4. Diseño

Introducción

Diseño implica que la base de requerimientos esté ya definida (SRS ya hecha). Es previo a la implementación y comprende la parte **creativa** de idear un **plano del sistema** teniendo en cuenta todas las restricciones que tengamos.

Algunos puntos fuertes son:

- Determina las características del sistema: hay cosas que se deciden aquí y que no se pueden cambiar.
- Tiene gran impacto en testing y en mantenimiento

El resultado de esta etapa es el diseño que se va a usar para implementar la solución al problema que tenemos.

Diseño General

Tipos de Diseño

Existen **tres tipos de diseño:**

1. Arquitectónico (el de componente y conectores): identifica los componentes necesarios del sistema, su comportamiento y relaciones.
2. Alto nivel (similar al análisis, tenemos orientado a funciones y orientado a obj): es la vista de módulos del sistema, elegir cuáles módulos habrá, qué deben hacer, cómo se organizan e interconecta.
3. Detallado: cómo se implementan los componentes/módulos de manera que satisfagan sus specs. Es muy cercano al código e incluye algoritmos y estructuras de datos.

Pero, **cómo encontramos el mejor diseño posible?**

Criterios Evaluatorios del Diseño

Normalmente tendremos varios diseños, veremos tres criterios útiles para evaluarlos:

- **Correctitud**: se fija que el diseño implemente los requerimientos del sistema y que sea factible con respecto a las restricciones en la SRS.
- **Eficiencia:** usa los recursos eficientemente? (principalmente CPU y memoria)
- **Simplicidad**: se fija que la comprensión del sistema sea fácil para que el software sea mantenible. Además, facilita el testing, el descubrimiento y

corrección de bugs, y la modificación del código.

Principios de Diseño

Ayudan a analizar qué es lo que se busca con nuestro diseño. En otras palabras, son las guías de las metodologías para un buen diseño. Los más fundamentales son:

1. Partición y Jerarquía:

   basado en *divide y conquistarás*. Se divide el problema en partecitas **manejables** (*manejables:* que dos partes no tengan el mismo grupo de trabajo, y que sean independientes entre sí). Por otro lado, claro que sí debe haber comunicación entre las partes, siempre y cuando un problema no se divida en tantas partes tal que el costo de comunicación entre todas ellas supere al beneficio.

2. Abstracción:

   describe el comportamiento externo sin dar detalles internos de cómo se produce dicho comportamiento. Es como una caja negra que representa a cada componente. En el caso de diseño, no tenemos componentes, por lo que debemos ser creativos al momento de detallar cómo funcionarían. Hay dos tipos de abstracción.

   - Funcional: módulos se tratan como funciones de entrada/salida. Se especifican con pre y postcondiciones.
   - Datos: entidad del mundo real que nos provee servicios al entorno. Se esperan ciertas operaciones de un objeto para pedirle cosas.

3. Modularidad:

   Un sistema es modular si tiene componentes discretas (independientes entre sí) tal que se pueden implementar separadamente y un cambio en una no influya mucho en otra. Esto reduce costos de testing, debugging y mantenimiento.

> 💡 Existen criterios de descomposición ya que no es trivial.

Estrategias en Diseño

Un sistema es una jerarquía de componentes, en específico, para construir un sistema:

- Estrategia top-down: comienza en la componente de más alto nivel hasta la de más bajo nivel.
  - Se comienza con la especificación del sistema
  - Se define el módulo que implementa la especificación
  - Especifica módulos subordinados
  - Se trata a cada módulo como un nuevo problema, de forma iterativa. Hasta llegar a un punto donde el diseño se implemente directamente.
    - Permite a que exista una clara imagen del diseño
    - Es un enfoque natural para manipular problemas complejos
    - La mayoría de metodologías se basan en este enfoque.
    - No se conoce factibilidad hasta el final.
- Estrategia bottom-up: al revés, del más bajo al más alto.

> 💡 Se suele usar más top-down (o una combinación de ambos), pero de todas formas, ninguno de los dos es práctico.

Diseño de Alto Nivel: Orientado a Funciones

El diseño de alto nivel se refiere al diseño dividido en **módulos**: partes lógicas discretas separadas en un programa. Los módulos se eligen siguiendo dos criterios:

- Acoplamiento: caracteriza el vínculo inter-modular, que se refiere a la dependencia que existe entre elementos de módulos distintos a la hora de relacionarse en su funcionamiento.
- Cohesión: caracteriza el vínculo intra-modular, es decir, qué tan cerca están los elementos de un mismo módulo a la hora de su funcionamiento.

En general, se busca disminuir el acoplamiento y aumentar la cohesión. Veamos en más detalle estos dos criterios.

Acoplamiento

Decimos que dos módulos son independientes si cada uno puede **funcionar completamente sin la presencia del otro**. Acoplamiento es el grado de dependencia que hay entre módulos. Que haya muy poco acoplamiento sirve para:

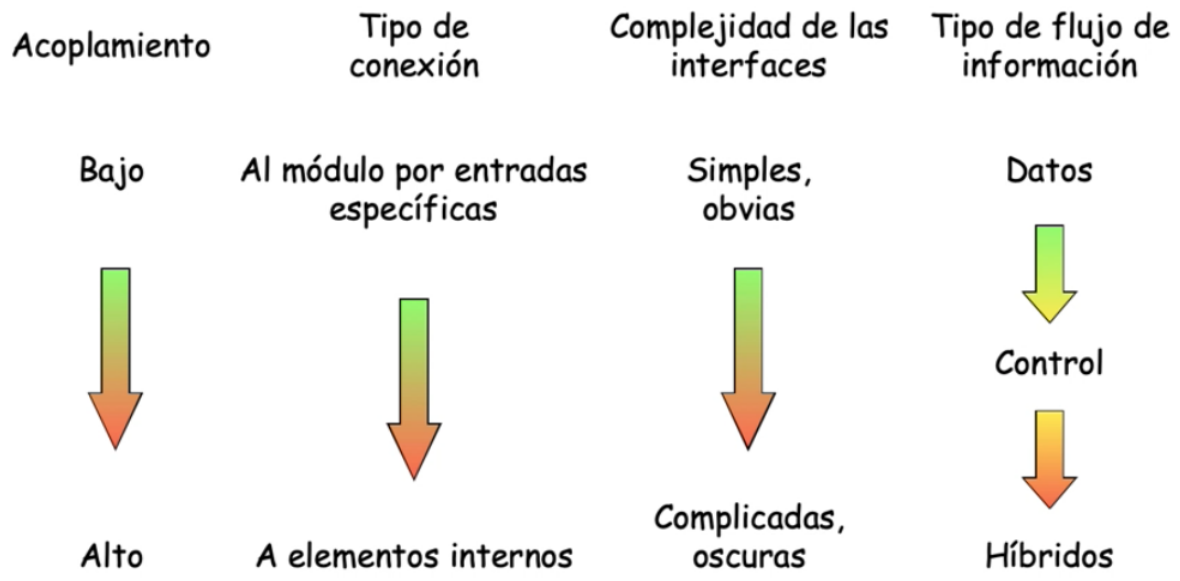
- Se puedan modificar módulos separadamente
- Se puedan testear independientemente
- El costo de programación decrezca

El acoplamiento puede **aumentar** o **disminuir** dependiendo de **tres factores importantes**:

1. Tipo de conexión entre módulos
    a. Los módulos deben tener entradas específicas por la que los otros módulos puedan comunicarse. No se debe acceder a un módulo a través de elementos internos!
    b. No queremos usar operaciones y atributos de un módulo, debe haber encapsulamiento.
    c. Entonces, cada módulo debe tener su respectiva entrada con sus parámetros, que deben ser simples y “mínimos”.
2. La complejidad que tienen las interfaces
    a. Las interfaces deben ser simples y obvias, debemos evitar tener interfaces que sean complejas (relacionado a parámetros y a si se comparten variables o no) y oscuras (que sea difícil obtener en un módulo, un dato a través de una entrada).
    b. Un ejemplo de una interfaz compleja es que se pase como parámetro un registro, cuando el módulo sólo necesita un campo del registro.
3. El tipo de flujo de información.
    a. El tipo de flujo de información entre módulos debe ser idealmente de **datos**, ya que nos importa ver a los módulos como funciones que toman una entrada y devuelven una sálida, nada más que eso.
    b. Ahora, si además de datos, se tiene un flujo de control, que supone el uso de flags, ya tenemos un problema. Esto se debe a que los datos de control hacen que un módulo actúe de forma diferente según la flag. En otras

palabras, no queremos un módulo que tenga un comportamiento distinto según lo que reciba, para eso, dividamos el módulo en dos distintos, cada uno con su propio camino según su entrada.

En resumen:



ESTO ENTRA AL PARCIAL, DEBEMOS EXPLICAR CUÁNDO ES PEOR Y MEJOR EL ACOPLAMIENTO PARA CADA UNO DE LOS TRES FACTORES.

Cohesión

La cohesión es un criterio que va de la mano del acoplamiento, ya que a mayor cohesión, es probable que tengamos bajo acoplamiento.

Este criterio va a representar cuan fuertemente están vinculados los elementos de un mismo módulo.

Tipos de niveles de cohesión (ordenados de menor cohesión, a mayor cohesión):

1. Casual: la relación entre elementos de un módulo que no tiene un significado. Ejemplos:
    a. El programa es “modularizado” cortándolo en pedazos.
    b. Un módulo es creado para evitar código repetido.
2. Lógica: existe alguna relación lógica entre los elementos de un módulo, y cada elemento realiza funciones dentro de la misma clase lógica. Ejemplos:
    a. Tengo un módulo que realiza todas las entradas, y por ende hace uso de flags para encaminar el input a su respectivo output.

3. Temporal: parecida a la lógica, pero los elementos estan relacionados en el tiempo y se ejecutan juntos. Ejemplos:
    a. Inicialización
    b. clean-up
    c. finalización
4. Procedural: tiene elementos que pertenecen a una misma unidad procedural. Ejemplos:
    a. Un ciclo o secuencia de decisiones
5. Comunicacional: elementos relacionados por una referencia a un mismo dato. En general es porque se hace algo con ese dato de entrada. Ejemplos:
    a. “print and punch”
    b. Pedir los datos de una cuenta personal y devolver todos los datos del registro.
6. Secuencial: los elementos están juntos porque la salida de un elemento corresponde a la entrada del próximo. Pueden contener varias funciones o parte de una. Ejemplo:
    a. Quiero pintar un Fiat 600. 1) Limpiar la chapa. 2) Reparar defectos. 3) Lijar. 4) Pintar.
7. Funcional: es la más fuerte, todos los elementos del módulo están relacionados para llevar a cabo una sola función. Ejemplos:
    a. Calcular seno de un ángulo.
    b. Ordenar un arreglo.
    c. Calcular salario neto.
    d. Reservar un asiento en el avión.

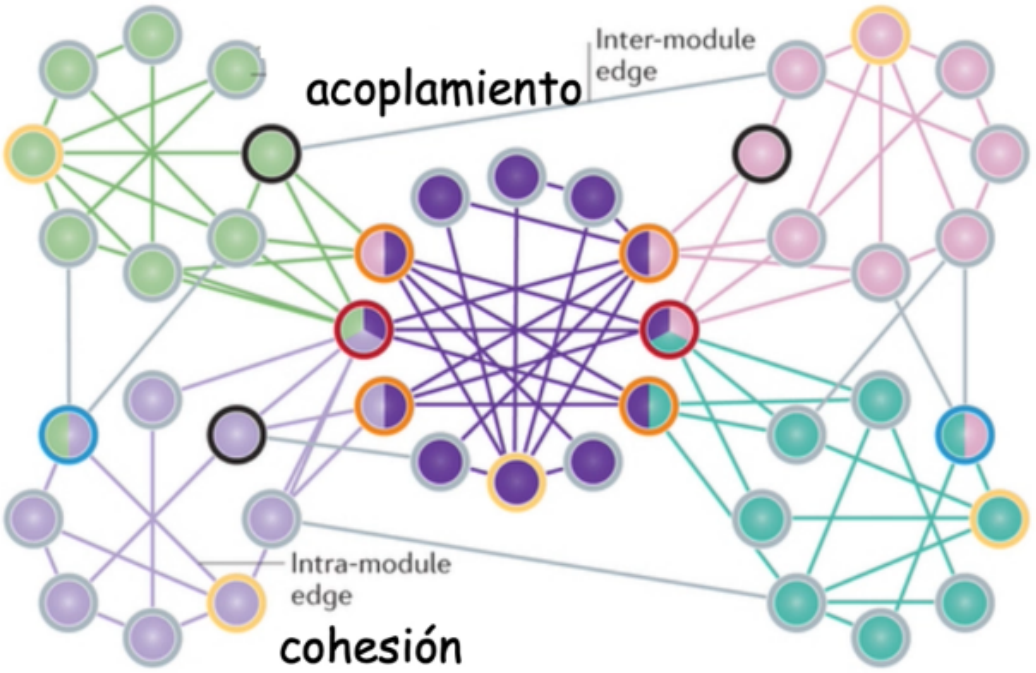
Analizar la cohesión de un módulo (esto no se toma)

Para analizar la cohesión de un módulo, lo que se hace es describir el propósito del módulo en una oración, ya que los módulos cohesivos se describen con oraciones simples. Entonces:

- Si la oración es compuesta, tiene **comas o más de un verbo** ⇒ el módulo esta realizando más de una función y probablemente tenga **cohesión secuencial** o

comunicacional.

- Si la oración contiene palabras relacionadas al **tiempo** (primero, luego, cuando, después) entonces es **cohesión secuencial** o **temporal**.
- Si el predicado de la oración no tiene un único objeto a continuación del verbo (ej: “editar los datos”) ⇒ **cohesión lógica**.
- Palabras como inicializar/limpiar/etc son **cohesión temporal**.



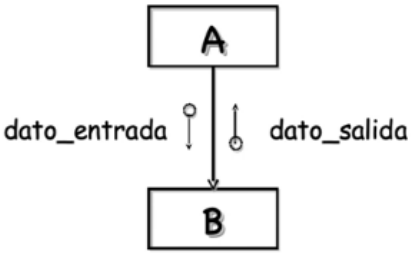
Cada módulo es un color, y cada elemento de cada módulo se relaciona con otro distinto a través de las flechas. Tenemos flechas de un mismo grupo, y flechas que conectan grupos entre sí. En este caso, el módulo violeta tiene cohesión alta, y su acoplamiento con los demás también, por ende esto es malo.

Notación y Especificación del Diseño

Una vez tenemos un diseño producido, es necesario plasmar o especificar ese diseño en un documento de manera precisa. Este documento suele ser un documento textual que incluye notaciones gráficas. Veremos algunas de estas notaciones.

Diagrama de Estructura

Este tipo de notación es gráfica para la estructura de un programa. Representa los módulos y sus interconexiones usando rectángulos y flechas respectivamente.

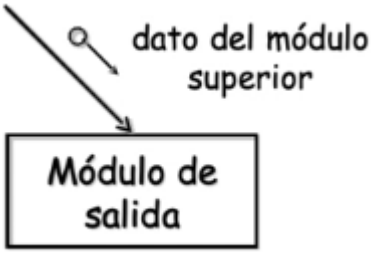


Cada programa tiene su diagrama.

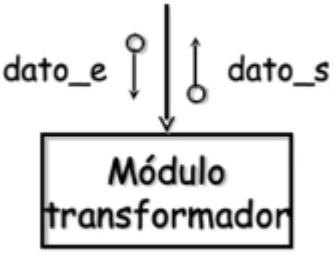
Tipos de Módulos y sus Diagramas

1. Módulo de entrada: recibe datos para el módulo superior.

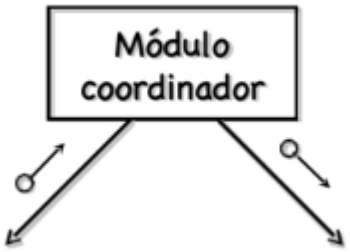
2. Módulo de salida: envía datos del módulo superior.



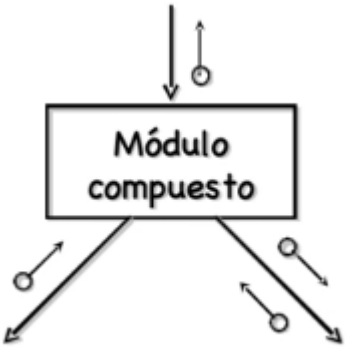
3. Módulos transformadores: reciben, hacen algo, y devuelven



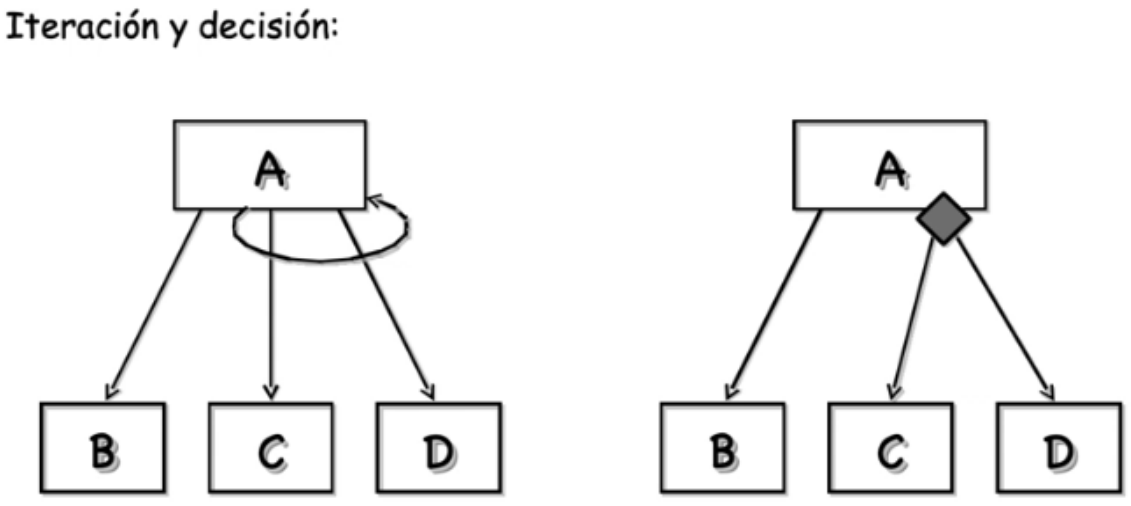
4. Módulo coordinador: recibe y envía



5. Modulo compuesto: módulos que usan otros módulos y que son usados a su vez.



Además, tenemos forma de describir iteraciones y decisiones:



El primero es un ciclo while o for, el rombo es o llamo al módulo C o al D.

> 💡 En estos diagramas sólo se muestra la estructura, no toda la lógica del programa.

Metodología de Diseño Estructurado

La estructura se decide durante el diseño y la misma no debe cambiar en la implementación. Las mismas son controladas por la SDM (metodología de diseño estructura).

Lo que provee la metodología son pautas para auxiliar al diseñador en el proceso de diseño para controlar la estructura o llegar a ella.

En esta metodología, se ve al software como una función de transformación que convierte una E en una S. Usa la abstracción y descomposición funcional. Se specifican módulos de funciones y sus conexiones siguiendo una estructura jerárquica con bajo acoplamiento y alta cohesión:

- Módulos con módulos subordinados no realizan mucha computación (la realizan los módulos subordinados solos).
- El módulo principal se encarga de la coordinación.
- La factorización es el proceso de descomponer un módulo de manera que el grueso de la computación se haga en los módulos subordinados.
- SDM apunta a acercarse a una factorización completa.

Pasos principales de la metodología:

1. **Reformular el problema como un DFD**

   El DFD debe capturar el flujo de datos del sistema propuesto. Este diagrama proveerá una visión de alto nivel del sistema, en donde se deben identificar las E, S, transformadores y sumideros. No se muestra la lógica de control (ni loops, ni condiciones).

   > 💡 Se usan estas notaciones en el DFD
   > +: decisión
   > *: unión de dos entradas.

2. **Identificar las entradas y salidas más abstractas**

Se trata sobre identificar los formatos intermedios que procesan las entradas y salidas en un sistema. Ejemplo: muchas veces, recibimos un dato que no es exactamente con lo que trabajamos, normalmente se “limpia” este input, o se parsea (caso de querer extraer palabras donde la E es un archivo), etc. Ese trabajo es a lo que hace referencia este paso. Tendremos dos tipos de entradas:

- MAI: entradas más abstractas. Son elementos de datos en el DFD que están mas distantes de la entrada real, pero que todavía se pueden considerar como entradas.
- MAO: Es dual para obtener las salidas más abstractas.

Para encontrar ambas, se va desde la entrada/salida física en dirección a la salida/entrada hasta que los datos no puedan considerarse entrantes.

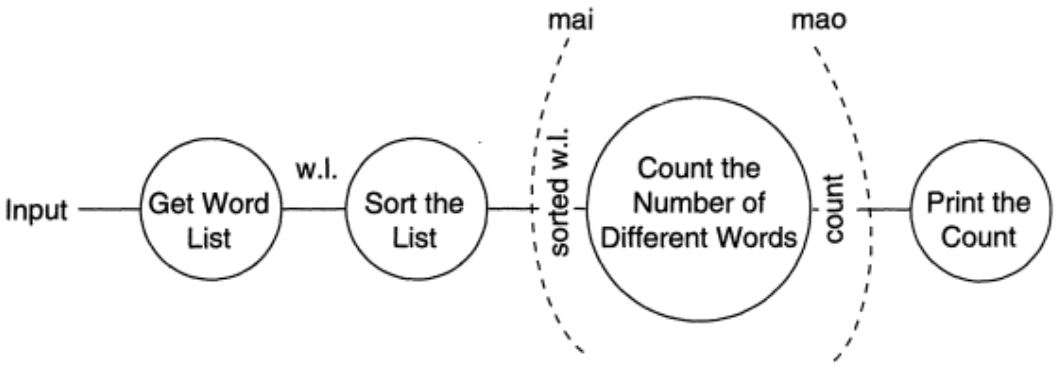
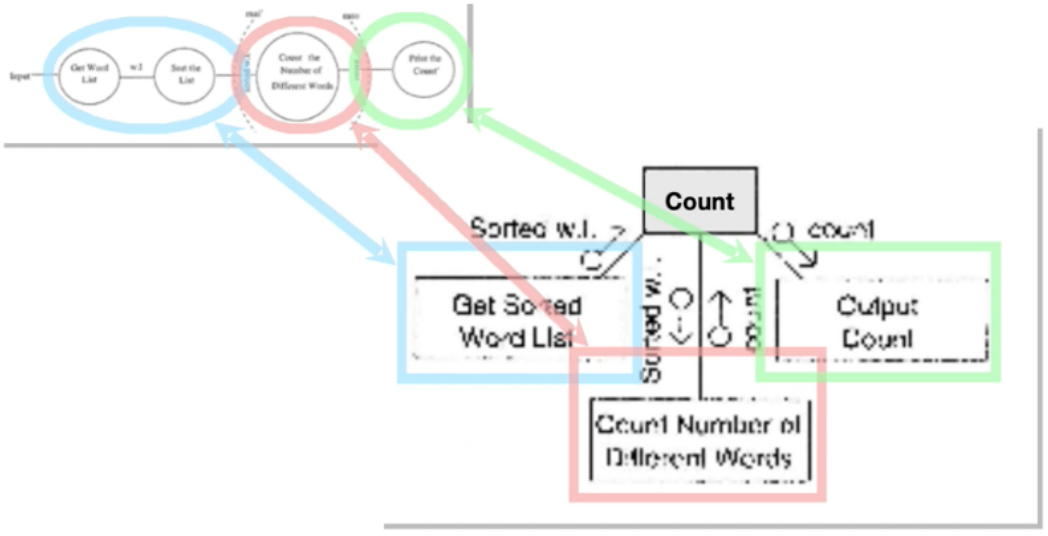


FIGURE 5.5. DFD for the word-counting problem.

3. **Hacer el 1er nivel de factorización**

   Aquí se especifica el módulo principal y un módulo subordinado por cada ítem de dato de la MAI (para enviar al módulo principal) y otro para la MAO (para obtener del modulo principal). Luego, se especifica un módulo transformador subordinado por cada transformador central. Las E y S de estos módulos transformadores están especificadas en el DFD.

   Entonces, el 1er nivel de factorización se va a dar por la división entre las MAI y las MAO.



4. **Factorizar los módulos de E y S y transformadores**

   Se repite el proceso de factorización pero aquí el módulo principal pasa a ser el módulo de E o S, depende el caso. En el caso de E, no hay módulo de salida. En el caso de S, no hay módulo de E. Este proceso se repite hasta llegar al módulo (burbuja) de entrada física, en donde no hay más divisiones.

5. **Mejorar la estructura (usando heurísticas, análisis)**

   Estos pasos, como dijimos, no son tan mecánicos. Son sensibles a modificación según la necesidad que se tenga. Como el objetivo siempre es disminuir acoplamiento y aumentar cohesión, se usan heurísticas de diseño.

Heurísticas de Diseño

Son un conjunto de reglas que dan como un OK sobre el diseño que se está teniendo.

1. Tamaño del módulo: indicador de la complejidad del módulo. Examinar cuidadosamente los módulos con muy poquitas líneas (tal vez no deberían

existir por sí solos) o con más de 100 líneas (tal vez son demasiado complejos).

2. Cantidad de flechas de salida y cantidad de flechas de llegada. Las flechas de salida no deben exceder 5 o 6 flechas; la segunda debería maximizarse.
3. Alcance del efecto de un módulo: los módulos afectados por una decisión que fue tomada en otro.
4. Alcance del control de un módulo: todos los subordinados

(3+4). Por cada módulo, el alcance del efecto debe ser un subconjunto del de control: los módulos afectados por una decisión en otro, deben ser los subordinados.

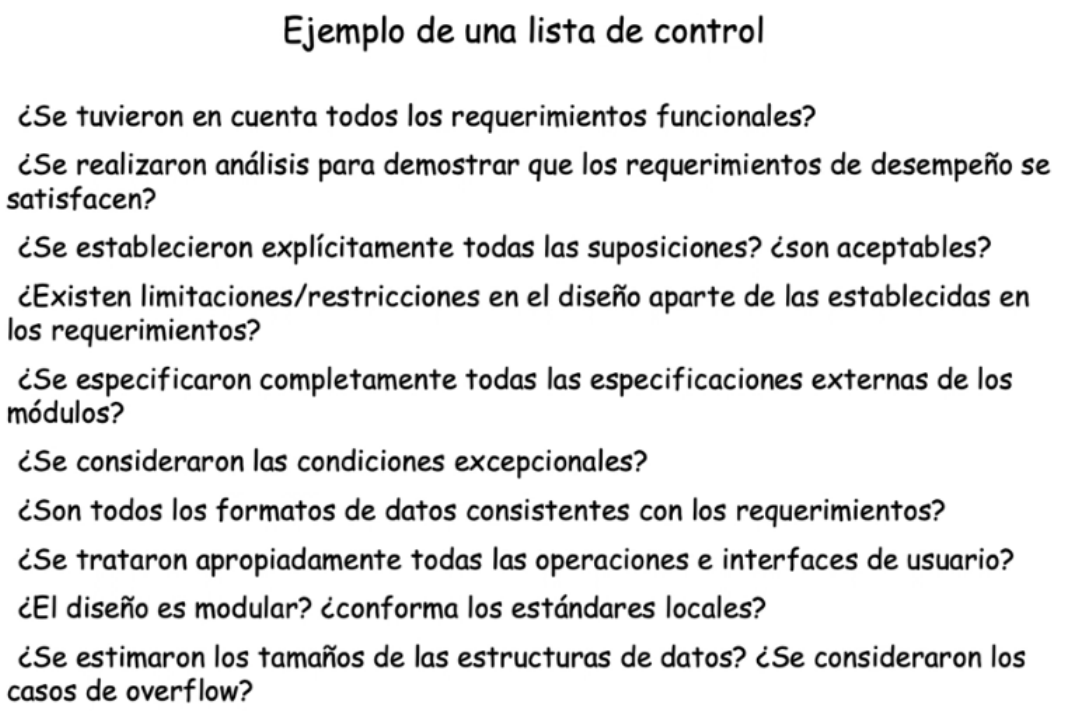
> 💡 Idealmente, una decisión debe tener efecto a los subordinados inmediatos, no más allá.

Verificación de Diseño

Para verificar que un diseño es correcto, es decir, que implemente los requerimientos, lo que se hace es:

- Se hacen análisis de desempeño, eficiencia, etc
- Si se usan lenguajes formales, usaremos las herramientas que asisten para verificar
- Se hace una **revisión** (lo vemos más adelante). Se usan listas de control.
- Chequeamos la modularidad (ya que hay una relación fuerte entre modularidad-calidad)

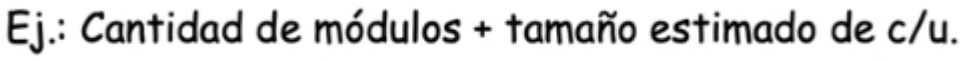
Listas de Control



Métricas

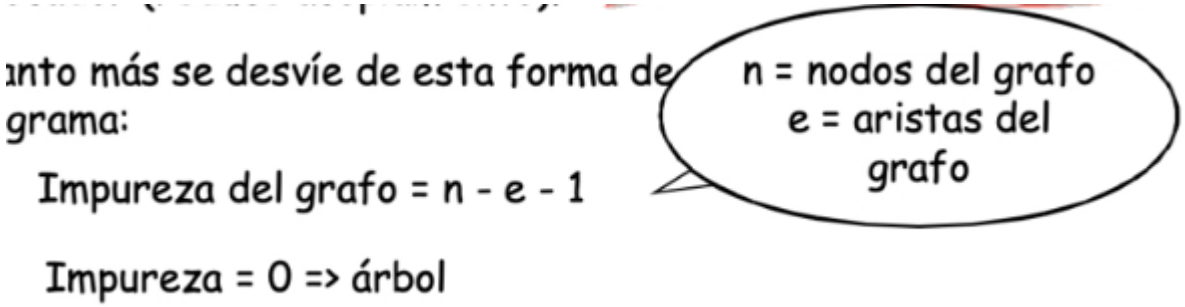
Nos proveen una evaluación cuantitativa del diseño (números) para poder mejorar mi producto.

Una métrica principal es el tamaño de nuestro sistema, que se puede estimar después de Diseño.



Métricas de Red

Se enfoca en la estructura del diagrama de estructuras. En esta métrica, se considera un buen diagrama a aquel en el cual cada módulo tiene sólo un módulo invocador: mientras nuestro diagrama sea más parecido a un árbol, mejor. Si nos desviamos de la forma de un árbol, más acoplamiento habrá (más impuro será).



Mientras más negativo el valor, más impuro el diagrama.

Métrica de Estabilidad

Captura el impacto de los cambios en el diseño. Mientras más estabilidad, mejor será la métrica.

Definiremos a la estabilidad de un módulo como **la cantidad de suposiciones por otros módulos sobre éste**.

> 💡 La estabilidad de un módulo depende de la interfaz del mismo y el uso de datos globales.

Métricas de Flujo de Información

Como vimos anteriormente, el acoplamiento también puede aumentar por la complejidad de una interfaz. Las métricas de flujo de info lo que hacen es tener esto en cuenta. Entonces, una forma de medir la cantidad de información que fluye hacia adentro y hacia afuera en un módulo es:

- Medir la complejidad intra-módulo, que se estima con el tamaño del módulo en LOC.
- y medir la complejidad inter-módulo, que se estima con:
  - Inflow: flujo de info entrante al módulo
  - Outflow: flujo de info saliente del módulo.

> 💡 Ambos puntos ayudan a medir el flujo de información. No son cosas distintas, si no la suma.

Entonces, definiremos a la complejidad del diseño del módulo C como:

$$DC = tamaño * (inflow * outflow)^2$$

$inflow * outflow$: total de combinaciones de entrada y salida.
Esto, en la fórmula se hace al cuadrado porque representa la importancia de la interconexión entre módulos con respecto a la complejidad interna (ej: el tamaño)

Luego, hay otra medida que mide la complejidad del diseño del módulo C pero también tiene en cuenta la cantidad de módulos desde y hacia donde fluye la info, no sólo la cantidad de flujo en las entradas y salidas. Se mide 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.

Ahora, cómo usamos estas métricas?

- Podemos decir que un módulo es propenso a tener errores si:

$$DC > complej\_media + desv\_std$$

- Decimos que un módulo es complejo si

$$complej\_media < DC < complej\_media + desv\_std$$

- C.C. normal.

Diseño de Alto Nivel: Orientado a Objetos

Aquí se ven los datos y las funciones como un todo. El propósito de OO es definir clases del sistema y así construir las relaciones que tenemos entre las clases.

Las técnicas OO para la construcción de las relaciones, se pueden usar tanto para el análisis como para el diseño, la diferencia es que AOO modela el problema mientras que DOO modela la solución.

Otra diferencia importante entre AOO y DOO es el tipo de objetos que manipulan:

- **AOO**: objetos semánticos
- **DOO**: interfaces (de usuario), apps (especifican mecanismos de control) y objetos de utilidad (necesarios para soportar servicios de objetos semánticos como árboles, pilas, diccionarios).

Clases y Objetos

La propiedad básica de los objetos es el **encapsulamiento**, en donde tenemos que la información es oculta al usuario a través de interfaces que se usan para acceder y modificar la información.

Los objetos tienen un estado persistente, que quiere decir que retienen la información en el transcurso del tiempo (cosa que no pasa con funciones ni procesos). Además, los objetos tienen **identidad**.

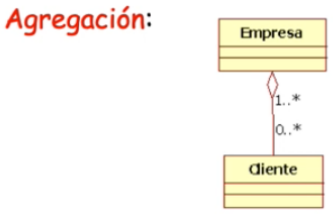
- Clases: plantilla de la cual se crean los objetos, define estructura y servicios. Tiene una interfaz y su cuerpo implementa operaciones que pueden ser *públicas, privadas o protegidas.* Definen un tipo.
- Objetos: son instancias de clases

Relaciones entre objetos

1. Los objetos puede estar vinculado con otro por un tiempo, a esto se lo llama **asociación**.

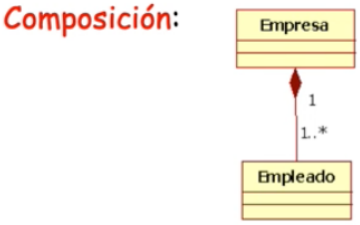
   Un objeto envía un mensaje a otro solicitando un servicio. El objeto receptor invoca al método que implementa tal servicio.

2. Si un objeto es parte de otra clase, a esto se lo llama **agregación** (relación derivada de una asociación). El ciclo de vida de los objetos no esta relacionado.



Rombo sin relleno

3. Si un objeto esta compuesto por otro, se lo llama **composición**. El ciclo de vida de los objetos esta muy relacionado.



Rombo relleno con color

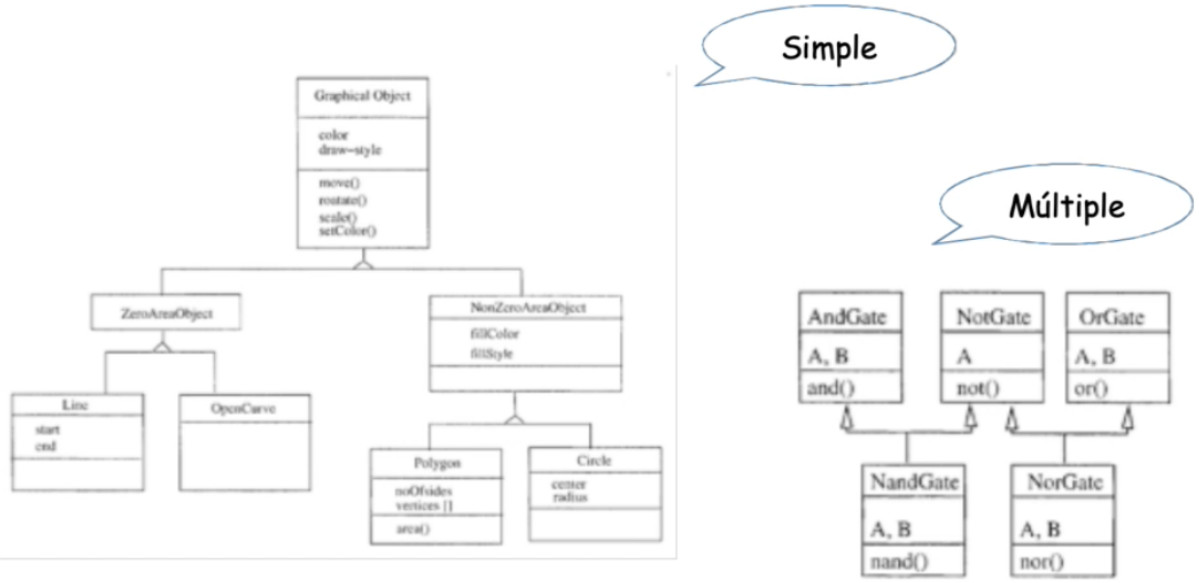
Herencia y Polimorfismo

Concepto único de OO. Es una relación entre clases que permite la definición e implementación de una clase basada en la definición de otra.

→ Podemos decir que una clase hereda de otra si la primera (SUBCLASE) hereda todos los atributos y operaciones de la segunda (SUPERCLASE).

> 💡 Las subclases pueden tener una parte incremental, es decir, operaciones o atributos no heredados, nuevos.

La herencia es una relación “es-un”, y forma una jerarquía entre las clases. Estas jerarquías pueden ser **simples** (se hereda de una superclase) o **múltiple** (de varias superclases).



No se recomienda la herencia múltiple.

Además, tenemos que la herencia va a inducir **polimorfismo**: un objeto puede ser de distinto tipo pero no tener conflicto en hacer operaciones del tipo de mi superclase.

Conceptos de Diseño

En OO, el corazón de nuestro diseño serán las clases, por lo que la especificación de las mismas es una parte fundamental. Cómo en orientado a funciones, también podemos tener diseños buenos o malos, relacionados con su eficiencia, modificabilidad, estabilidad, etc, por lo que también tendremos criterios para evaluar esos diseños según:

- Acoplamiento
- Cohesión
- Principio abierto-cerrado

Acoplamiento

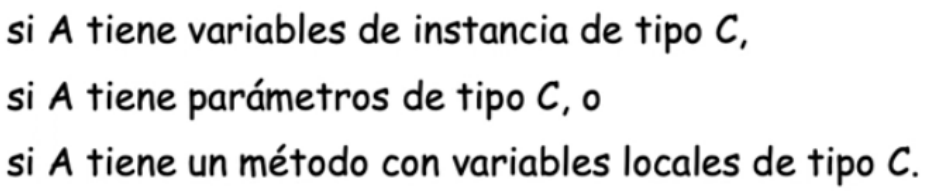
Lo mismo que en orientado a funciones. La diferencia es que en OO tenemos tres tipos de acoplamiento:

1. Por interacción: ocurre debido a métodos de una clase que tocan a métodos de otra clase.

a. Mayor acoplamiento si los métodos acceden a partes internas de otros métodos, si manipulan variables directamente, y si la información se pasa por variables temporales.

b. Menor acoplamiento si se tiene un número mínimo de parámetros, si pasamos la menor cantidad de info posible, y también, pasando sólo datos, no control (como en funciones).

2. Por componente: ocurre cuando una clase A tiene variables de otra clase C en particular:



a. Mayor acoplamiento: si las variables de clase C en A son, o bien atributos, o bien parámetros en un método.

3. Herencia: siempre que haya herencia hay acoplamiento! Queremos hacer herencia de forma correcta.

a. Mayor acoplamiento si una subclase quiere modificar/eliminar métodos heredados

b. Menor acoplamiento si la subclase sólo agrega variables de instancia y métodos distintos y nuevos!

Cohesión

Igual que en funciones. Tenemos tres tipos de cohesión:

1. De método: cohesión mayor si cada método implementa única y exclusivamente una función. Se debería poder escribir con una oración simple lo que hace el método.
2. De clase: una clase debería representar un único concepto con todos sus elementos. Si se encapsula muchos conceptos, la clase pierde cohesión.
3. De herencia: tenemos más cohesión si la jerarquía de herencia se produce por generalización-especialización y no por reuso.

Principio abierto-cerrado

"Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas" (B. Meyer).

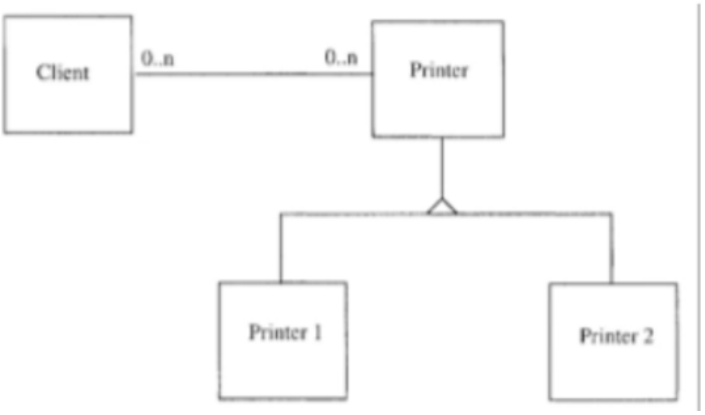
El principio consiste en que el código pueda ser extensible para adaptar el sistema a nuevos requerimientos, pero el código que ya existe no debe ser modificado. Hay que minimizar el riesgo de añadir código nuevo!

En OO este principio es satisfecho si se usa de forma correcta la herencia y polimorfismo. Recordemos que con herencia podemos crear nuevos comportamientos sin modificar métodos bases!

Ej.: Un objeto cliente que interactúa con un objeto impresora:

Client llama directamente a los métodos de Printer1.

Client no necesita ser modificado.



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 se sigue este principio (Liskov), se cumple con el de abierto-cerrado!

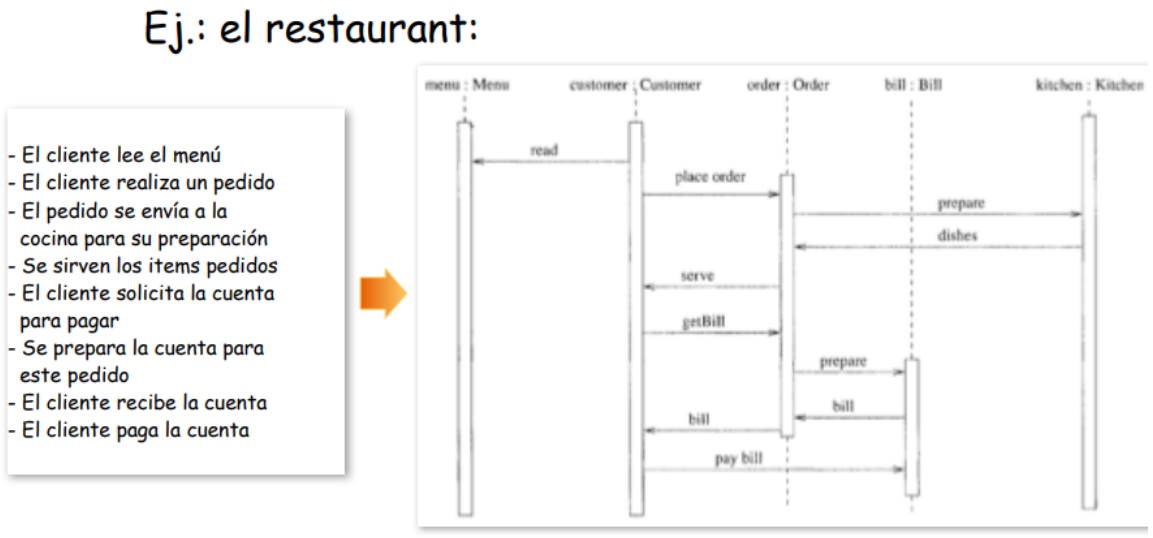
Metodología de Diseño

La idea es partir desde el modelo obtenido en el análisis OO y con ese, hacer un modelo detallado final. Se usa una metodología llamada OMT (object modelling technique) que involucra los siguientes pasos:

1. Hacer diagrama de clases (el obtenido en análisis OO)
2. Hacer modelo dinámico y usarlo para definir ops de las clases.

   Esto es porque el diagrama de clases es estático, entonces con el dinámico podemos especificar cómo cambia el estado de los objetos cuando sucede un evento (solicitud de operación). Para hacerlo, comenzaremos por:

   - Escenarios iniciados por eventos externos, partiendo por los exitosos y luego los de excepciones. Podemos graficar los escenarios usando un diagrama de secuencia.



Todo lo que pasa se expresa en este gráfico.

> 💡 Le decimos escenarios a una secuencia de eventos que ocurre en una ejecución particular del sistema (recordar casos de uso).

   - Una vez tenemos los escenarios, y un diagrama de secuencia como el de arriba, podremos refinar la visión que tenemos de los objetos y agregar operaciones necesarias que puede que nos hayamos salteado (ej: placeOrder, getBill de el restaurante).

3. Hacer el modelo funcional y usar para definir ops de las clases

Describe las ops que toman lugar en el sistema. Especifica como computar valores de S a partir de una E. Representamos a las funciones como los transformadores en los DFD.

4. Identificar clases internas y sus ops.

   Acá definimos algoritmos y optimizaciónes + evaluar cada clase para ver si son necesarias tal como están o si puede modificarse. Luego, consideraremos las implementaciones de las ops de cada clase.

5. Optimizar y empaquetar

   Se agregan asociaciones redundantes para optimizar accesos a datos. También podemos guardar atributos derivados para no repetir cálculos complejos. También usar tipos genéricos para reusar código (no tipos muy complejos). Finalmente, podemos ajustar la herencia a través del análisis de la jerarquía, la generación de clases abstractas por sobre otras clases, o mejorar reusabilidad.

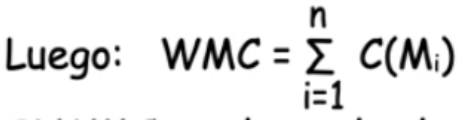
Métricas

Hay varias métricas en OO:

Métodos Pesados por Clases (WMC)

La complejidad de la clase depende de la cantidad de métodos en la misma y su complejidad.

Sean $M_1, \dots, M_n$ los métodos de una clase C, la complejidad de la clase C:



Profundidad del Árbol de Herencia (DIT)

DIT de una clase C es la profundidad de la clase C a la raíz, y mide la cantidad de métodos que hereda una clase, ya que mientras mayor el DIT mayor probabilidad de errores en la clase.

Mientras más herencia + DIT.

Cantidad de Hijos (NOC)

Mide la cantidad de subclases inmediatas de una clase C. Mide el reuso: mayor NOC, mayor reuso.

Da una idea sobre la influencia directa de la clase C sobre otros elementos de diseño.

> influencia => > importancia en la corrección del diseño de esta clase.

Acoplamiento Entre Clases (CBC)

Cantidad de clases a las cuales una clase C esta acoplada: si los métodos de una usan métodos de otra.

Usualmente se puede determinar fácilmente desde el código.

< acoplamiento de una clase => > independencia de la clase => más fácilmente modificable.

> CBC => > probabilidad de error en esa clase.

Respuesta para una Clase (RFC)

CBC de C captura el número de clases a la cual C está acoplada, pero no captura la fuerza de la conexión. Entonces, RFC captura el grado de conexión tal que:

RFC de una clase C es la cantidad de métodos que pueden ser invocados como respuesta de un mensaje recibido por un objeto de la clase C.

Nos dará la cantidad de todos los métodos de C + los métodos de otras clases que reciben un mensaje de un método de C.

Si RFC es alto, será más difícil testear clases.

Una vez finalizado el mock de clases, estas métricas nos ayudarán a implementar clases con cuidado y a atender a aquellas que puedan generar más errores que otras (a través del uso de mecanismos específicos).