos pero no modifica los existentes en la superclase

Cohesión OO

Concepto intra-modular que captura cuán relacionado están los elementos de un módulo. El objetivo es lograr una alta cohesión. Existen tres tipos :

- Cohesión de método.
- Cohesión de clase.
- Cohesión de la herencia.

Cohesión de método : "¿Por qué los elementos están juntos en el mismo método?". **Es mayor si cada método implementa una única función claramente definida con todos sus elementos contribuyendo a implementar esta función**. Además, se debería poder describir con una oración simple que es lo que un método hace.

Cohesión de clase : "¿Por qué distintos atributos y métodos están en la misma clase?". Una **clase debería representar un único concepto** con todos sus elementos contribuyendo a este concepto y **si una clase encapsula múltiples conceptos, la clase pierde cohesión**. **Un síntoma de múltiples conceptos se produce cuando los métodos se pueden separar en diversos grupos**, cada grupo accediendo a distintos subconjuntos de atributos.

Cohesión de herencia : "¿Por qué distintas clases están juntas en la misma jerarquía?". **Existen dos razones para definir subclases** :

- **Generalización-Especialización. → Cohesión más alta.**
- **Reuso.**

Principio abierto-cerrado
"Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas".

- 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.
- Minimiza el riesgo de **"dañar"** la funcionalidad existente al ingresar cambios (lo cual es una consideración muy importante al modificar código).
- Además es positivo para los programadores ya que ellos 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**. La herencia permite crear una nueva subclase para extender el comportamiento sin modificar la clase original.

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.

Metodología de diseño
El punto de partida del diseño OO es el modelo obtenido durante el análisis OO, y usando este modelo se debe producir un modelo detallado final.
La metodología OMT (Object Modeling Technique) involucra los siguientes pasos:

1. **Producir el diagrama de clases**.
2. **Producir el modelo dinámico y usarlo para definir operaciones de las clases.**
3. **Producir el modelo funcional y usarlo para definir operaciones de las clases.**
4. **Identificar las clases internas y sus operaciones.**
5. **Optimizar y empaquetar.**

Producir el modelo dinámico : apunta a especificar c**ómo cambia el estado de los objetos cuando ocurre un evento** (solicitud de operación). **Una secuencia de eventos que ocurren en una ejecución particular del sistema, forman un escenario**. Estos últimos, **permiten identificar los eventos que realizan los objetos**. Se comienza por los escenarios iniciados por eventos externos, modelando los escenarios exitosos y luego los excepcionales. Los distintos **escenarios juntos permiten caracterizar el comportamiento completo del sistema**. Una vez modelados los escenarios se reconocerán eventos de los distintos objetos, utilizando esta información para expandir el diagrama de clases. En

general, para cada evento en el diagrama de secuencia habrá una operación en el objeto sobre el cual el evento es invocado.
Producir el modelo funcional : describe las operaciones que toman lugar en el sistema y especifica cómo computar los valores de salida a partir de los valores de la entrada. No considera los aspectos de control y en OO, las operaciones se realizan sobre objetos (los transformadores del DFD las representan).
Definir las clases y operaciones internas : el diseño final debe ser un plano de la implementación por lo que hay que considerar cuestiones de implementación (incluyendo algoritmos y optimización). **Es importante, evaluar críticamente cada clase para ver si es necesaria en su forma actual y considerar luego las implementaciones de las operaciones de cada clase**. Puede que necesiten operaciones de más bajo nivel sobre clases auxiliares más simples, estas se denominan clases contenedoras.
Optimizar : los siguientes enumerados, ayudan a optimizar el sistema :

- Agregar asociaciones redundantes.
- Guardar atributos derivados.
- Usar tipos genéricos
- Ajustar la herencia

Métricas - WMC
Denominados métodos pesados por claves, en esta métrica **la complejidad de la clase depende de la cantidad de métodos en la misma y su complejidad**. Sean M1 ... Mn los métodos de la clase C en consideración y a C(Mi) la complejidad del método i. Luego, WMC = ∑ C(Mi). Finalmente, **si WMC es alto → la clase es más propensa a errores.**
Métricas - DIT
Corresponde a la profundidad del árbol de herencia, **una clase muy por debajo en la jerarquía de clases puede heredar muchos métodos, y esto dificulta la predicción de sus comportamientos**. **DIT de C es la profundidad desde la raíz, siendo la longitud del camino de la raíz a la clase C**. Si la herencia es múltiple, el camino es más largo. Significativo en detección de **clases propensas a errores** ya que un mayor DIT incrementa la probabilidad de error en esa clase.
Métricas- NOC
Corresponde a la cantidad de hijos, **o subclases inmediatas de C**. Da una idea del **reuso**, a **mayor NOC mayor reuso**. También da la **idea de la influencia directa de la clase C sobre otros elementos de diseño** ya que a mayor influencia mayor importancia en la corrección del diseño de esta clase.
Métricas - CBC
Corresponde al acoplamiento entre clases, es decir, la **cantidad de clases a las cuáles esta clase está acoplada**. Dos clases están acopladas si los métodos de una usan métodos o atributos de la otra y usualmente se puede determinar fácilmente desde el código aunque existen formas indirectas de acoplamiento que no se pueden determinar estáticamente. Por lo que, a menor acoplamiento de una clase hay mayor independencia de la misma y es más fácil de modificar. **Finalmente, a mayor CBC, mayor probabilidad de error en esa clase.**
Métricas - RFC
CBC de C captura el número de clases a la cual C está acoplada, sin embargo, no captura la fuerza de las conexiones. **RFC captura el grado de conexión de los métodos de una clase con otras clases**, **siendo la cantidad de métodos que puede ser invocados como respuesta de un mensaje recibido por un objeto de la clase C**. **Es probable que sea más difícil testear clases con RFC más alto**, siendo muy significativo en la predicción de clases propensas a errores.