

Ingeniería del Software I

5 - Diseño (Capítulo 6-7)

Diseño del software

- Comienza una vez que los requerimientos están definidos.
- Se realiza 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 para implementar el sistema.

Diseño del software

“There are two ways of constructing a software design: one way is to make it so **simple** that there are obviously no deficiencies and the other way is to make it so **complicated** that there are no obvious deficiencies. The first method is far more difficult... It requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late...”

C. A. R. Hoare ('80)

Diseño del software

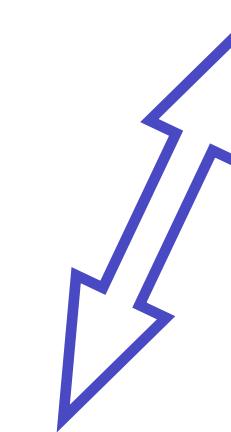
- El diseño es una actividad **creativa**.
- Objetivo: crear un “plano del sistema” que satisfaga los requerimientos.
- Quizás sea la actividad más crítica durante el desarrollo del sistema.
- El diseño determina las mayores características de un sistema.
- Tiene un gran **impacto en testing** y **mantenimiento**.
- 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/interconectan.
- Diseño detallado o diseño lógico:
 - Establece cómo se implementan las componentes/módulos de manera que satisfagan sus especificaciones.
 - Incluye detalles del procesamiento lógico (i.e. algoritmos) y de las estructuras de datos.
 - Muy cercano al código.

Ya lo vimos

Orientado a funciones
Orientado a objetos



Criterios para evaluar el diseño

Objetivo: encontrar el mejor diseño posible.

- Se deberán explorar diversos diseños alternativos.
- Los criterios de evaluación son usualmente subjetivos y no cuantificables.
- Principales **criterios** para evaluar el diseño:
 - Corrección
 - Eficiencia
 - Simplicidad

Criterios para evaluar el diseño

- Corrección:
 - Es fundamental (¡pero no el único!)
 - ¿el diseño implementa los requerimientos?
 - ¿es factible el diseño dada las restricciones?
- Eficiencia:
 - Le compete el uso apropiado de los recursos del sistema (principalmente CPU y memoria).
 - Debido al abaratamiento del hardware toma un segundo plano.
(OJO: muy importante en ciertas clases de sistemas, ej.: sistemas integrados o de tiempo real).

Criterios para evaluar el diseño

Simplicidad:

- Tiene impacto directo en mantenimiento.
- El mantenimiento es caro.
- Un diseño simple facilita la comprensión del sistema => hace al software mantenible.
- Facilita el testing.
- Facilita el descubrimiento y corrección de bugs.
- Facilita la modificación del código.

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

Principios fundamentales de diseño

El diseño es un proceso creativo.

No existe una serie de pasos que permitan derivar el diseño de los requerimientos.

Sólo hay principios que guían el proceso de diseño.

Principios fundamentales que guían el diseño:

- Partición y jerarquía
- Abstracción
- Modularidad

Estos principios forman la base de la mayoría de las metodologías de diseño.

Principios de diseño

Partición y jerarquía

- Principio básico: “divide y conquistarás”
- Dividir el problema en pequeñas partes que sean manejables:
 - Cada parte debe poder **solucionarse** separadamente.
 - Cada parte debe poder **modificarse** separadamente.
- Las partes no son totalmente independientes entre sí: deben comunicarse/ cooperar para solucionar el problema mayor.
- La comunicación agrega complejidad.
- A medida que la cantidad de componentes aumenta, el costo del particionado (incluyendo la complejidad de la comunicación) también aumenta.
- Detener el particionado cuando el costo supera al beneficio.

Principios de diseño

Partición y jerarquía

- Tratar de mantener la mayor **independencia** posible entre las distintas partes:
Simplifica el diseño y facilita mantenimiento.
- El particionado del problema determina una jerarquía de componentes en el diseño.
- Usualmente la jerarquía se forma a partir de la relación “es parte de”.

Principios de diseño

Abstracción

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

Principios de diseño

Abstracción

Abstracción durante el proceso de diseño:

- Las componentes no existen.
- Para decidir como 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 para solucionar las partes separadamente.

Principios de diseño

Abstracción

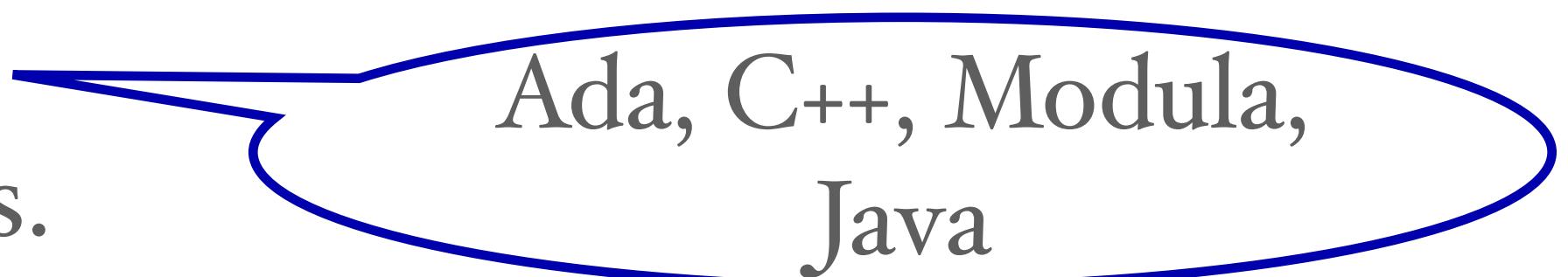
- Dos mecanismos comunes de abstracción:
 - Abstracción funcional.
 - Abstracción de datos.
- Abstracción funcional:
 - Especifica el comportamiento funcional de un módulo.
 - Los módulos se tratan como funciones de entrada/salida.
 - La mayoría de los lenguajes proveen características para soportarla.
Ej.: procedimientos, funciones.
 - Un módulo funcional puede especificarse usando pre y postcondiciones.
 - Forma la base de las metodologías orientadas a funciones.

Principios de diseño

Abstracción

Abstracción de datos:

- Una entidad del mundo real provee servicios al entorno.
- Es el mismo caso para las entidades de datos: se esperan ciertas operaciones de un objeto de dato.
- Los detalles internos no son relevantes.
- La abstracción de datos provee esta visión:
 - Los datos se tratan como objetos junto a sus operaciones.
 - Las operaciones definidas para un objeto sólo pueden realizarse sobre este objeto.
 - Desde fuera, los detalles internos del objetos permanecen ocultos y sólo sus operaciones son visibles.
- Muchos lenguajes soportan abstracción de datos.
- Forma la base de las metodologías orientadas a objetos.



Ada, C++, Modula,
Java

Principios de diseño

Modularidad

Un sistema se dice modular si consiste de componentes discretas tal que puedan implementarse separadamente y un cambio a una de ellas tenga mínimo impacto sobre las otras.

- Modularidad:
 - Provee la abstracción en el software.
 - Es el soporte de la estructura jerárquica de los programas.
 - Mejora la claridad del diseño y facilita la implementación.
 - Reduce los costos de testing, debugging y mantenimiento.
- No se obtiene simplemente recortando el programa en módulos.
- Necesita criterios de descomposición: resulta de la conjunción de la abstracción y el particionado.

Principios de diseño

Estrategias top-down y bottom-up

Un sistema es una jerarquía de componentes.

- Dos enfoques para diseñar tal jerarquía:
 - Top-down:
 - comienza en la componente de más alto nivel, la más abstracta;
 - prosigue construyendo las componentes de niveles más bajos descendiendo en la jerarquía.
 - Bottom-up:
 - comienza por las componentes de más bajo nivel en la jerarquía, las más simples;
 - prosigue hacia los niveles más altos hasta construir la componente más alta.

Principios de diseño

Estrategias top-down y bottom-up

Refinamiento paso a paso

Enfoque top-down:

- El diseño comienza con la especificación del sistema.
- Define el módulo que implementará la especificación.
- Especifica los módulos subordinados.
- Luego, iterativamente, trata cada uno de estos módulos especificados como el nuevo problema.
- El refinamiento procede hasta alcanzar un nivel donde el diseño pueda ser implementado directamente.
- En cada paso existe una clara imagen del diseño.
- Enfoque más natural para manipular problemas complejos.
- La mayoría de las metodologías de diseño se basan en este enfoque.
- La factibilidad es desconocida hasta el final.
- Top-down o bottom-up puros no son prácticos.
- En general se utiliza una combinación de ambos.

Pros

Cons

Nota:

Los conceptos tratados anteriormente se aplican al diseño de alto nivel en general.

En lo que sigue nos enfocaremos en el diseño orientado a funciones.

Ingeniería del Software I

4 – Diseño orientado a función

Conceptos a nivel de módulo

- Un módulo es una parte lógicamente separable de un programa.
- Es una unidad **discreta e identifiable** respecto a la compilación y carga.
Ej.: macro, función, procedimiento, “package”.
- Criterios utilizados para seleccionar módulos que soporten abstracciones bien definidas y solucionables/modificables separadamente:
 - Acoplamiento
 - Cohesión

Conceptos a nivel de módulo

Acoplamiento

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

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

Conceptos a nivel de módulo

Acoplamiento

Objetivo: los módulos deben estar tan **débilmente acoplados** como sea posible.

- Cuando sea posible => 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 **inter-modular**.
- Factores más importantes que influyen en el acoplamiento:
 - Tipo de conexiones entre módulos.
 - Complejidad de las interfaces.
 - Tipo de flujo de información entre módulos.

Ej.: ¿Utilizo sólo las interfaces especificadas o también uso datos compartidos?

Ej.: ¿Estoy pasando sólo los parámetros necesarios?

Ej.: ¿Estoy pasando parámetros de control (ej.: flag)?

Conceptos a nivel de módulo

Acoplamiento

El acoplamiento entre módulos queda definido por la “fuerza de conexión” entre dichos módulos.

- En general: 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.

Conceptos a nivel de módulo

Acoplamiento

La complejidad y oscuridad de las interfaces de un módulo, i.e. el **tipo de conexión** incrementan el acoplamiento.

- Minimizar la cantidad de interfaces por módulo.
- Minimizar la complejidad de cada interfaz.
- El acoplamiento disminuye si:
 - Sólo las entradas definidas en un módulo son utilizadas por otros.
 - La información se pasa exclusivamente a través de parámetros.
- El acoplamiento se incrementa si:
 - Se utilizan interfaces indirectas y oscuras.
 - Se usan directamente operaciones y atributos internos al módulo.
 - Se utilizan variables compartidas.

Conceptos a nivel de módulo

Acoplamiento

El acoplamiento se incrementa con la **complejidad de cada interfaz**, ej.: cantidad y complejidad de los parámetros.

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

Conceptos a nivel de módulo

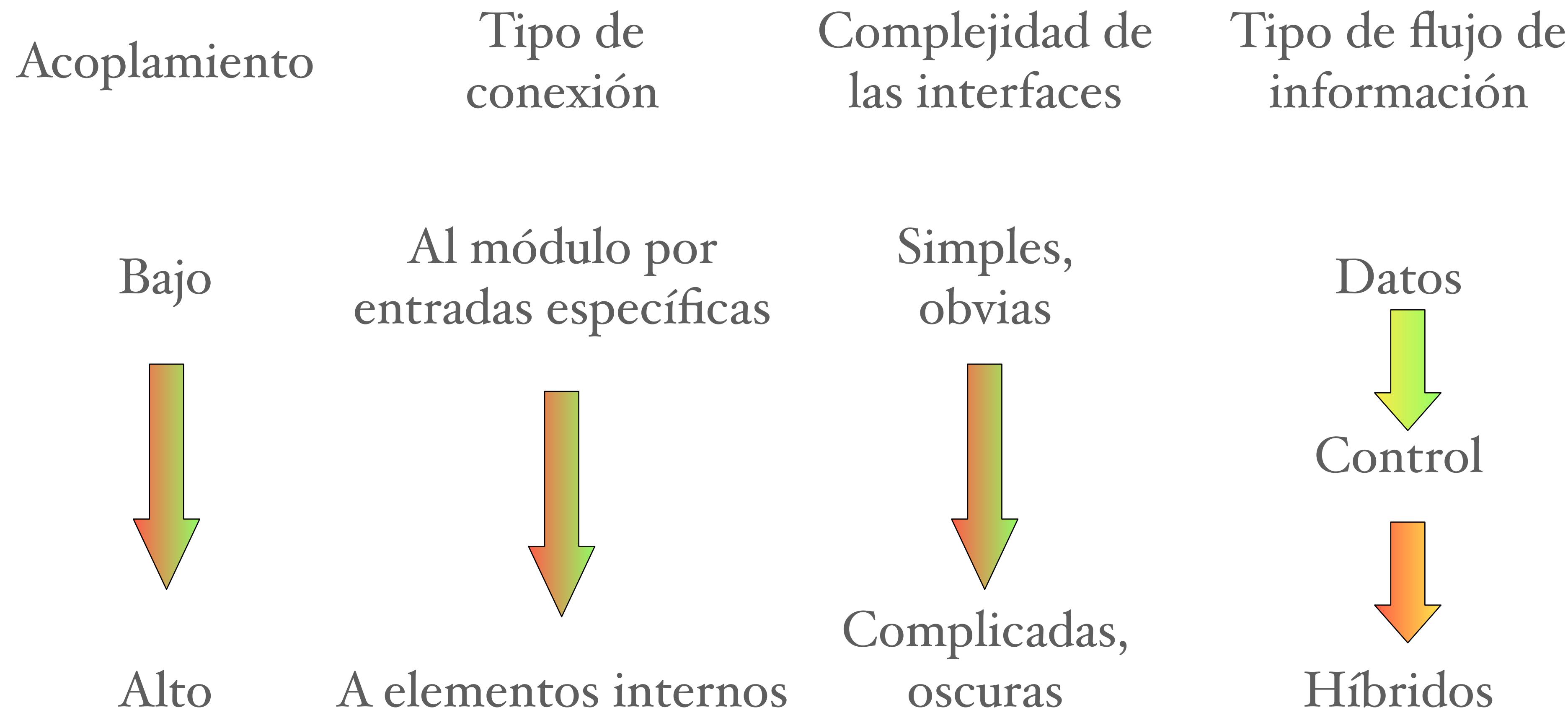
Acoplamiento

El acoplamiento depende del **tipo del flujo de información**.

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

Conceptos a nivel de módulo

Acoplamiento



Conceptos a nivel de módulo

Cohesión

El acoplamiento caracteriza el vínculo inter-modular.

- Se reduce minimizando las relaciones entre los elementos de los distintos módulos.

Otra forma de lograr un efecto similar es maximizando las relaciones entre los elementos del mismo módulo.

- La cohesión considera esta relación.
- La cohesión caracteriza el vínculo **intra-modular**.
- Con la cohesión intentamos capturar cuan cercanamente están relacionados los elementos de un módulo entre sí.

Buscamos: menor acoplamiento y mayor cohesión.

Conceptos a nivel de módulo

Cohesión

La cohesión de un módulo representa cuán fuertemente vinculados están los elementos de un módulo.

- Da una idea de si los distintos elementos de un módulo tienen características comunes.
- Objetivo: **Alta cohesión**.
- La cohesión y el acoplamiento están correlacionados.
- Usualmente, a mayor cohesión de los módulos, menor acoplamiento entre los módulos.
- Pero la correlación no es perfecta.

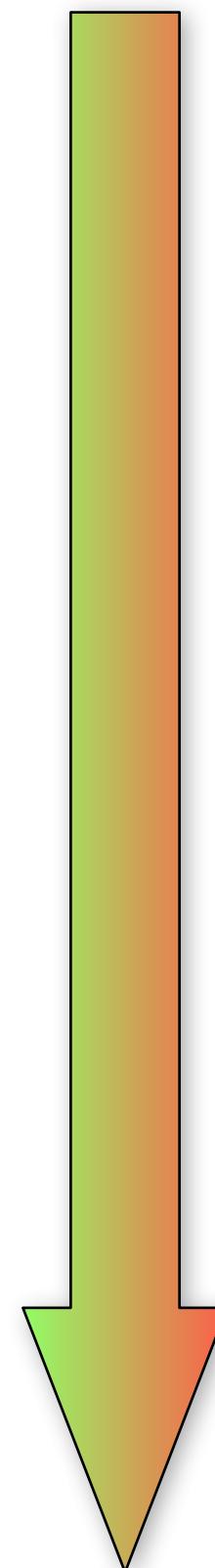
Conceptos a nivel de módulo

Cohesión

Hay varios niveles de cohesión:

- Casual
- Lógica
- Temporal
- Procedural
- Comunicacional
- Secuencial
- Funcional

Más débil



La escala NO
es lineal

Más fuerte

Conceptos a nivel de módulo

Hay varios niveles de cohesión.

- Casual
- Lógica
- Temporal
- Procedural
- Comunicacional
- Secuencial
- Funcional

Cohesi

La relación entre los elementos del módulo no tiene significado.

Ej.: (1) El programa es “modularizado” cortándolo en pedazos. (2) Un módulo es creado para evitar código repetido

Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.
Ej.: Un módulo que realiza todas las entradas. =>
Necesito flags para indicar que tipo de registro quiero imprimir

Más fuerte

Parecido a cohesión lógica pero los elementos están relacionados en el tiempo y se ejecutan juntos.
Ej.: inicialización, clean-up, finalización.
Mayor cohesión que lógica

Conceptos a nivel de módulo

Hay varios niveles de cohesión:

- Casual
- Lógica
- Temporal
- Procedural
- Comunicacional
- Secuencial
- Funcional

Cohesión

Contiene elementos que pertenecen a una misma unidad procedural.

Ej.: un ciclo o secuencia de decisiones.

Usualmente corta a través de bloques funcionales: tiene un trozo de función o parte de muchas funciones

Tiene elementos que están relacionados por una referencia al mismo dato, i.e. los elementos están juntos porque operan sobre el mismo dato.
Ej.: (1) “print and punch” (2) Pedir los datos de una cuenta personal y devolver todos los datos del registro

Más fuerte

Conceptos a nivel de módulo

Hay varios niveles de cohesión:

- Casual
- Lógica
- Temporal
- Procedural
- Comunicacional
- Secuencial
- Funcional

Cohesión

Los elementos están juntos porque la salida de uno corresponde la entrada del otro.

Puede contener varias funciones o parte de una.

Ej.: Quiero pintar el Fiat 600 de verde: (1) limpiar la chapa, (2) reparar defectos, (3) lijado (4) pintar. => Podría haber una segunda mano.

Es relativamente buena cohesión y relativamente fácil de mantener, pero difícil de reusar

Es la más fuerte de todas las cohesiones: Todos los elementos del módulo están relacionados para llevar a cabo una sola función.

Ej.: (1) calcular el seno de un ángulo; (2) ordenar un arreglo; (3) calcular el salario neto; (4) reservar un asiento en el avión

Más

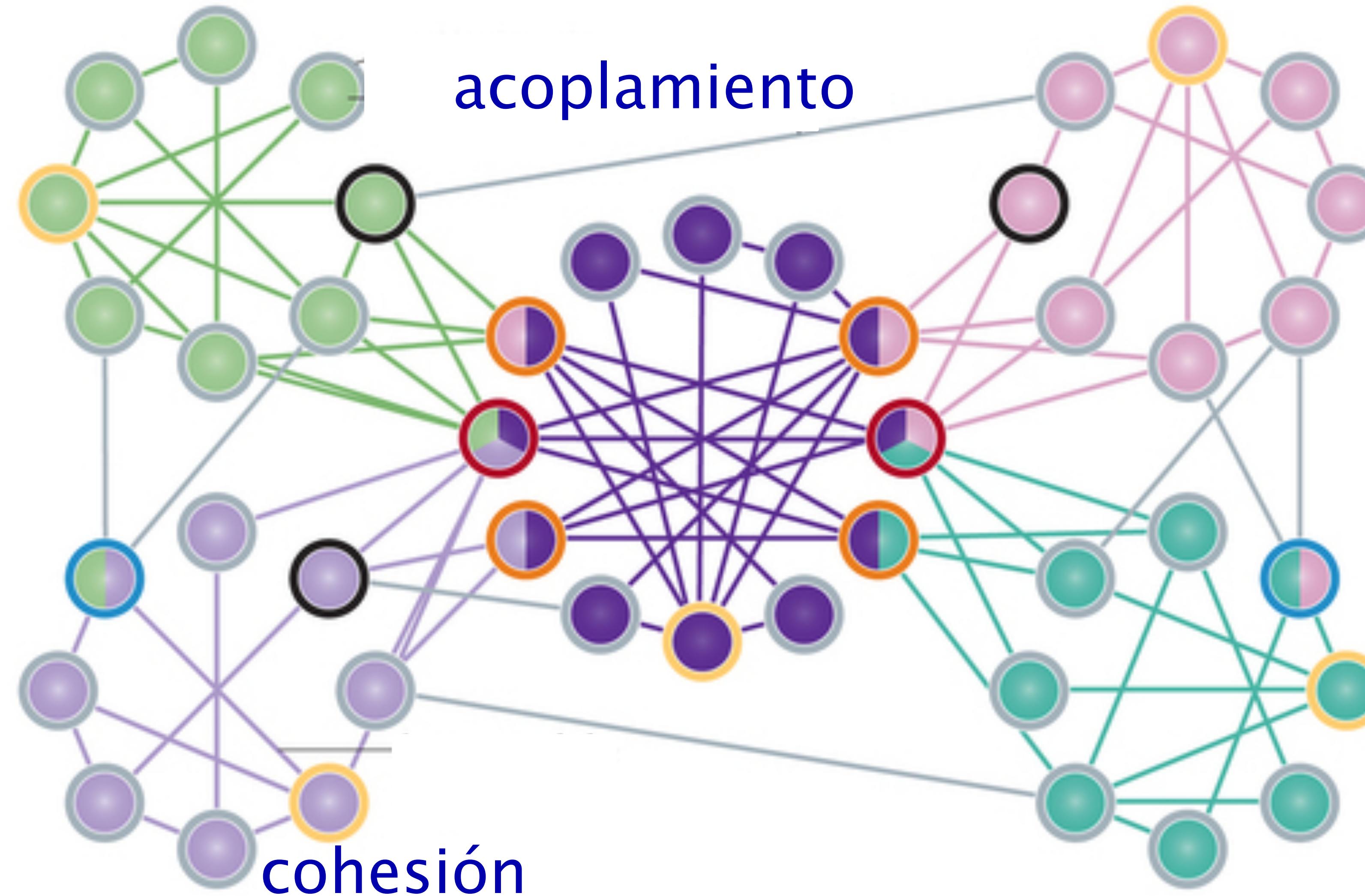
Conceptos a nivel de módulo

Cohesión

¿Cómo determinar la cohesión de un módulo?

- Describir el propósito del módulo con una oración.
- Realizar el siguiente test:
 - Si la oración es compuesta, **tiene comas o más de un verbo =>** el módulo está probablemente realizando más de una función. Probablemente tenga cohesión **secuencial o comunicacional**.
 - Si la oración contiene palabras relacionadas al **tiempo** (ej.: primero, luego, cuando, después) => probablemente el módulo tenga cohesión **secuencial o temporal**.
 - Si el predicado **no contiene un único objeto específico** a continuación del verbo (como es el caso de “editar los datos”) => probablemente tenga cohesión **lógica**.
 - Palabras como **inicializar/limpiar/...** implican cohesión **temporal**.
- Los módulos funcionalmente cohesivos siempre pueden describirse con una **oración simple**.

Conceptos a nivel de módulo



Notación y especificación del diseño

Dos aspectos de interés en la fase de diseño:

- El diseño del sistema (el producto de la fase).
- El proceso que lleva a cabo el diseño.

Una vez satisfecho con el diseño producido, éste debe especificarse en un documento de forma precisa. El documento con la especificación de diseño es usualmente textual, con notaciones como auxilio para la comprensión

Para esto necesitamos principios y métodos. Además se requieren notaciones para registrar ideas y decisiones, y para desarrollar borradores a los cuales evaluar. Usualmente, estas notaciones son gráficas

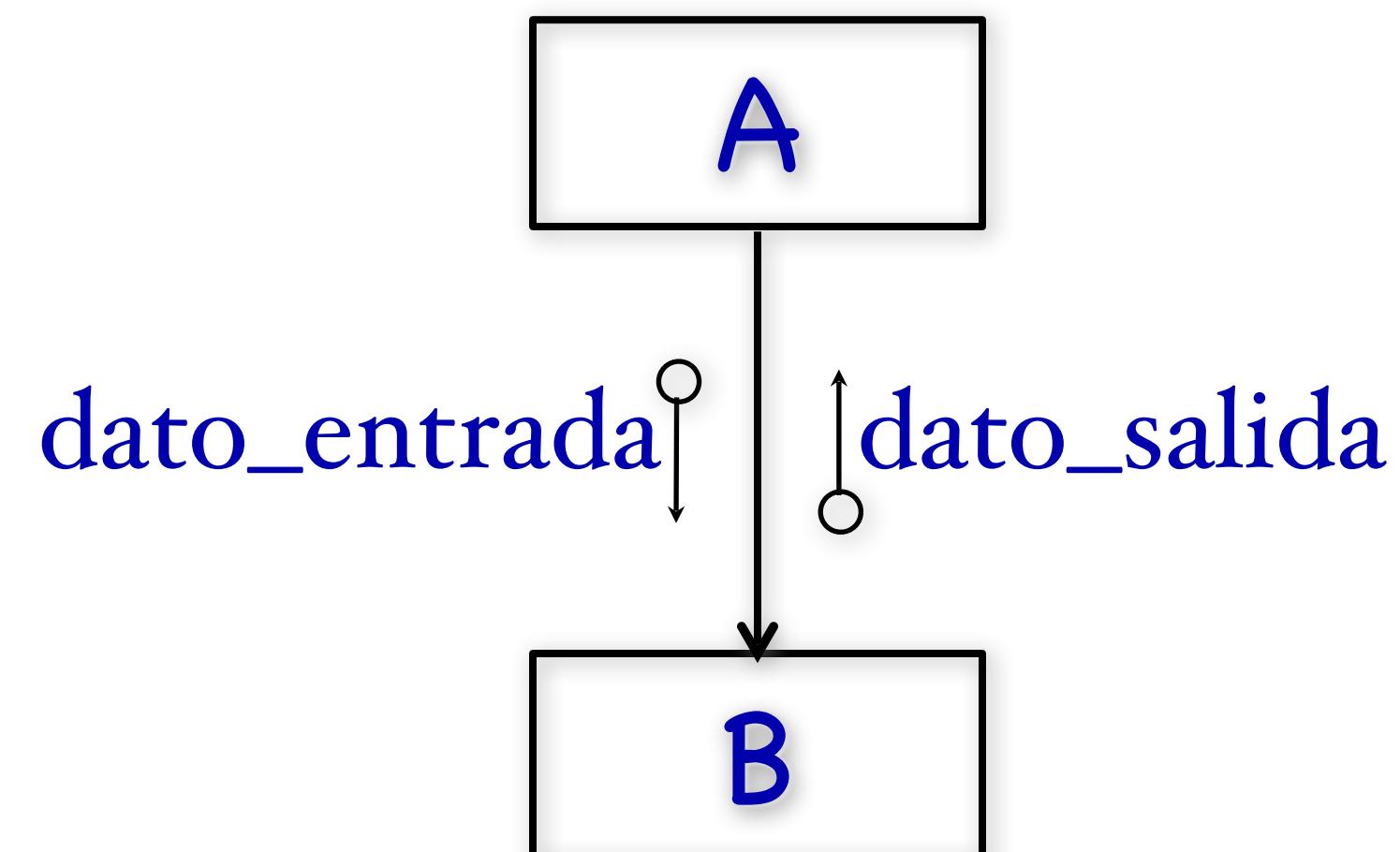
Notación y especificación del diseño

Diagrama de estructura

Todo programa tiene estructura.

- Diagrama de estructura:

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



Notación y especificación del diseño

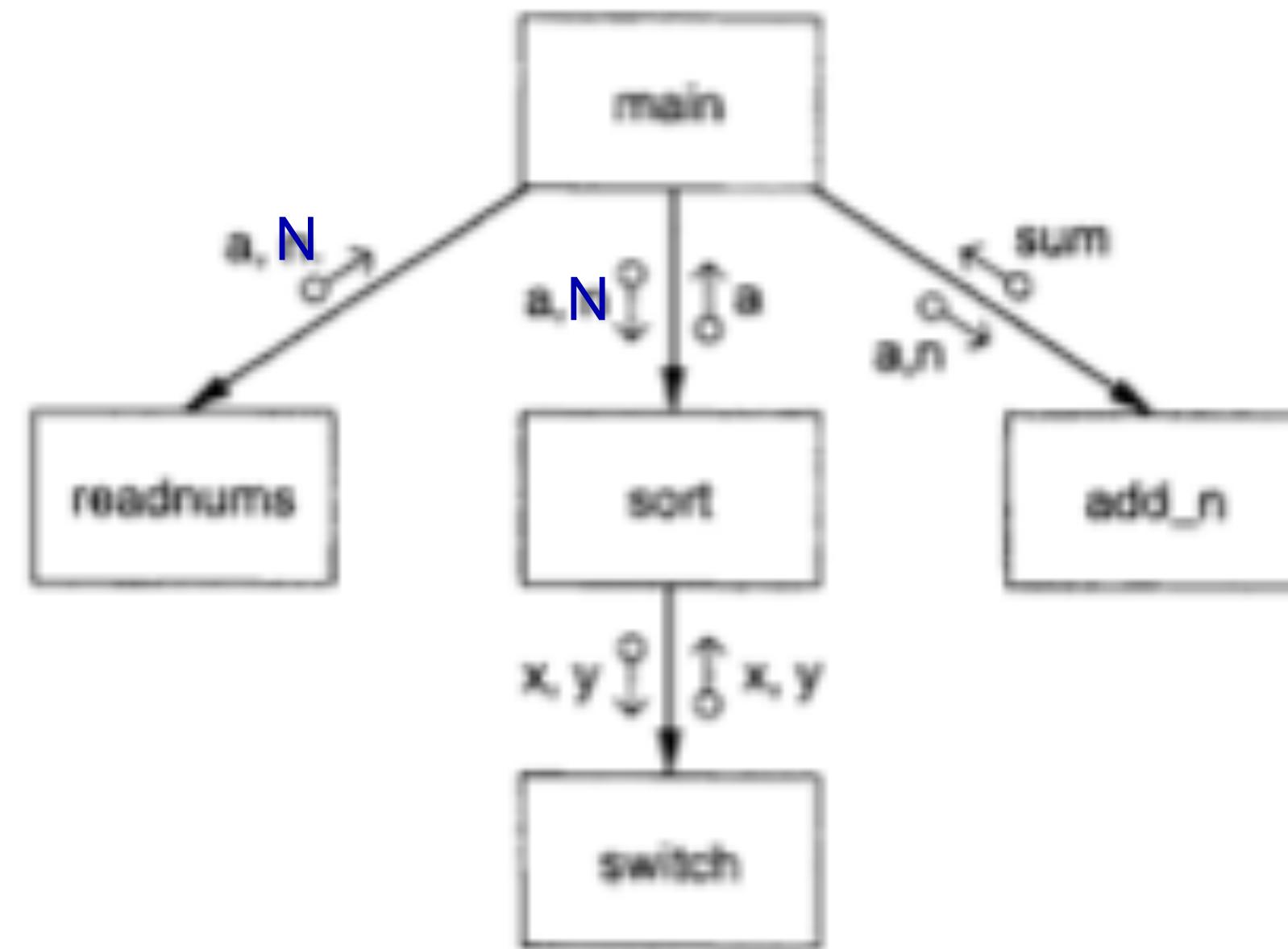
Diagrama de estructura

```
main()
{
    int sum, n, N, a[MAX];
    readnums(a, &N); sort(a, N); scanf(&n);
    sum = add_n(a, n); printf(sum);
}

readnums(a, N)

int a[], *N;
{
    :
}

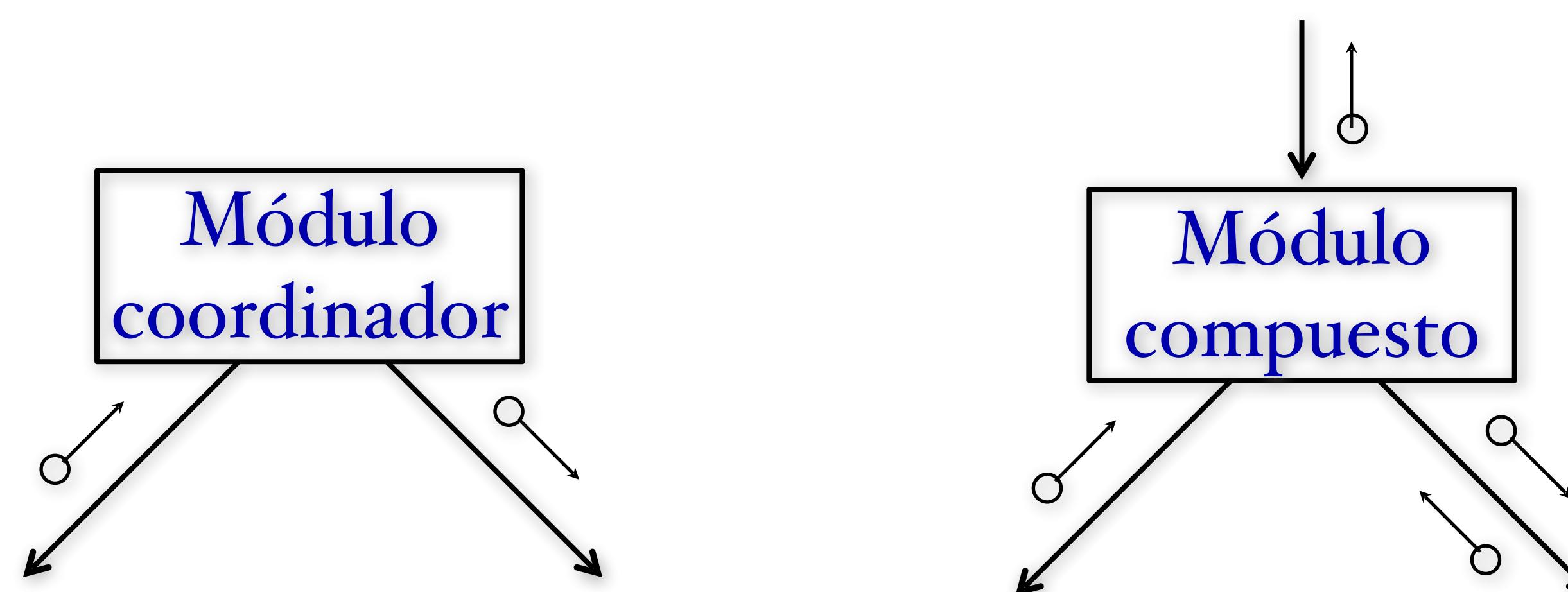
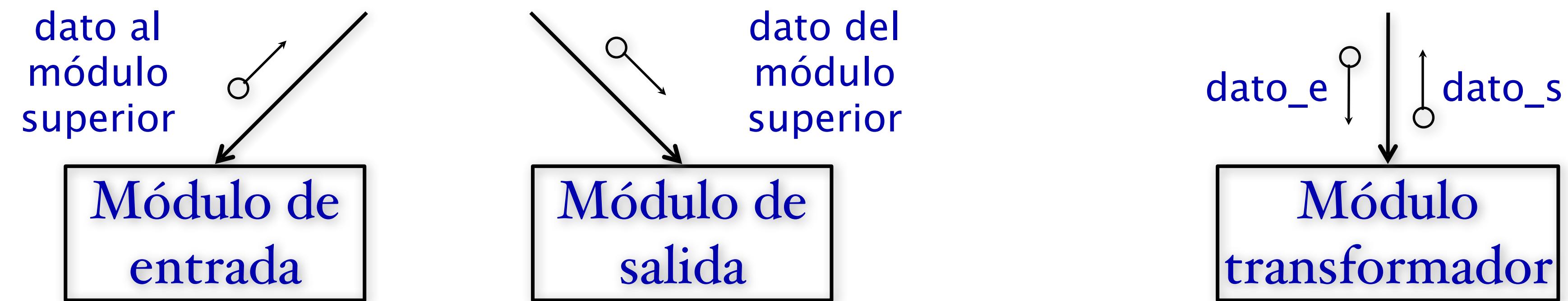
sort(a, N)
int a[], N;
{
    :
    if (a[i] > a[t]) switch(a[i], a[t]);
    :
}
/* Add the first n numbers of a */
add_n(a, n)
int a[], n;
{
    :
}
```



Notación y especificación del diseño

Diagrama de estructura

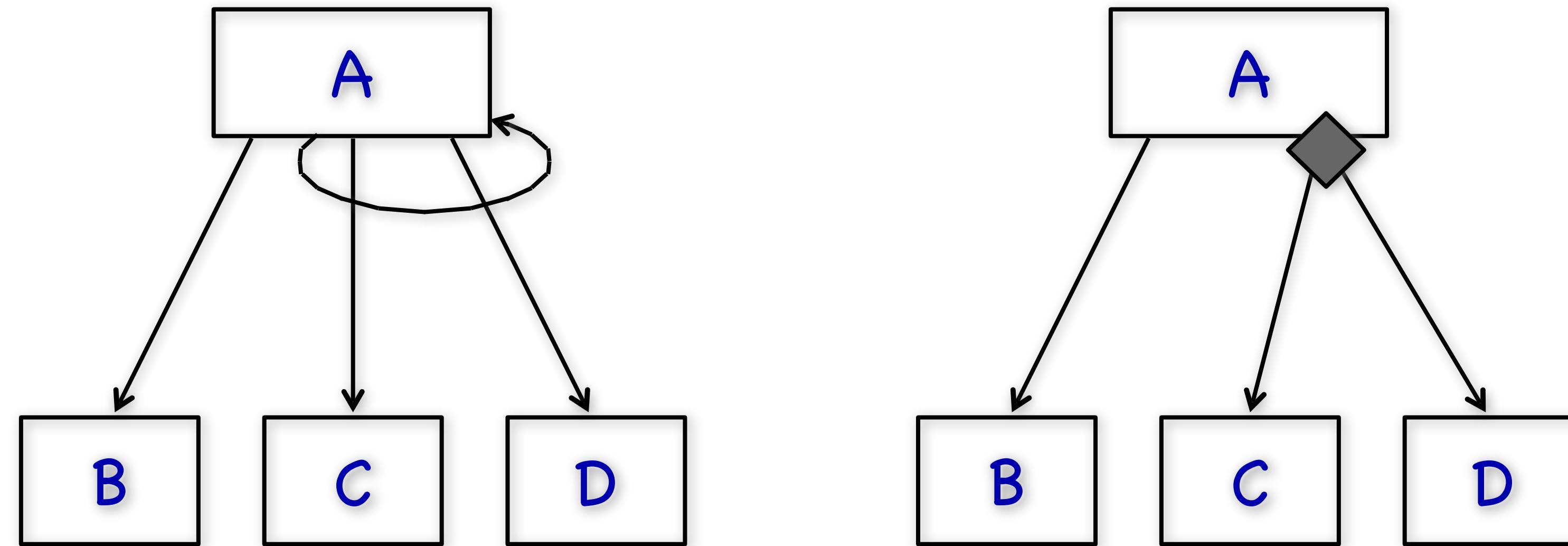
Tipos de módulos:



Notación y especificación del diseño

Diagrama de estructura

Iteración y decisión:



- Las mayores iteraciones y decisiones pueden indicarse.
- No es intención de los diagramas de estructura mostrar la lógica del programa, sólo la estructura.

Metodología de diseño estructurado

- La estructura se decide durante el diseño.
- La implementación **no** debe cambiar la estructura.
- 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.

Metodología de diseño estructurado

SDM ve al software como una función de transformación que convierte una entrada dada en la salida esperada.

- El foco en el diseño estructurado es la función de transformación.
=> SDM es una metodología orientada a funciones.
- Utiliza abstracción funcional y descomposición funcional.
- Objetivo de la SDM:
 - Especificar módulos de funciones y sus conexiones siguiendo una estructura jerárquica con bajo acoplamiento y alta cohesión.

Metodología de diseño estructurado

- Los módulos con módulos subordinados no realizan mucha computación.
- La mayoría de la computación se realiza en los módulos subordinados.
- El módulo principal se encarga de la coordinación.
- Así sucesivamente hasta los módulos “atómicos”.
- La factorización es el proceso de descomponer un módulo de manera que el grueso de la computación se realice en los módulos subordinados.
- Sistema completamente factorizado: el procesamiento real se realiza en los módulos atómicos del nivel más bajo.
- SDM apunta a acercarse a la factorización completa.

Metodología de diseño estructurado

Pasos principales de esta metodología:

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).

Metodología de diseño estructurado

Paso 1: Reformular el problema con DFDs

- El diseño estructurado comienza con un DFD que capture el flujo de datos del sistema propuesto.
- DFD es una representación importante: provee una visión de alto nivel del sistema.
- Enfatiza el flujo de datos a través del sistema.
- Ignora aspectos procedurales.
Aunque la notación es la misma, el propósito aquí es diferente de DFD en análisis de requerimientos.

Metodología de diseño estructurado

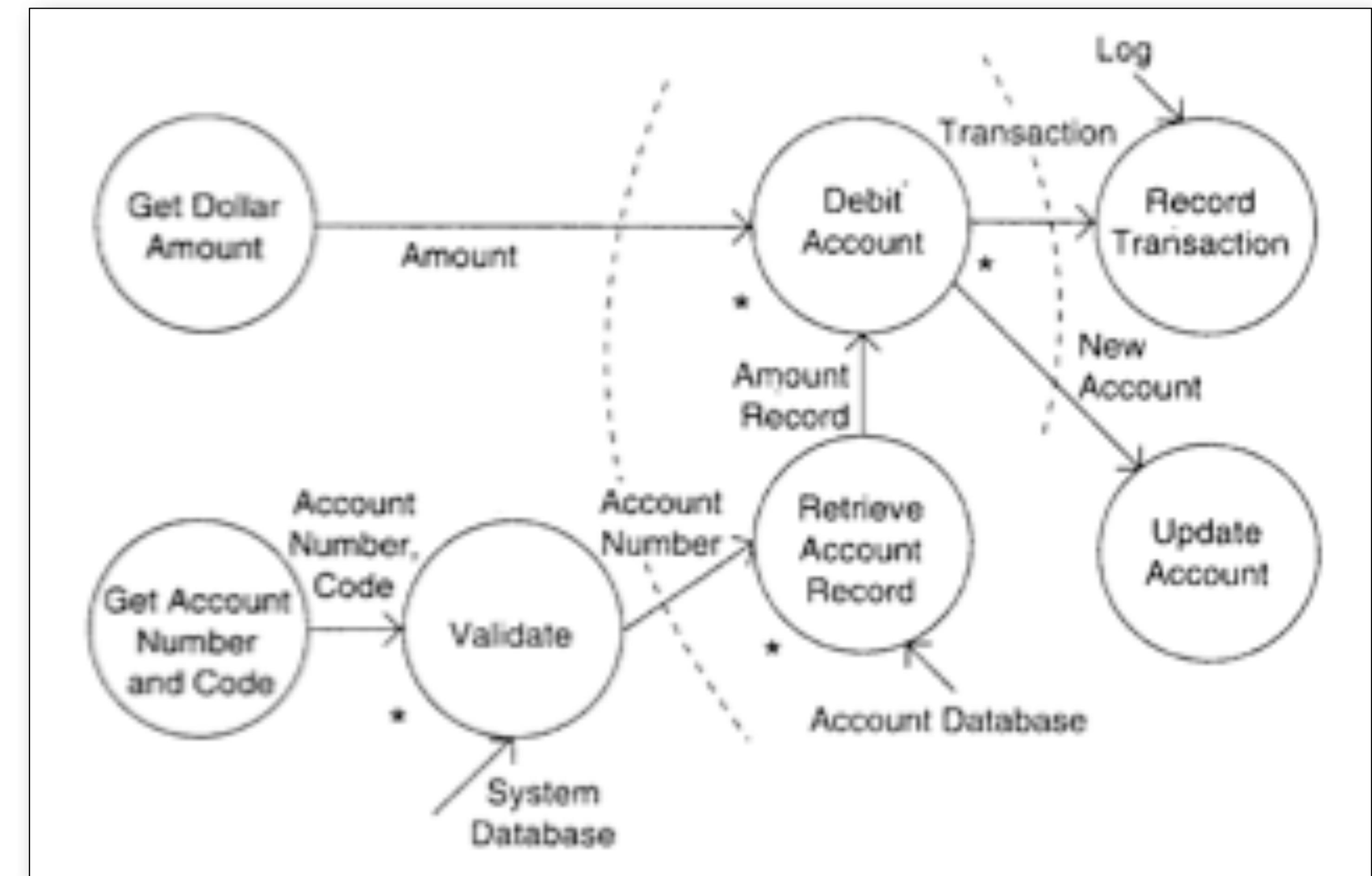
Paso 1: Reformular el problema con DFDs

- Identificar las entradas, salidas, fuentes, sumideros del sistema.
- Trabajar consistentemente desde la entrada hacia la salida o al revés.
 - Si se complica => cambiar el sentido.
 - Identificar los transformadores que convierten las entradas en salidas.
- No mostrar nunca lógica de control; si se comienza a pensar en término de loops/ condiciones: parar y recomenzar.
- Etiquetar cada flecha y burbuja. Identificar cuidadosamente las entradas y salidas de cada transformador.
- Hacer uso de + y *.
- Ignorar las funciones menores al principio.
- En sistemas complejos realizar DFD de manera jerárquica.
- Dibujar DFDs alternativos antes de definirse por uno.

Metodología de diseño estructurado

Paso 1: Reformular el problema con DFDs

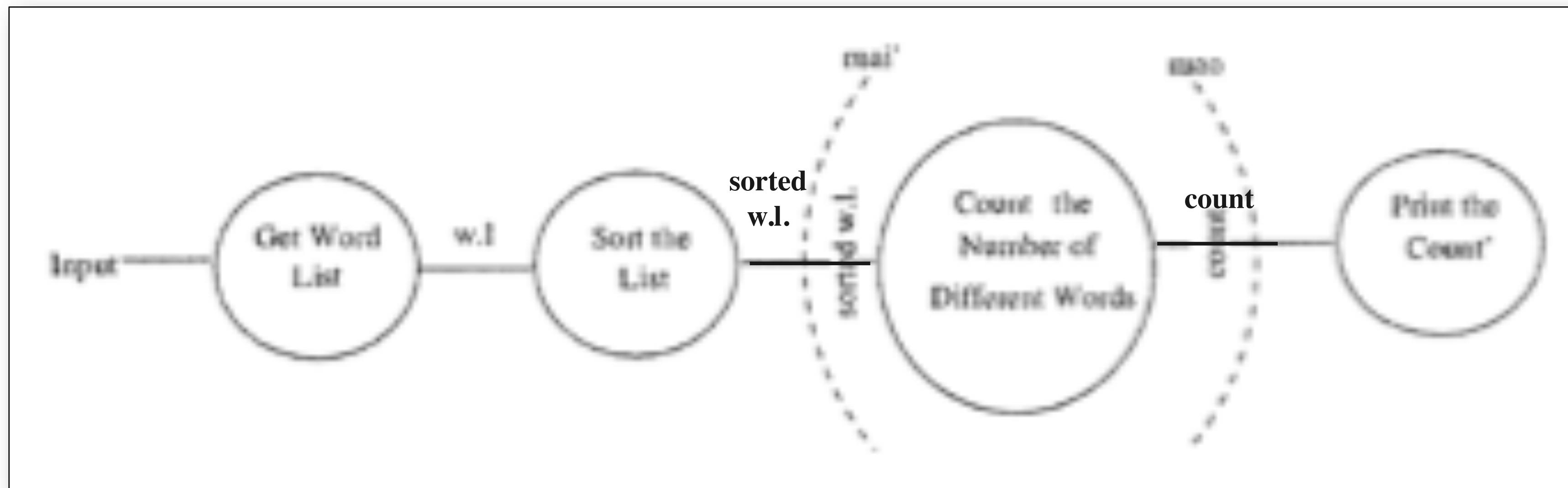
Ej.: Retiro de dinero en un cajero automático



Metodología de diseño estructurado

Paso 1: Reformular el problema con DFDs

Ej.: Conteo de las distintas palabras en un archivo



Metodología de diseño estructurado

Paso 2: Identificar las entradas/salidas más abstractas

- Generalmente los sistemas realizan una función básica.
- Pero usualmente no se realiza sobre la entrada directamente.
- Primero la entrada debe convertirse en un formato adecuado.
- Similarmente las salidas producidas por los transformadores principales deben transformarse a salidas físicas adecuadas.
- Se requieren varios transformadores para procesar las entradas y las salidas.

Objetivo de este paso: separar tales transformadores de los que realizan las transformaciones reales.

Metodología de diseño estructurado

Paso 2: Identificar las entradas/salidas más abstractas

- Entradas más abstractas (MAI): elementos de datos en el DFD que están más distantes de la entrada real, pero que aún puede considerarse como entrada.
- Podrían tener poca semejanza con la entrada real.
- En general son ítems de datos obtenidos luego de chequeo de errores, formateo, validación de datos, conversión, etcétera.
- Para encontrarla:
 - Ir desde la entrada física en dirección de la salida hasta que los datos no puedan considerarse entrantes.
 - Ir lo más lejos posible sin perder la naturaleza entrante.
- Es dual para obtener las salidas más abstractas (MAO).

Metodología de diseño estructurado

Paso 2: Identificar las entradas/salidas más abstractas

- Representa un juicio de valor, aunque usualmente la elección es bastante obvia.
- Las burbujas entre la MAI y la MAO corresponden a los transformadores centrales.
- Son los que realizan la transformación básica.
- Con las MAI y MAO, los transformadores centrales se concentran en la transformación sin importar el formato/validación/... de las entradas y salidas.

Metodología de diseño estructurado

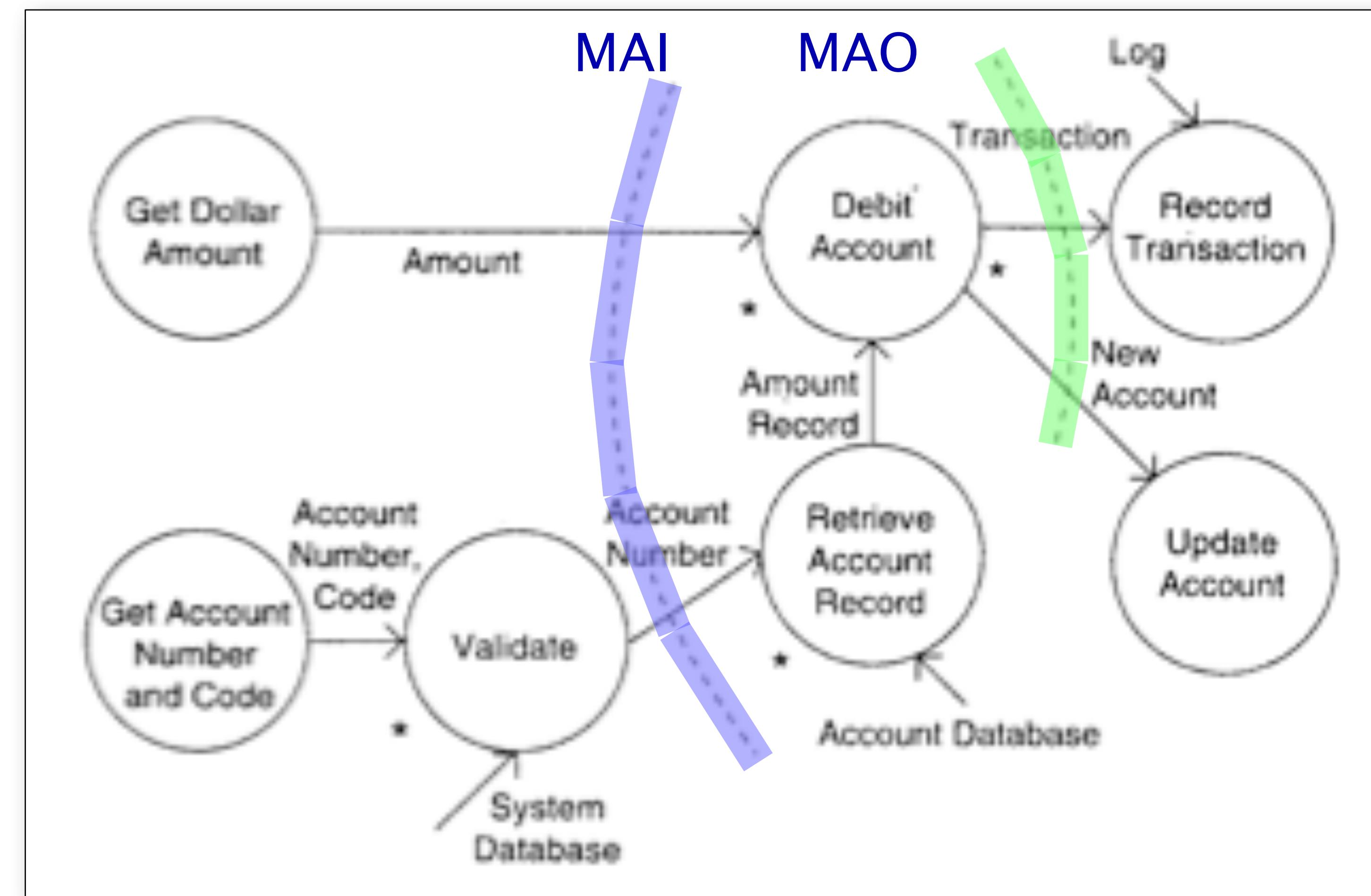
Paso 2: Identificar las entradas/salidas más abstractas

- Visión del problema: cada sistema hace alguna E/S y algún procesamiento.
- En muchos sistemas el procesamiento de E/S forma una gran parte del código.
- Este enfoque separa las distintas funciones:
 - Subsistema que realiza principalmente la entrada.
 - Subsistema que realiza principalmente las transformaciones.
 - Subsistema que realiza principalmente la presentación de la salida.

Metodología de diseño estructurado

Paso 2: Identificar las entradas/salidas más abstractas

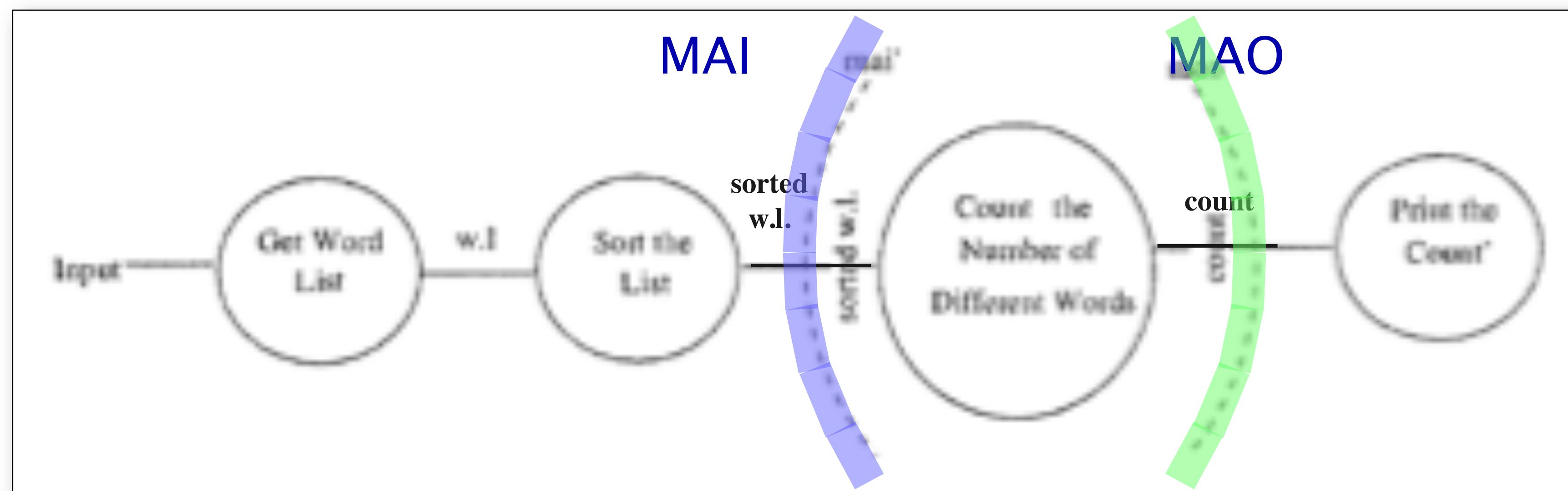
Ej.: Retiro de dinero en un cajero automático



Metodología de diseño estructurado

Paso 2: Identificar las entradas/salidas más abstractas

Ej.: Conteo de las distintas palabras en un archivo



Metodología de diseño estructurado

Paso 3: Realizar el primer nivel de factorización

Primer paso para obtener el diagrama de estructura.

- Especificar el módulo principal.
- Especificar un módulo de entrada subordinado por cada ítem de dato de la MAI.
- El propósito de estos módulos de entrada es enviar al módulo principal los ítems de datos de la MAI.
- Especificar un módulo de salida subordinado por cada ítem de dato de la MAO.
- Especificar un módulo transformador subordinado por cada transformador central.
- Las entradas y salidas de estos módulos transformadores están especificadas en el DFD.

Metodología de diseño estructurado

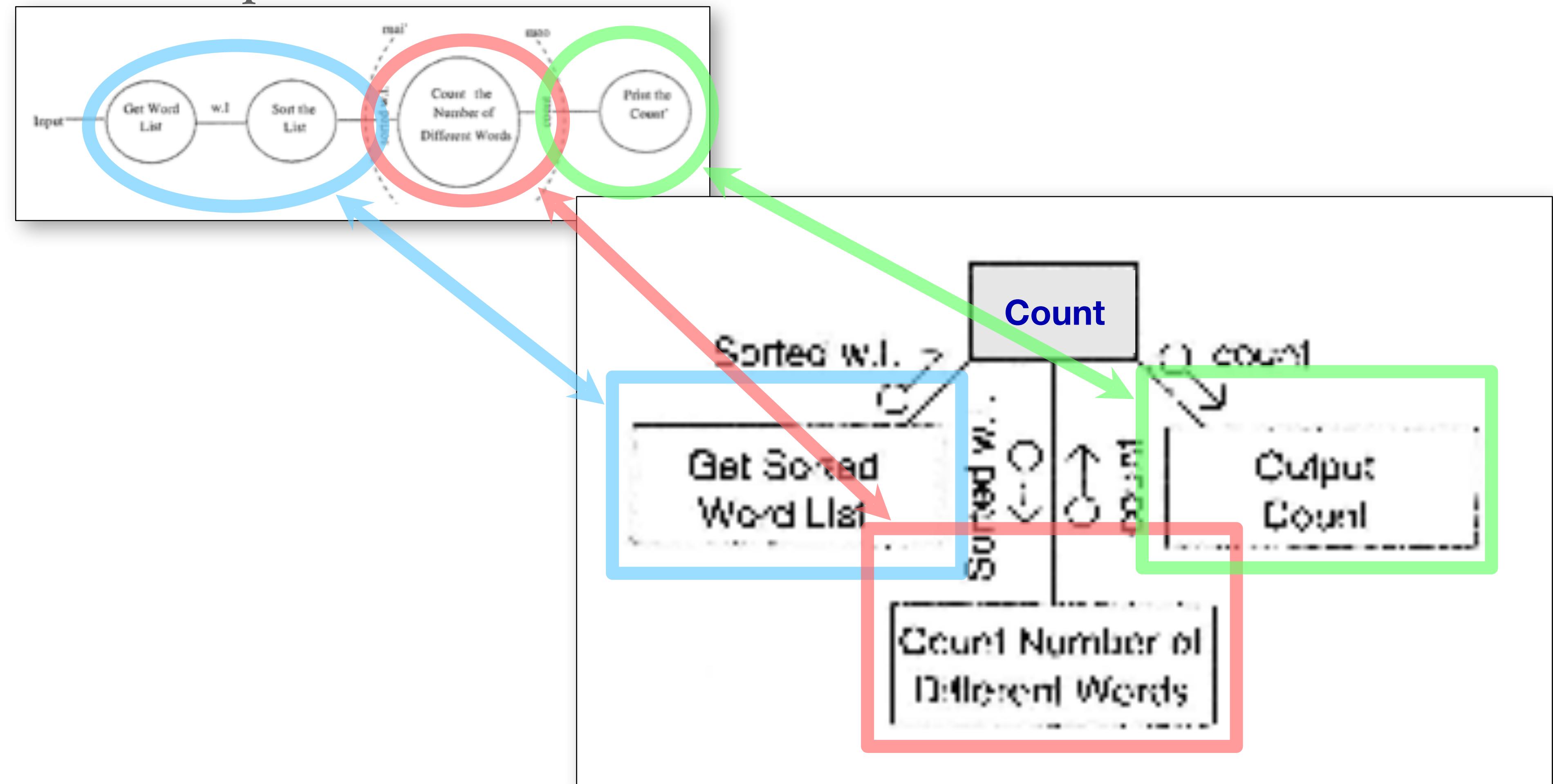
Paso 3: Realizar el primer nivel de factorización

- El primer nivel de factorización es sencillo.
- El módulo principal es un módulo coordinador.
- Algunos módulos subordinados son responsables de entregar las entradas lógicas.
- Estas se pasan a los módulos transformadores para obtener las salidas lógicas.
- Estas salidas lógicas son consumidas por los módulos de salida.
- Divide el problema en tres problemas separados.
- Cada uno de los tres tipos distintos de módulos pueden diseñarse separadamente.
- Estos módulos son independientes.

Metodología de diseño estructurado

Paso 3: Realizar el primer nivel de factorización

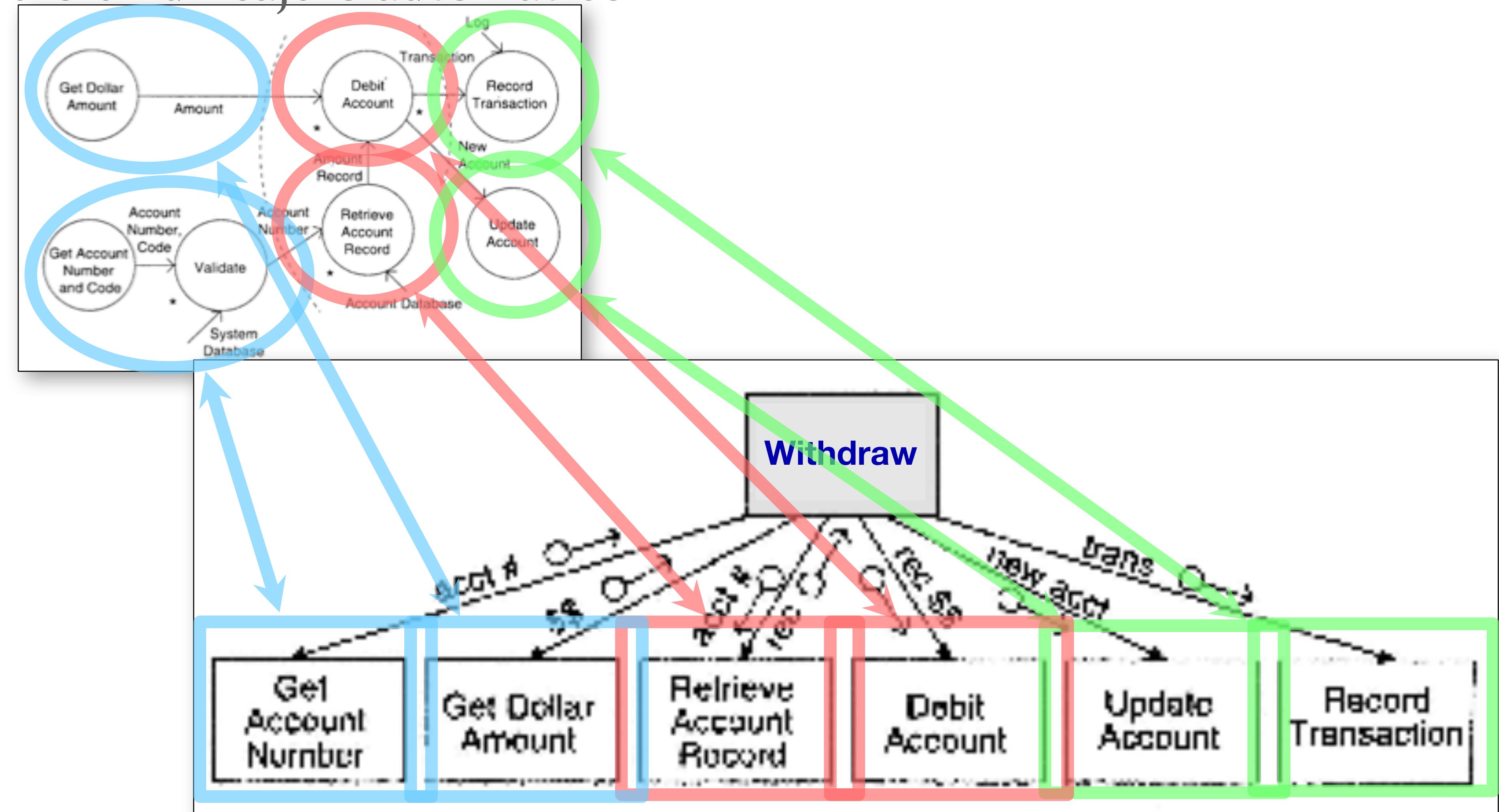
Ej.: Conteo de las distintas palabras en un archivo



Metodología de diseño estructurado

Paso 3: Realizar el primer nivel de factorización

Ej.: Retiro de dinero en un cajero automático



Metodología de diseño estructurado

Paso 4.1: Factorizar los módulos de entrada

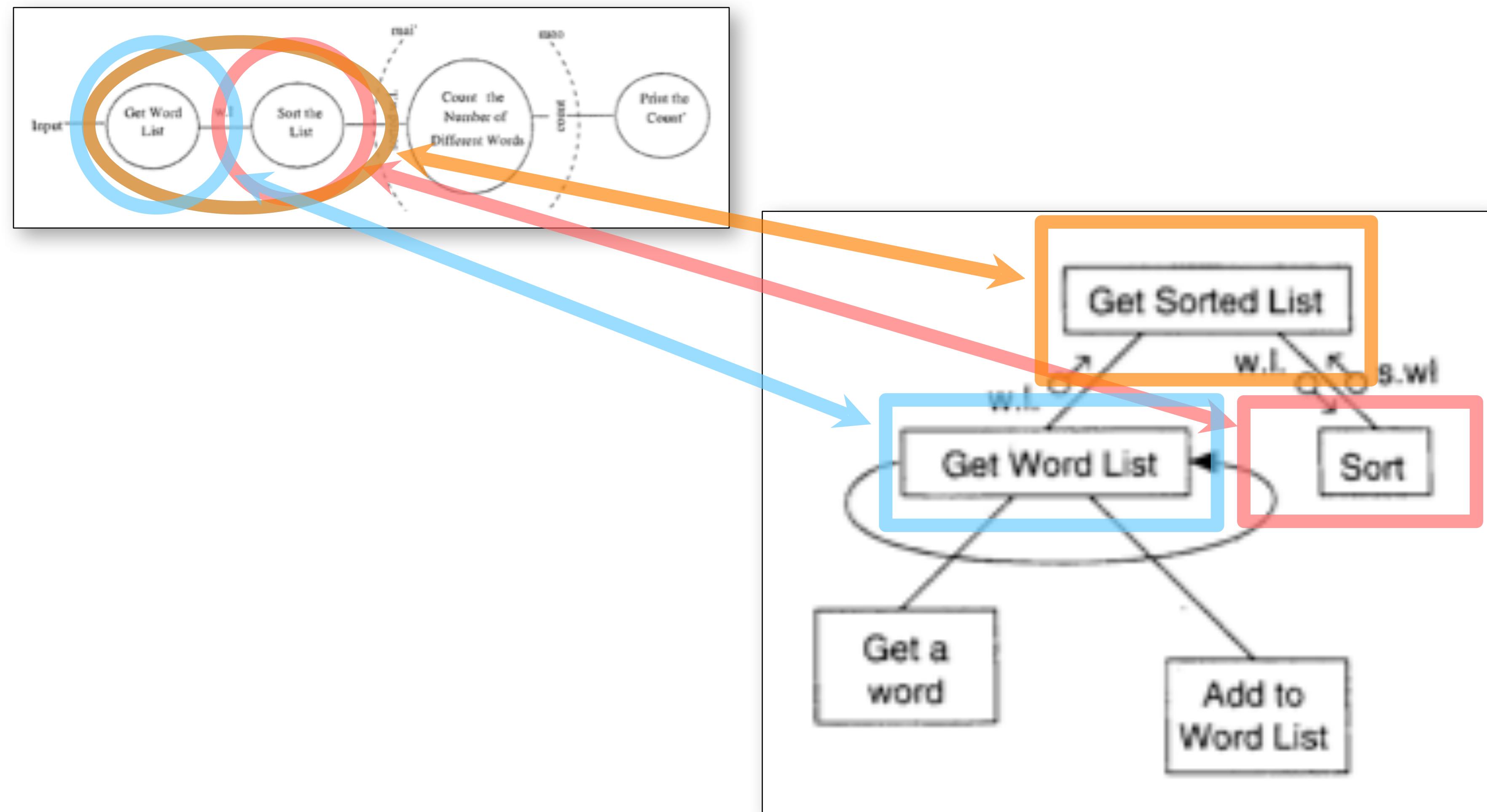
El transformador que produce el dato de MAI se trata ahora como un transformador central.

- Se repite el proceso del primer nivel de factorización considerando al módulo de entrada como si fuera el módulo principal.
- Se crea un módulo subordinado por cada ítem de dato que llega a este nuevo transformador central.
- Se crea un módulo subordinado para el nuevo transformador central.
- Usualmente no debería haber módulos de salida.
- Los nuevos módulos de entrada se factorizan de la misma manera hasta llegar a la entrada física.

Metodología de diseño estructurado

Paso 4.1: Factorizar los módulos de entrada

Ej.: Conteo de las distintas palabras en un archivo



Metodología de diseño estructurado

Paso 4.2: Factorizar los módulos de salida

La factorización de la salida es simétrica.

- Módulos subordinados: un transformador y módulos de salida.
- Usualmente no debería haber módulos de entrada.

Metodología de diseño estructurado

Paso 4.3: Factorizar los transformadores centrales

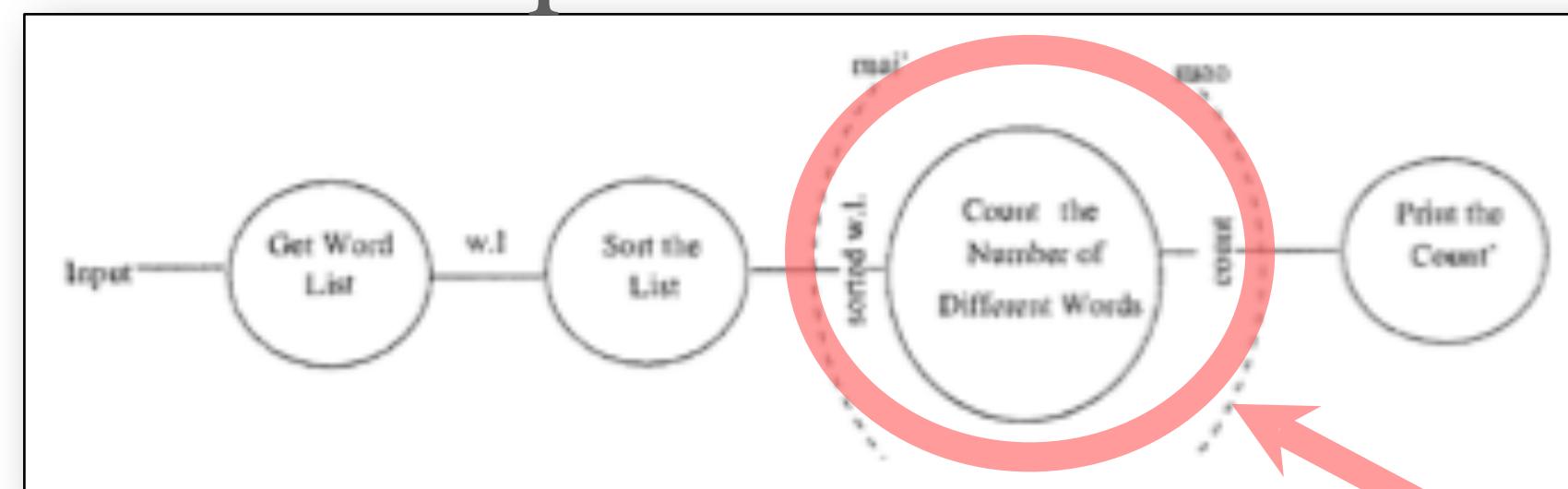
La factorización de los módulos de entrada y salida es simple si el DFD es detallado.

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

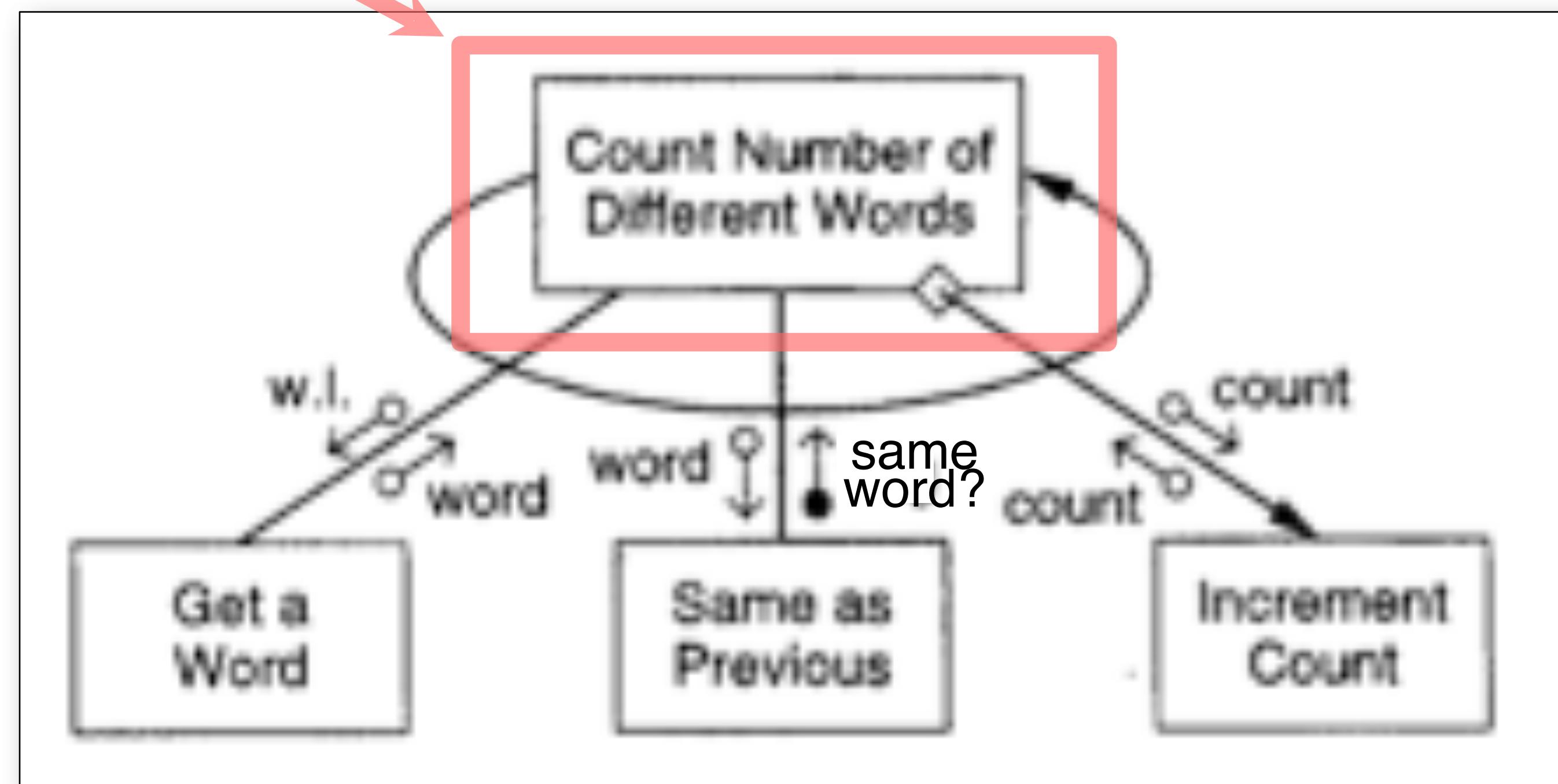
Metodología de diseño estructurado

Paso 4.3: Factorizar los transformadores centrales

Ej.: Conteo de las distintas palabras en un archivo



En este caso, el diagrama de estructura es lo suficientemente simple como para no realizar DFD



Metodología de diseño estructurado

Heurísticas de diseño

Los pasos anteriores no deberían seguirse ciegamente.

- La estructura obtenida podría modificarse si fuera necesario (si uno lo considera así).
- El objetivo, siempre, es lograr bajo acoplamiento y alta cohesión.
 - Utilizar heurísticas de diseño para modificar el diseño inicial.
- Heurística de diseño: un conjunto de “rules of thumb” que generalmente son útiles.

Metodología de diseño estructurado

Heurísticas de diseño

- Tamaño del módulo: Indicador de la complejidad del módulo.
 - Examinar cuidadosamente los módulos con muy poquitas líneas o con mas de 100 líneas.
- Cantidad de flechas de salida y cantidad de flechas de llegada:
 - La primera no debería exceder las 5 o 6 flechas; la segunda debería maximizarse.
- Alcance del efecto de un módulo: los módulos afectados por una decisión en este módulo.
- Alcance del control de un módulo: Todos los subordinados.
- “Rule of thumb”: Por cada módulo, el alcance del efecto debe ser un subconjunto del de control.
- Idealmente una decisión solo debe tener efecto en los inmediatos subordinados.

Verificación del diseño

Objetivo principal: Asegurar que el diseño implemente los requerimientos (corrección).

- También debe realizarse análisis de desempeño, eficiencia, etcétera.
- Si se usan lenguaje formales para representar el diseño => existen herramientas que asisten a estos efectos.
- De todas maneras, la revisión del diseño es la forma más común de realizar la verificación.
- La calidad del diseño se completa con una buena modularidad, bajo acoplamiento y alta cohesión.

Las listas de control
siempre son útiles

Verificación del diseño

Ejemplo de una lista de control

- ¿Se tuvieron en cuenta todos los requerimientos funcionales?
- ¿Se realizaron análisis para demostrar que los requerimientos de desempeño se satisfacen?
- ¿Se establecieron explícitamente todas las suposiciones? ¿son aceptables?
- ¿Existen limitaciones/restricciones en el diseño aparte de las establecidas en los requerimientos?
- ¿Se especificaron completamente todas las especificaciones externas de los módulos?
- ¿Se consideraron las condiciones excepcionales?
- ¿Son todos los formatos de datos consistentes con los requerimientos?
- ¿Se trataron apropiadamente todas las operaciones e interfaces de usuario?
- ¿El diseño es modular? ¿conforma los estándares locales?
- ¿Se estimaron los tamaños de las estructuras de datos? ¿Se consideraron los casos de overflow?

Métricas

Propósito básico: proveer una evaluación cuantitativa del diseño, así el producto final puede mejorarse.

- Tamaño
- Complejidad
- De red
- De estabilidad
- Flujo de la información

Métricas

Tamaño y Complejidad

- El tamaño es siempre una métrica: luego del diseño se puede estimar mejor.
Ej.: Cantidad de módulos + tamaño estimado de c/u.
- La complejidad es otra métrica de interés.

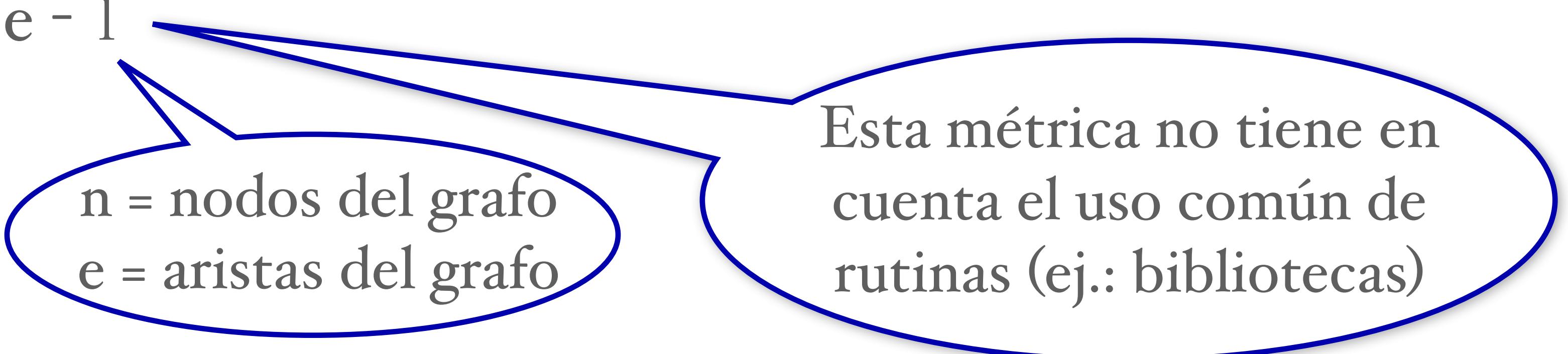
Métricas

Métricas de red

Se enfoca en la estructura del diagrama de estructuras; se considera un buen diagrama a aquel en el cual cada módulo tiene sólo un módulo invocador (reduce acoplamiento).

- Cuanto más se desvíe de esta forma de árbol, más impuro es el diagrama:

- Impureza del grafo = $n - e - 1$
- Impureza = 0 => árbol



- A medida que este valor se hace más negativo, se incrementa la impureza.

Métricas

Métricas de estabilidad

La estabilidad trata de capturar el impacto de los cambios en el diseño.

- Cuanto mayor estabilidad, mejor.
- Estabilidad de un módulo: la cantidad de suposiciones por otros módulos sobre éste.
 - Depende de la interfaz del módulo y del uso de datos globales.
 - Se conocen luego del diseño.

Métricas

Métricas de flujo de información

Métricas de red: parte de la hipótesis que si la impuridad del grafo se incrementa
=> el acoplamiento se incrementa.

- Pero: el acoplamiento también se incrementa con la complejidad de la interfaz.
- Las métricas de flujo de información tienen en cuenta:
 - la complejidad intra-módulo, que se estima con el tamaño del módulo en LOC.
 - la complejidad inter-módulo que se estima con
 - inflow: flujo de información entrante al módulo.
 - outflow: flujo de información saliente del módulo.
- La complejidad del diseño del módulo C se define como
 - $DC = \text{tamaño} * (\text{inflow} * \text{outflow})^2$

Métricas

Métricas de flujo de información

Métricas de red: parte de la hipótesis que si la impuridad del grafo se incrementa
=> el acoplamiento se incrementa.

- Pero: el acoplamiento también se incrementa con la complejidad.
- Las métricas de flujo de información tienen en cuenta:
 - la complejidad intra-módulo, que se estima con el número de LOC.
 - la complejidad inter-módulo que se estima con:
 - inflow: flujo de información entrante al módulo.
 - outflow: flujo de información saliente del módulo.
- La complejidad del diseño del módulo C se define como:
 - $DC = \text{tamaño} * (\text{inflow} * \text{outflow})^2$

Su cuadrado representa la importancia de la interconexión entre módulos con respecto a la complejidad interna, i.e. el tamaño

$(\text{inflow} * \text{outflow})$ representa el total de combinaciones de entradas y salidas

Métricas

Métricas de flujo de información

La métrica anterior define la complejidad sólo en la cantidad de información que fluye hacia adentro y hacia fuera y el tamaño del módulo.

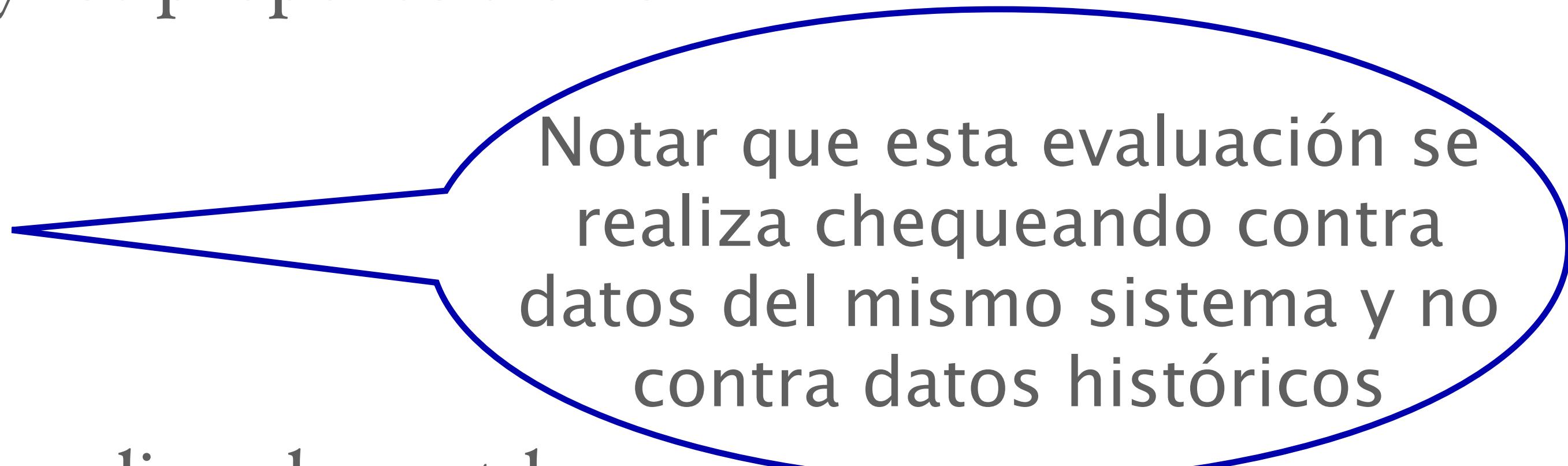
- Ya vimos en la métrica de red que también es importante la cantidad de módulos desde y hacia donde fluye la info.
- En base a esto, el impacto del tamaño del módulo empieza a resultar considerarse insignificante.
- En base a esto, la complejidad del diseño del módulo C se puede definir como:
$$DC = fan_in * fan_out + inflow * outflow$$
donde fan_in representa la cantidad de módulos que llaman al módulo C, y fan_out los llamados por C.

Métricas

Métricas de flujo de información

Ahora: ¿cómo utilizamos esta métrica?

- Se usa el promedio de la complejidad de los módulos y su desviación estándar para identificar los módulos complejos y los propenso a error
- Propenso a error si
 $DC > complej_media + desv_std$
- Complejo si
 $complej_media < DC < complej_media + desv_std$
- Normal en caso contrario.



Notar que esta evaluación se realiza chequeando contra datos del mismo sistema y no contra datos históricos

Ingeniería del Software I

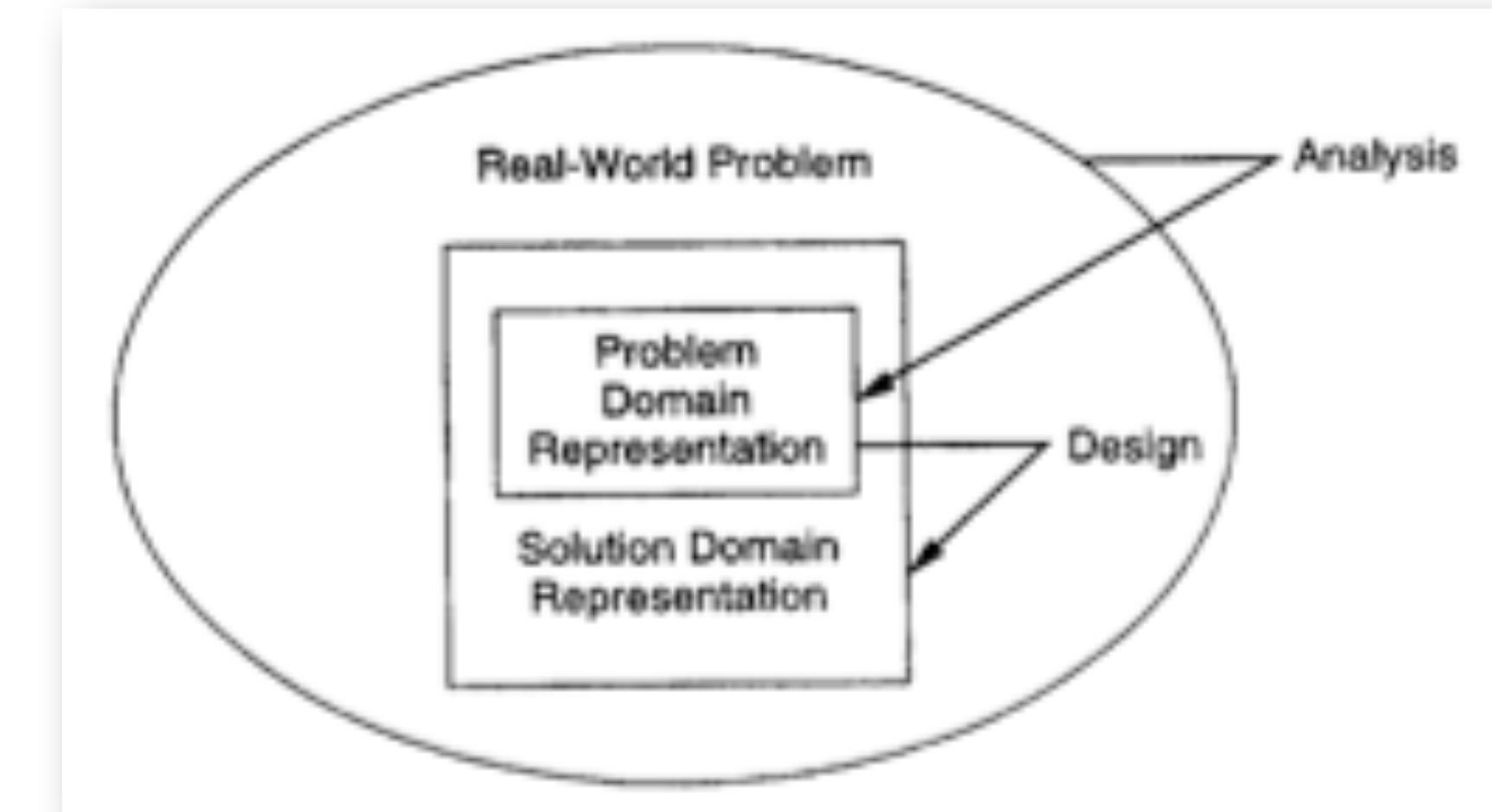
4 – Diseño orientado a objetos

Orientación 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.
- Pero: 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.



Análisis OO y diseño OO

- 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 del AOO representan un concepto del problema => objetos semánticos.
 - Además de los objetos semánticos, el DOO producen otros tipos de objetos:
 - objetos de interfaces,
 - objetos de aplicaciones, y
 - objetos de utilidad.
 - Además, el DOO hace mayor incapié en el comportamiento **dinámico** del sistema.

Análisis OO y diseño OO

- 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 del AOO representan el problema => objetos semánticos.
 - Además de los objetos semánticos, el DOO incluye otros tipos de objetos:
 - objetos de interfaces,
 - objetos de aplicaciones, y
 - objetos de utilidad.
 - Además, el DOO hace mayor incapié en el comportamiento de los objetos.
-
- Se encargan de la interfaz con el usuario
- Especifican los mecanismos de control para la solución propuesta
- Son los necesarios para soportar los servicios de los objetos semánticos (ej.: pilas, árboles, diccionarios, etc.)

Conceptos de la orientación a objetos

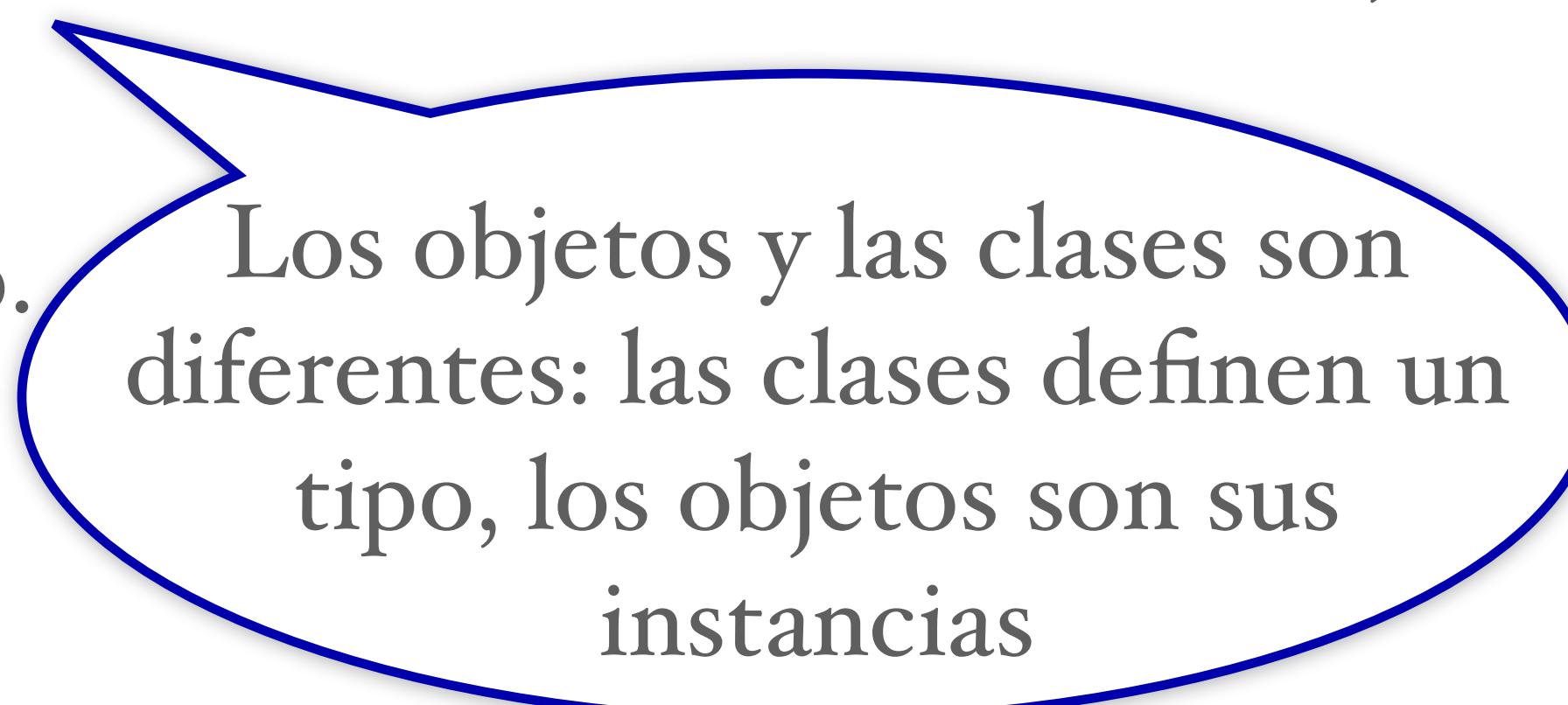
Clases y objetos

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

Conceptos de la orientación a objetos

Clases y objetos

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



Los objetos y las clases son diferentes: las clases definen un tipo, los objetos son sus instancias

Conceptos de la orientación a objetos

Relaciones entre objetos

- Si un objeto está vinculado durante un tiempo con otro hay una **asociación** entre ellos.
- Un objeto interactúa con otro enviándole un mensaje solicitando un servicio. Tras la recepción del mensaje, el objeto receptor invoca al método que implementa tal servicio.
- Si un objeto es parte de otra clase hay una **agregación**, relación derivada de una asociación. En general es una colección. Sus ciclos de vida no están relacionados. Es una relación unidireccional.
- Si un objeto está compuesto por otros se dice que hay una **composición**. El ciclo de vida de ambos objetos están muy relacionados.

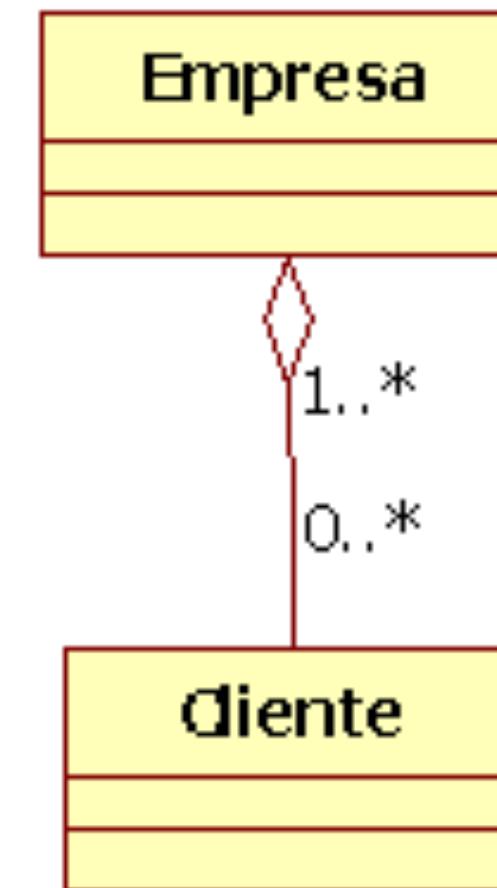
Conceptos de la orientación a objetos

Relaciones entre objetos

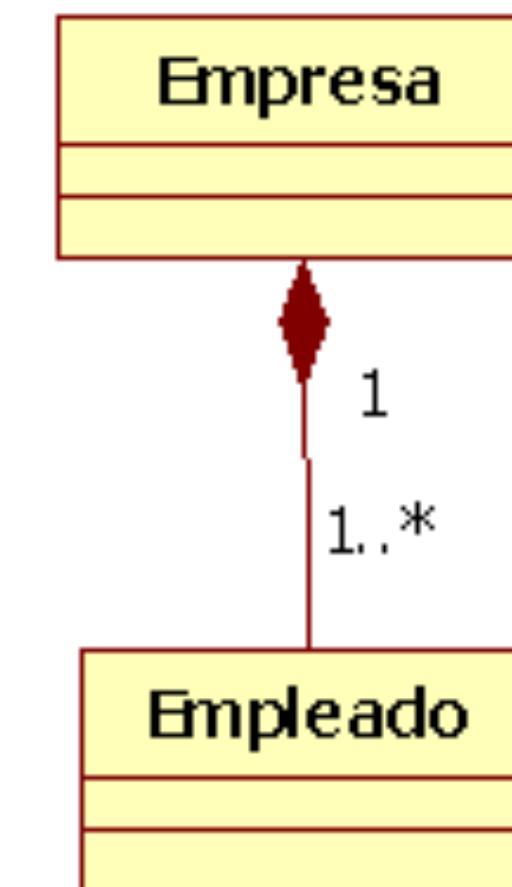
Asociación:



Agregación:



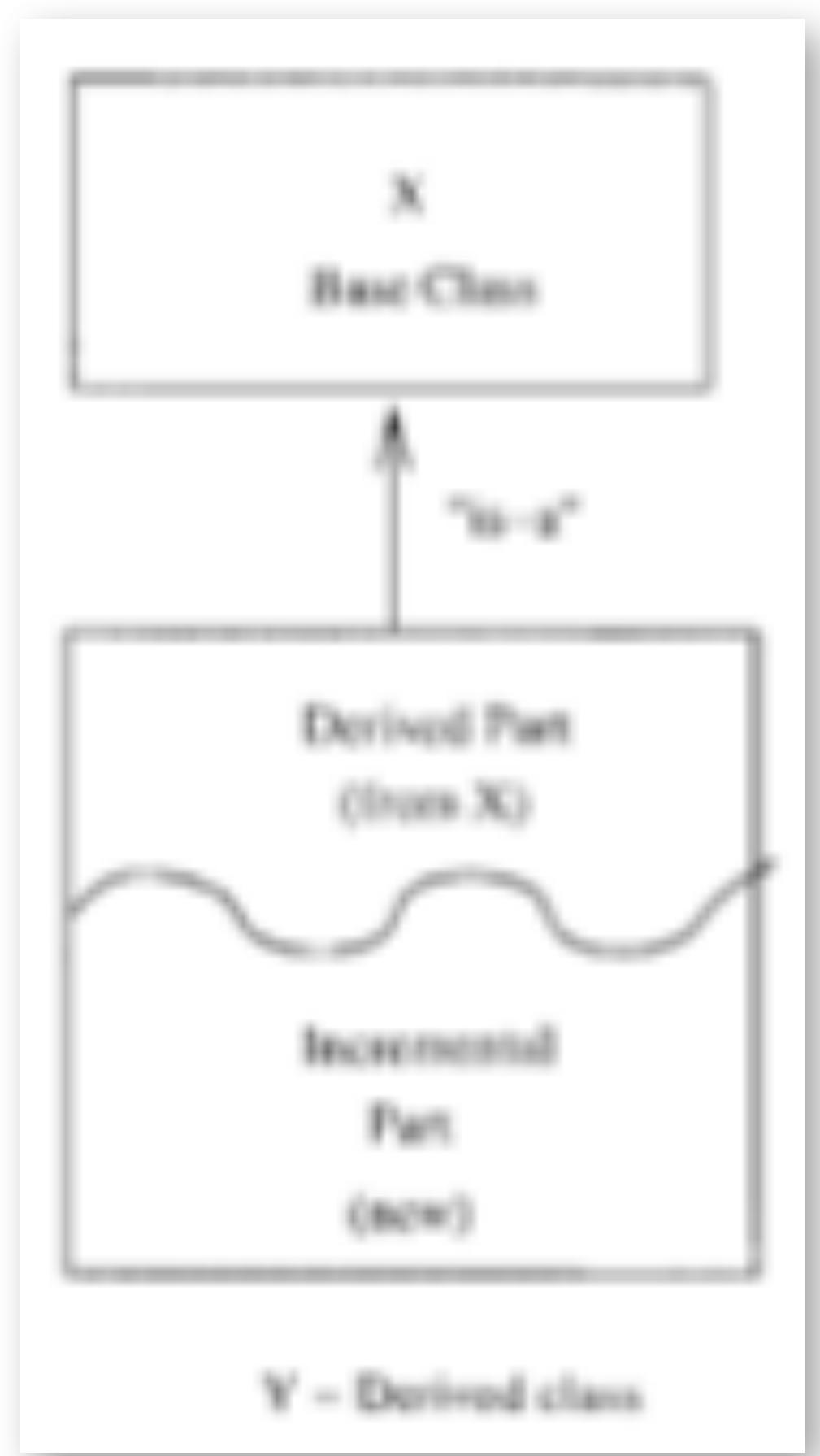
Composición:



Conceptos de la orientación a objetos

Herencia y polimorfismo

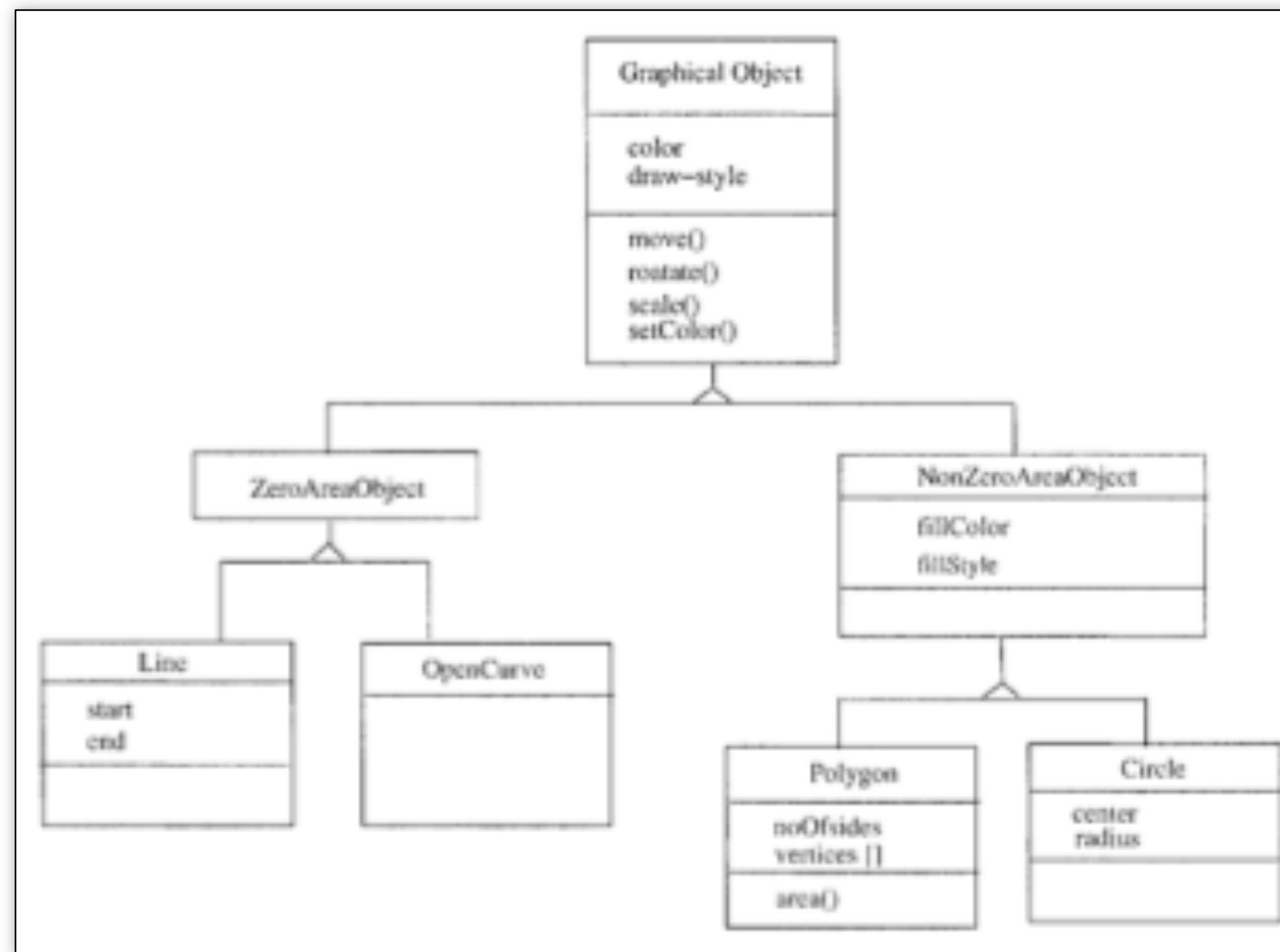
- La herencia es un concepto único en OO.
- La **herencia** es una relación entre clases que permite la definición e implementación de una clase basada en la definición de una clase existente.
- Cuando una clase Y hereda de una clase X, Y toma implícitamente todos los atributos y operaciones de X.
 - Y se denomina la subclase o clase derivada.
 - X se denomina la superclase o clase base.
- La subclase Y tiene un parte derivada (heredada de X) y una parte incremental (nueva).
- La parte incremental es la única que necesita definirse en Y.
- La herencia crea una relación “es-un”: un objeto de clase Y también es un objeto de clase X.



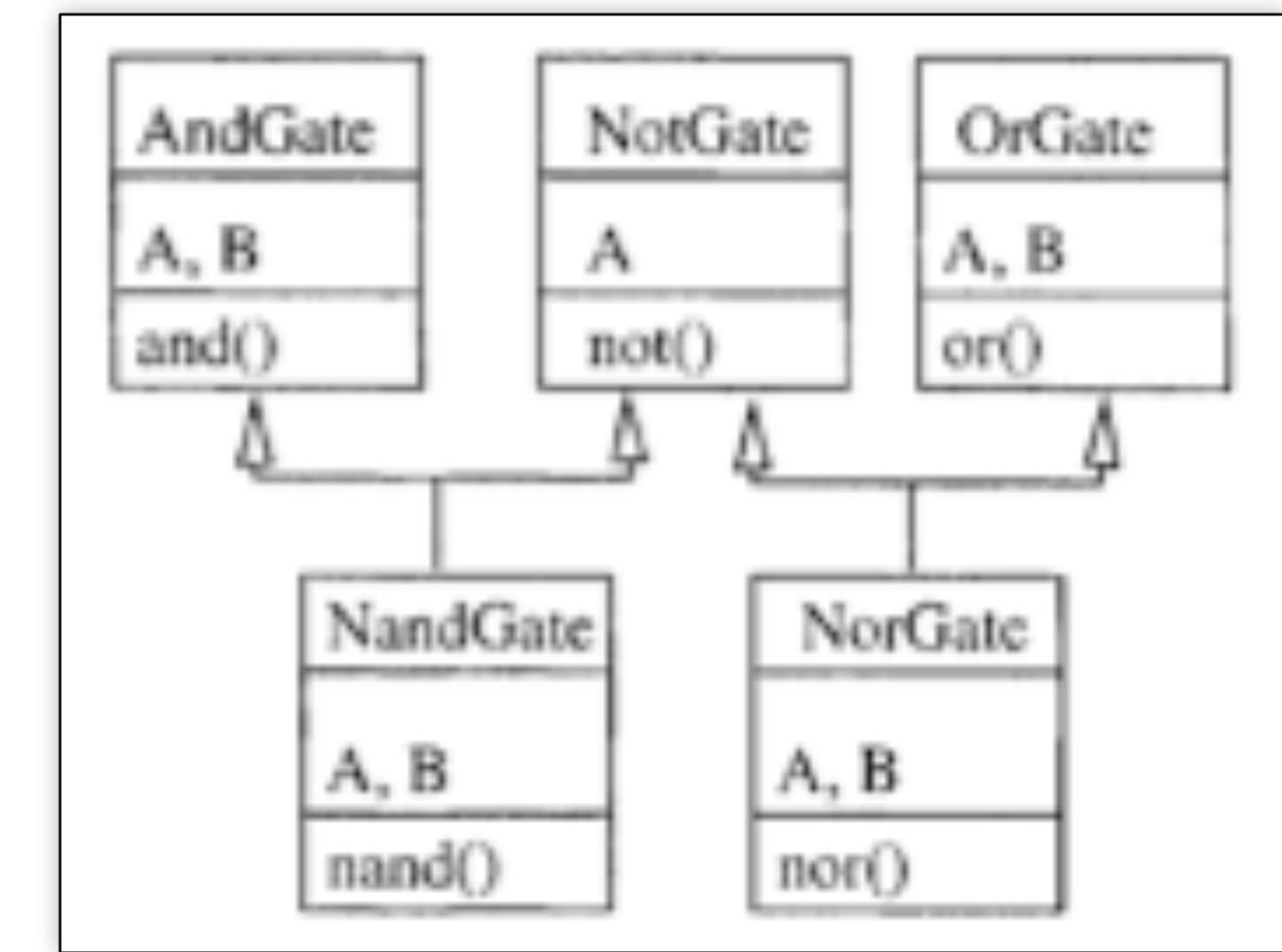
Conceptos de la orientación a objetos

Herencia y polimorfismo

La relación de herencia forma una jerarquía entre clases.



Simple



Múltiple

Conceptos de la orientación a objetos

Herencia y polimorfismo

- **Herencia estricta:** una subclase toma todas las características de su superclase.
 - Sólo agrega características para especializarla.
- **Herencia no estricta:** la subclase redefine alguna de las características.
- La herencia estricta representa limpiamente la relación “es-un” y tiene muchos menos efectos colaterales.
Es suficiente con mantener el invariante de clase y las pre/post-condiciones en las interfaces (métodos).

Conceptos de la orientación a objetos

Herencia y polimorfismo

- **La herencia induce polimorfismo:** un objeto puede ser de distintos tipos, i.e. pertenecer a distintas clases. Un objeto de tipo Y es también un objeto de tipo X, si Y es subclase de X.
 - Un objeto tiene un tipo estático y un tipo dinámico.
 - Vinculación dinámica (dynamic binding) de operaciones: el código asociado con una operación se conoce sólo durante la ejecución.

Conceptos de diseño

- Durante el diseño la actividad clave es la especificación de clases del sistema a construir.
- La corrección del diseño es fundamental.
- Pero el diseño también debe ser “bueno”: eficiente, modificable, estable, etc.
- El diseño se puede evaluar usando:
 - Acoplamiento
 - Cohesión
 - Principio abierto-cerrado

Conceptos de diseño

Acoplamiento

El acoplamiento es un concepto inter-modular, captura la fuerza de interconexión entre módulos.

Objetivo: **bajo acoplamiento**.

Tres tipos de acoplamiento:

- Acoplamiento por interacción
- Acoplamiento de componentes
- Acoplamiento de herencia

Conceptos de diseño

Acoplamiento

Acoplamiento por interacción:

- Ocurre debido a métodos de una clase que invocan a métodos de otra clase.
- Es similar al acoplamiento que se produce en diseño orientado a funciones.
- Mayor acoplamiento 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.
- 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).

Conceptos de diseño

Acoplamiento

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, o
si A tiene un método con **variables locales** de tipo C.
- Cuando A está acoplado con C, también está acoplado con todas sus subclases.
- Menor acoplamiento si las variables de clase C en A son, o bien atributos, o bien
parámetros en un método, i.e. son visibles.

Conceptos de diseño

Acoplamiento

Acoplamiento de herencia:

- Dos clases están acopladas si una es subclase de otra.
- Peor forma de acoplamiento 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 éste (debería preservar la pre/postcondición para que no sea malo).
- Menor acoplamiento si la subclase sólo agrega variables de instancia y métodos pero no modifica los existentes en la superclase.

Conceptos de diseño

Cohesión

La cohesión es un concepto intra-modular; captura cuán relacionado están los elementos de un módulo.

Objetivo: **alta cohesión**.

Tres tipos de cohesión:

- Cohesión de método
- Cohesión de clase
- Cohesión de herencia

Conceptos de diseño

Cohesión

Cohesión de método:

- ¿Por qué los elementos están juntos en el mismo método?
Similar a la cohesión en diseño orientado a funciones.
- La cohesión es mayor si cada método implementa una **única función** claramente definida con todos sus elementos contribuyendo a implementar esta función.
- Se debería poder describir con una oración simple que es lo que el método hace.

Conceptos de diseño

Cohesión

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.
- 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.

Conceptos de diseño

Cohesión

Cohesión de la **herencia**:

- ¿Por qué distintas clases están juntas en la misma jerarquía?
- Hay dos razones para definir subclases:
 - Generalización-Especialización
 - Reuso
- La cohesión es más alta si la jerarquía se produce como consecuencia de la generalización-especialización.

Conceptos de diseño

Principio abierto-cerrado (OCP)

“Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas”

B. Meyer

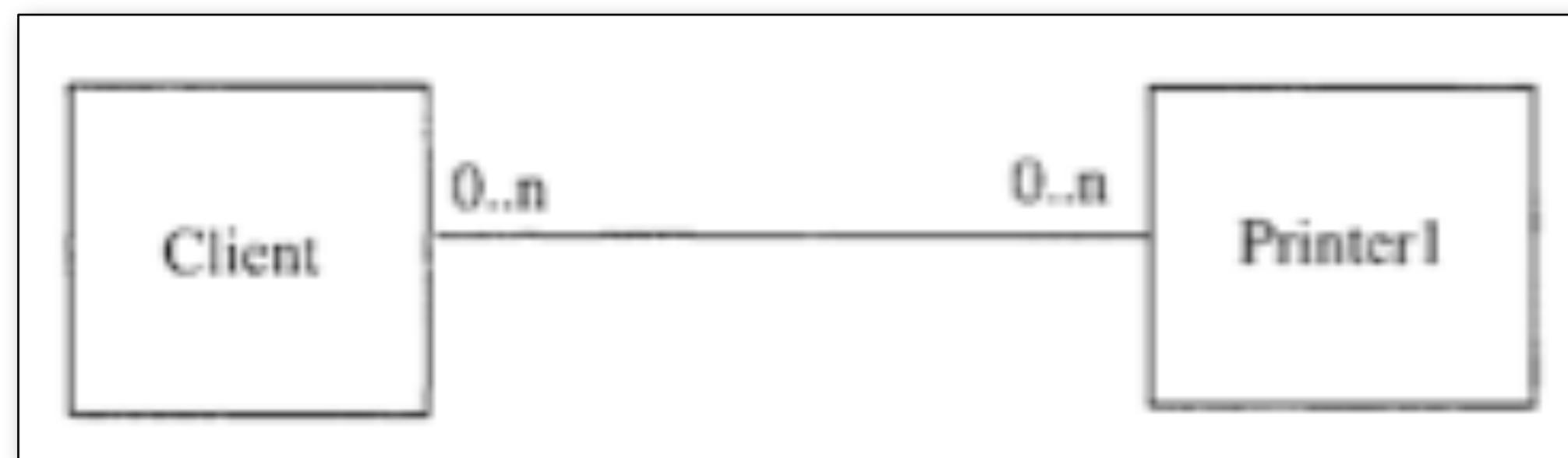
- El comportamiento puede extenderse para adaptar el sistema a nuevos requerimientos, pero el código existente no debería modificarse.
i.e.: permitir agregar código pero no modificar el existente.
- Minimiza el riesgo de “dañar” la funcionalidad existente al ingresar cambios, lo cual es una consideración muy importante al modificar código.
- Además es positivo para los programadores ya que ellos prefieren escribir código nuevo en lugar de cambiar el existente.

Conceptos de diseño

Principio abierto-cerrado (OCP)

- En OO este principio es satisfecho si se usa apropiadamente la herencia y el polimorfismo.
- La herencia permite crear una nueva (sub)clase para extender el comportamiento sin modificar la clase original.

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



Client llama
directamente a los
métodos de
Printer1

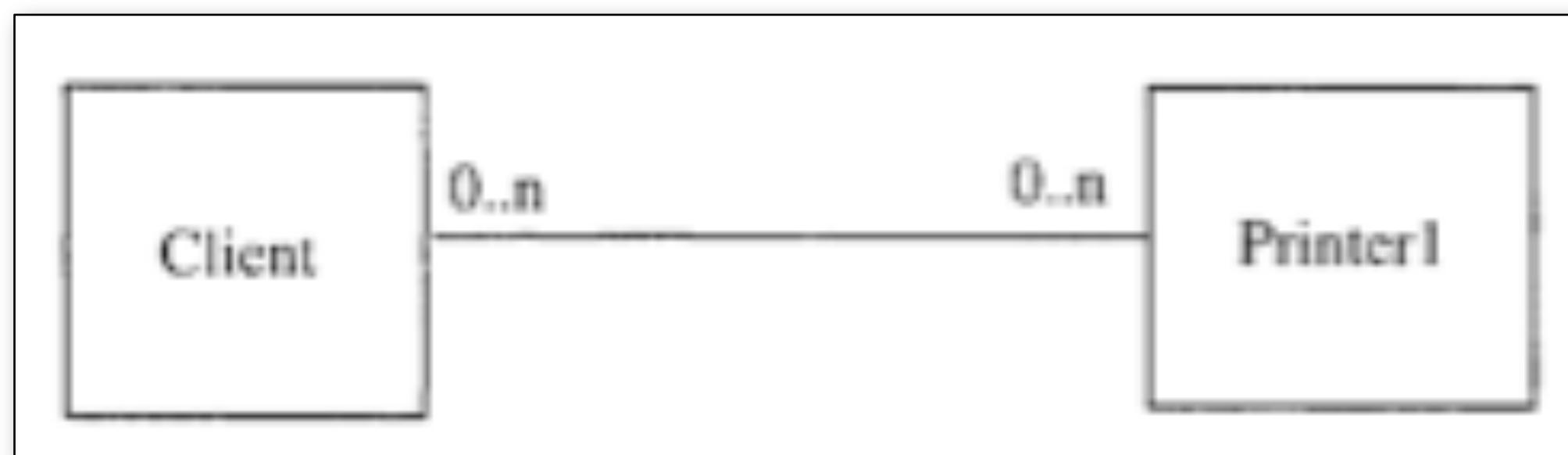
Conceptos de diseño

Principio abierto-cerrado (OCP)

- En OO este principio es satisfecho si se usa herencia y el polimorfismo.
- La herencia permite crear una nueva clase sin modificar la clase original.

Ej.: Un objeto cliente que interactúa con un

Si se permite otra clase de impresora se necesita crear otra clase “Printer2”.
Pero además se debe modificar al cliente para que la pueda usar:



Client llama directamente a los métodos de Printer1

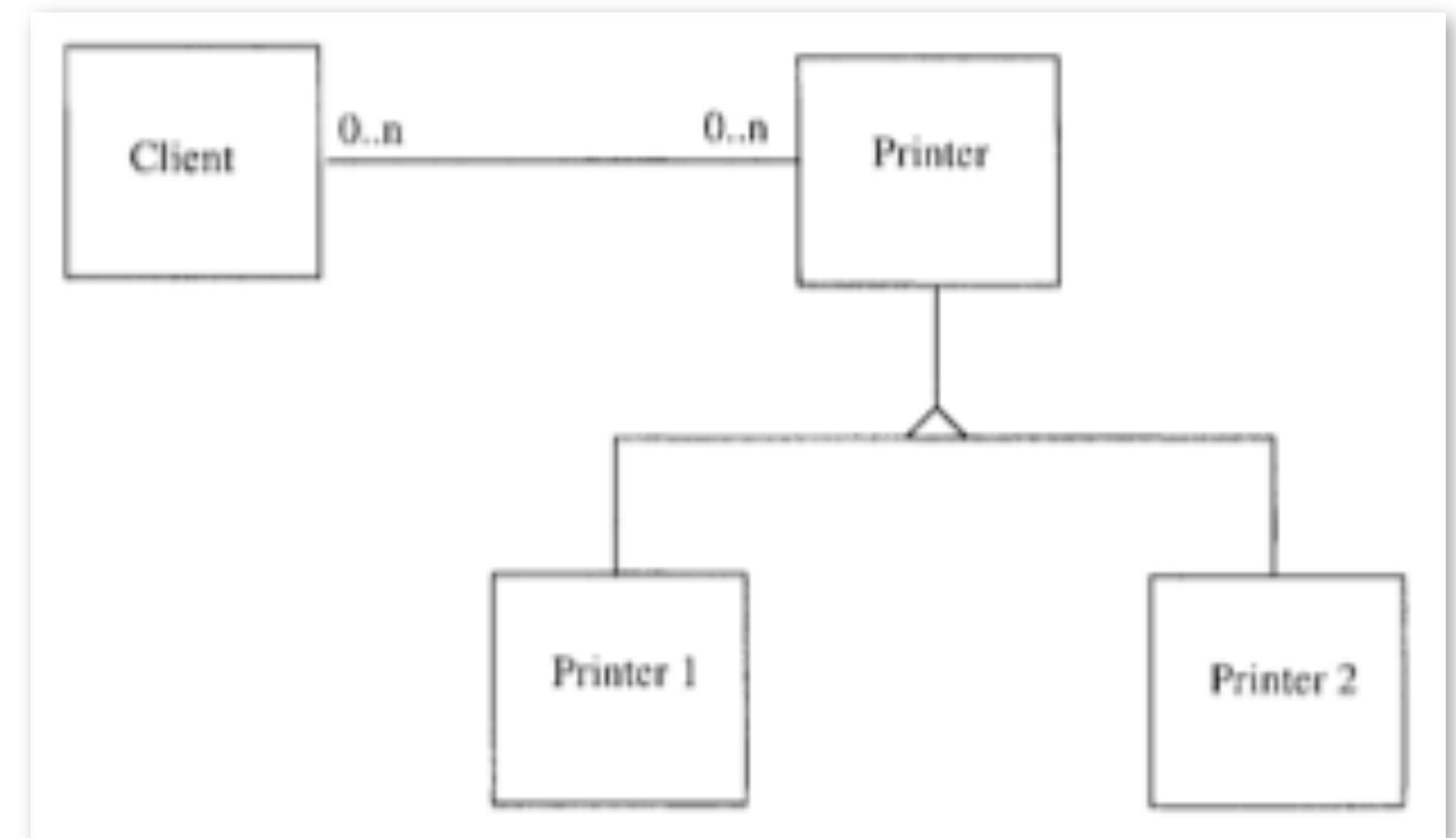
¿Qué ocurre si se permite otra clase de impresora?

Conceptos de diseño

Principio abierto-cerrado (OCP)

Alternativa:

- Definir Printer1 como subclase de una clase Printer más general.
- Client accede sólo a Printer.
- Para modificar este caso, sólo es necesario agregar Printer2 como subclase de Printer.
- Client no necesita ser modificado.



Conceptos de diseño

Principio de sustitución de Liskov (LSP)



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.

Bárbara Huberman

- Si las jerarquías de un programa siguen este principio, entonces el programa responde al principio abierto-cerrado.

Conceptos de diseño

Principio de Responsabilidad Única (RSP)

Una clase debe tener sólo una razón para cambiar, lo que significa que sólo debe tener un trabajo o responsabilidad.

- Más fácil de probar: las clases con una sola responsabilidad son más sencillas de entender y probar.
- Complejidad reducida: limita el impacto de los cambios, ya que cada clase solo se centra en una tarea.

Conceptos de diseño

Principio de segregación de interfaces (ISP)

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

- Este principio sugiere dividir las interfaces grandes en otras más pequeñas.
- Flexibilidad: Los clientes sólo necesitan conocer los métodos que les interesan.
- Mantenibilidad: Es más fácil realizar cambios ya que es menos probable que los cambios en una parte del sistema afecten a otras partes.

Conceptos de diseño

Principio de inversión de dependencia (DIP)

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

Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones. Además, las abstracciones no deberían depender de los detalles, pero los detalles deberían depender de las abstracciones.

- Desacopla: al reduce la dependencia entre diferentes partes del código.
- Permite flexibilidad haciendo más fácil de refactorizar, cambiar e implementar.

Conceptos de diseño

Principios

Principio de Responsabilidad Única (**S**ingle Responsibility)

Principio abierto-cerrado (**O**pen-Closed)

Principio de sustitución de Liskov (**L**iskov Substitution)

Principio de segregación de interfaces (**I**nterface Segregation)

Principio de inversión de dependencia (**D**evelopment Inversion)

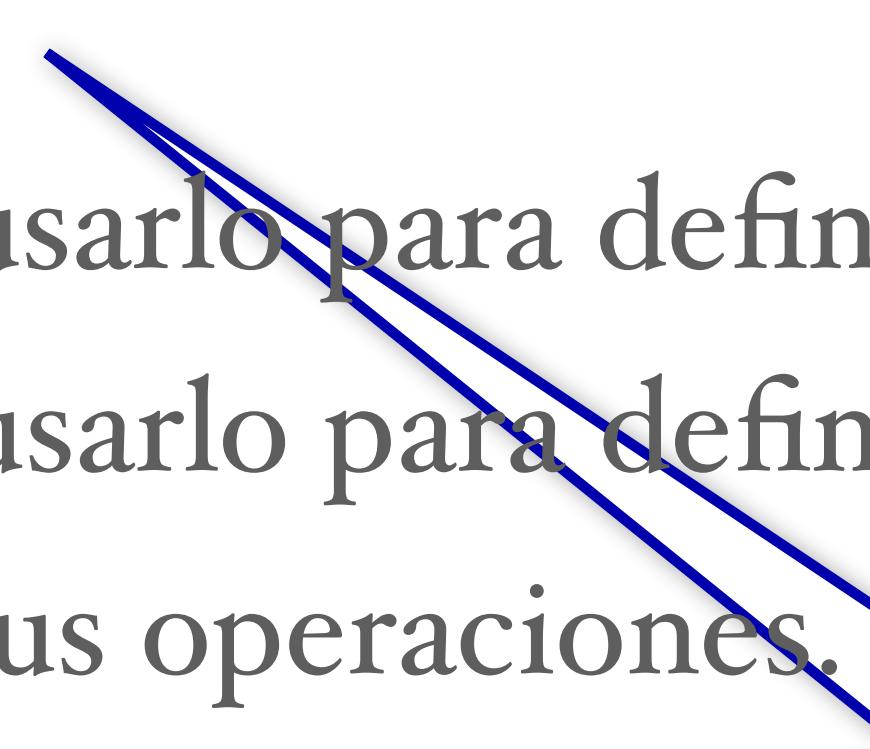
Arquitectura del software

“Some behavior suggests that while software architects like **simplicity of implementation**, software developers are often choosing whatever is **easiest to build**, rather than whatever is least risky to maintain over time”

Abdalkareem 2017

Una metodología de diseño

- El punto de partida del diseño OO es el modelo obtenido durante el análisis OO.
- 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.



Es básicamente el
diagrama obtenido en
análisis

Una metodología de diseño

2. Producir el modelo dinámico

- El diagrama de clases es una estructura estática.
- El modelo dinámico apunta a especificar cómo cambia el estado de los distintos objetos cuando ocurre un evento.
- Evento: desde el punto de vista de un objeto, es una solicitud de operación.
- Escenario: secuencia de eventos que ocurre en una ejecución particular del sistema, recordar casos de uso.
- Los escenarios permiten identificar los eventos que realizan los objetos, i.e., los servicios de los objetos.

Una metodología de diseño

2. Producir el modelo dinámico

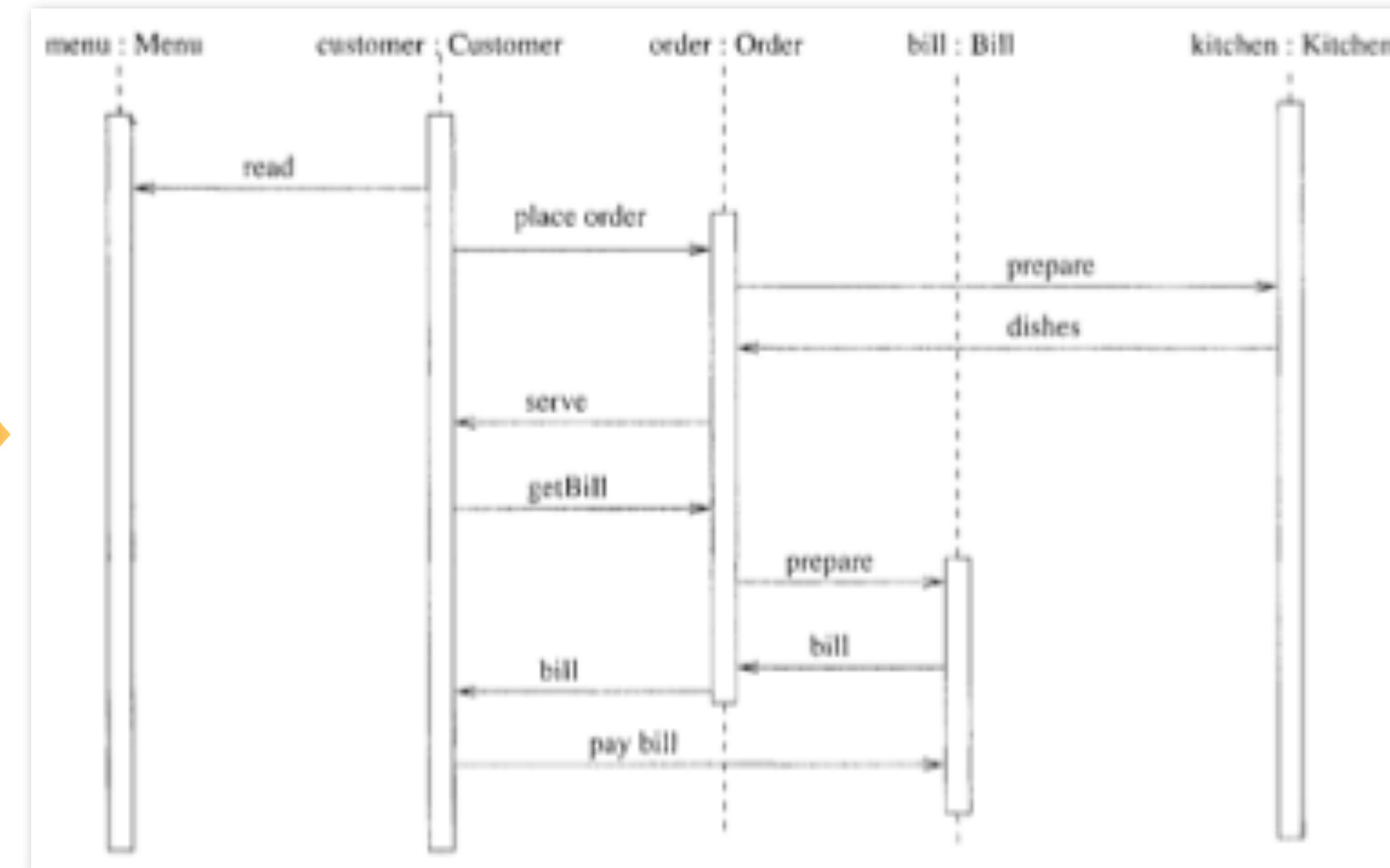
- Comenzar por los escenarios iniciados por eventos externos.
- Primero modelar los escenarios exitosos, luego los escenarios excepcionales.
- Los distintos escenarios juntos permiten caracterizar el comportamiento completo del sistema.
utilizar diagramas de interacción (secuencia/collaboración) o diagramas de estado
(ver Sección 8.1.3).

Una metodología de diseño

2. Producir el modelo dinámico

Ejemplo, el restaurant (digrama de secuencia):

- El cliente lee el menú
- El cliente realiza un pedido
- El pedido se envía a la cocina para su preparación
- Se sirven los items pedidos
- El cliente solicita la cuenta para pagar
- Se prepara la cuenta para este pedido
- El cliente recibe la cuenta
- El cliente paga la cuenta



Una metodología de diseño

2. Producir el modelo dinámico

- Una vez modelados los escenarios se reconocerán eventos de los distintos objetos (i.e. operaciones).
- Esta información se utilizará 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.
- Por lo tanto, los diagramas de secuencias nos permiten refinar nuestra visión de un objeto y agregar las operaciones necesarias que pueden no haber sido identificadas previamente.

Ejemplo: “placeOrder” y “getBill” para el objeto “Order”

Una metodología de diseño

3. Producir el modelo funcional

- El modelo funcional describe las operaciones que toman lugar en el sistema.
- Especifica cómo computar los valores de salida a partir de los valores de la entrada.
- No considerar los aspectos de control, usar DFD.
- En OO las operaciones se realizan sobre objetos.
- Los transformadores del DFD representan esas operaciones.

Deben aparecer en alguna clase como una sola operación, o como múltiples operaciones en varias clases.

Una metodología de diseño

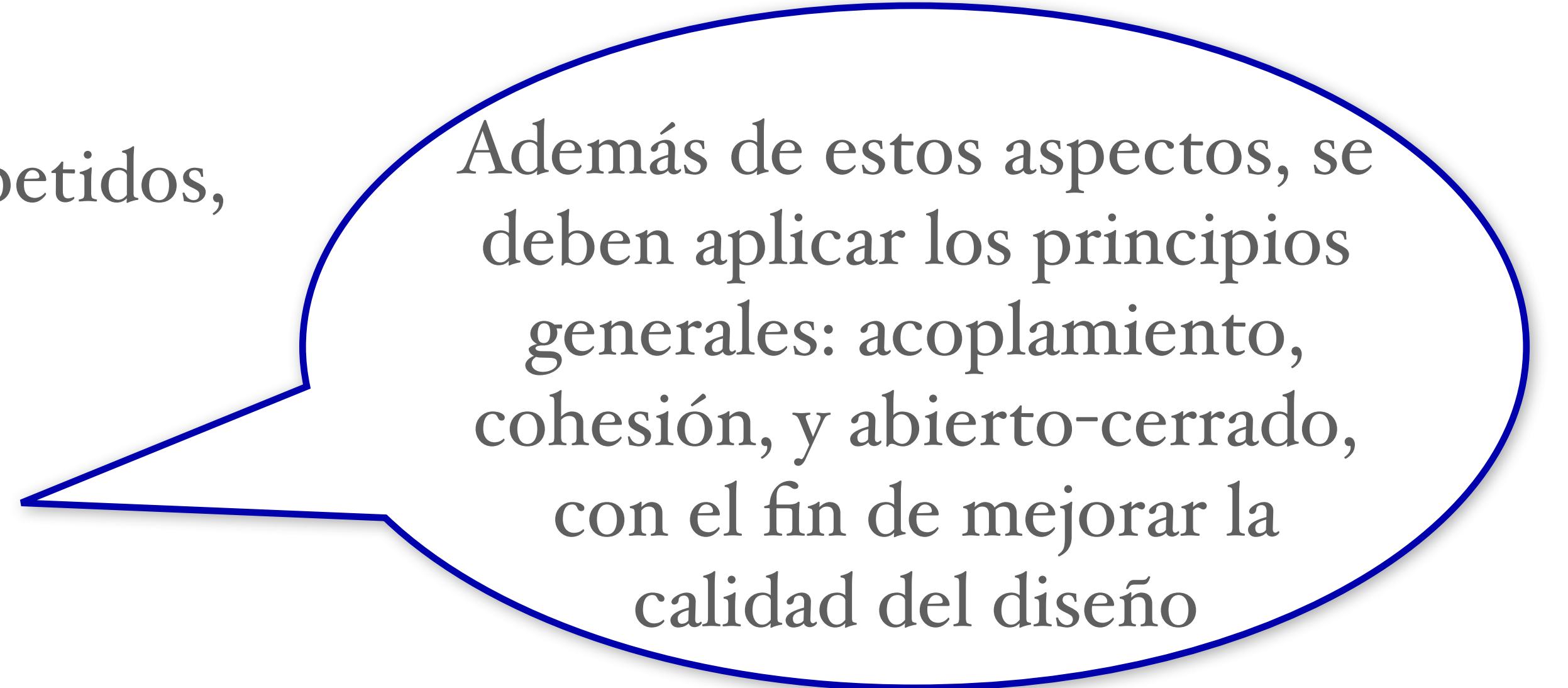
4. Definir las clases y operaciones internas

- Hasta ahora las clases provienen del dominio del problema.
- Sin embargo, el diseño final debe ser un plano de la implementación => considerar cuestiones de implementación, incluyendo algoritmos y optimización.
- Evaluar críticamente cada clase para ver si es necesaria en su forma actual, puede que algunas clases innecesarias para la implementación se descarten.
- Considerar luego las implementaciones de las operaciones de cada clase.
- Puede que necesiten operaciones de más bajo nivel sobre clases auxiliares más simples. Ejemplos: árboles, diccionarios.
- Estas clases se denominan clases contenedoras.

Una metodología de diseño

5. Optimizar

- Agregar asociaciones redundantes,
con el fin de optimizar acceso a datos.
- Guardar atributos derivados,
con el fin de evitar cálculos complejos repetidos,
asegurar consistencia.
- Usar tipos genéricos,
permite reusabilidad de código.
Ej.: lista de ... , BTree de ...
- Ajustar la herencia,
considerar subir en la jerarquía las operaciones comunes,
considerar la generación de clases abstractas por sobre otras clases,
mejora reusabilidad.



Además de estos aspectos, se deben aplicar los principios generales: acoplamiento, cohesión, y abierto-cerrado, con el fin de mejorar la calidad del diseño

Métricas

WMC

- Métodos pesados por clases (WMC)
- Profundidad del árbol de herencia (DIT)
- Cantidad de hijos (NOC)
- Acoplamiento entre clases (CBC)
- Respuesta para una clase (RFC)

Métricas

WMC

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 la clase C en consideración.
- Sea $C(M_i)$ la complejidad del método i. Ejemplo: la longitud estimada, complejidad de la interfaz, complejidad del flujo de datos, etcétera.
- Luego: $WMC = \sum_{i=1}^n C(M_i)$
- Si WMC es alto => la clase es más propensa a errores.

Métricas

DIT

Profundidad del árbol de herencia (DIT)

- Una clase muy por debajo en la jerarquía de clases puede heredar muchos métodos
=> dificulta la predicción de su comportamiento.
- DIT de C es la profundidad desde la raíz.
- Es la longitud del camino de la raíz a la clase C.
- Si herencia múltiple => camino más largo.
- Significativo en detección de clases propensas a errores:
> DIT => > probabilidad de error en esa clase.

Métricas NOC

Cantidad de hijos (NOC)

- Cantidad de subclases inmediatas de C.
- Da una idea del reuso:
 - > NOC => > reuso
- También da la idea de la influencia directa de la clase C sobre otros elementos de diseño:
 - > influencia => > importancia en la corrección del diseño de esta clase.

Métricas

CBC

Acoplamiento entre clases (CBC)

Cantidad de clases a las cuales esta clase está acoplada.

- Dos clases están acopladas si los métodos de una usan métodos o atributos de la otra.
- Usualmente se puede determinar fácilmente desde el código aunque existen formas indirectas de acoplamiento que no se pueden determinar estáticamente.
- < acoplamiento de una clase =>
 - > independencia de la clase => más fácilmente modificable.
- > CBC => > probabilidad de error en esa clase.

Métricas

RFC

Respuesta para una clase (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.
- 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.
- Es decir, la cantidad de todos los métodos de C más los métodos de otras clases que reciben un mensaje de un método de C.
- Es probable que sea más difícil testear clases con RFC más alto.
- Muy significativo en la predicción de clases propensas a errores.

Diseño del software

Lectura complementaria:

- Capítulo 6-7 Jalote