Diseño de alto nivel

- Se realiza luego de que los requerimientos estén definidos y antes de la implementación.
- Es el lenguaje intermedio entre los requerimientos y el código.
- Se comienzan a hacer representaciones más concretas.
- El resultado es un plano del sistema que **satisfaga los requerimientos** y que se utilizará para la implementación.
- Determina las mayores características de un sistema.
- Tiene un gran **impacto en testing** y **mantenimiento**.

Lo ideal es que sea **simple** y **entendible**.

Niveles en el proceso

1. Diseño arquitectónico:
   - Identifica las componentes necesarias del sistema, su comportamiento y relaciones.
   - Es más general que el diseño de alto nivel.
2. **Diseño de alto nivel**:
   - Es la vista de los módulos del sistema.
   - Cuáles son los módulos del sistema, qué deben hacer, y cómo se organizan/interconectan
   - Opciones:
     - Orientado a funciones
     - Orientado a objetos
3. Diseño detallado o diseño lógico:
   - Establece *cómo* se implementan las componentes de manera que satisfagan sus especificaciones.
   - Muy cercano al código: incluye detalles del procesamiento lógico (por ejemplo algoritmo) y de estructuras de datos.

Criterios para Evaluar el Diseño

Los criterios son usualmente subjetivos y no cuantificables. :(
Principales criterios para evaluar:

1. **Corrección**
   - ¿El diseño implementa todos los *requerimientos*?
   - ¿Es *factible* el diseño dada las restricciones?
2. **Eficiencia**

- Apropiado *uso de los recursos* del sistema (principalmente CPU y memoria).

3. **Simplicidad**

    - Facilita la *comprensión* del sistema.
    - Facilita el testing, modificación de código, mantenimiento, descubrimiento y corrección de bugs.

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

Principios de diseño

No hay una serie de pasos que permitan derivar el diseño a partir de los requerimientos. Pero existen "ayudas" que son **principios fundamentales** que guían el proceso de diseño:

Partición y jerarquía.

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

La comunicación agrega complejidad: A medida que la cantidad de componentes aumenta, el *costo del particionado* + la *complejidad de la comunicación* también aumenta. Hay que detener el particionado cuando el costo supera al beneficio.

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".

La abstracción es esencial en el particionado del problema, pues `Jerarquía => Abstracción` aunque no necesariamente se cumple `Abstracción => Jerarquía`.

Abstracción

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

Abstracción durante el proceso de diseño:

- 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.

Mecanismos comunes de abstracción

1. *Abstracción funcional*
    - Especifica el comportamiento funcional de un módulo.
    - Los módulos se tratan como funciones de entrada/salida.
    - Puede especificarse usando pre y post condiciones.
2. *Abstracción de datos*
    - Una entidad del mundo que provee servicios al entrono.
    - Los datos se tratan como objetos junto a sus operaciones (orientación a objetos). Y las operaciones definidas para un objetos solo pueden realizarse sobre este objeto.
    - Lenguajes que soportan abstracción de datos: Ada, C++, Modula, Java

Modularidad

Un sistema se dice modular si consiste de *componentes discretas* que se pueden *implementar separadamente*; donde un cambio a una de ellas tenga *mínimo impacto* sobre las otras.

- Prove la abstracción en el software.
- Es el soporte de la estructura jerárquica de los programas.
- Mejora claridad de diseño y facilita la implementación.
- Reduce costos de testing, debugging y mantenimiento.

Necesita *criterios de descomposición*: resulta de la conjunción de la *abstracción* y el *particionado*.

⤵ Estrategias top-down y bottom-up ⤴

Enfoques para diseñar la jerarquía de componentes.

Top-down:

- El *refinamiento* de más general a más especifico. Hasta que pueda ser implementado directamente.
- Ventaja: En cada paso existe una clara imagen del diseño.
- Desventaja: se puede asumir que un módulo se pueda hacer pero al final no sea posible: *factibilidad desconocida* hasta el final.

Bottom-up:

- Comienza por las componentes de más bajo nivel en la jerarquía.

- Se usa cuando hay mucho re-uso.

Diseño orientado a funciones

Módulos

Un módulo es una **parte lógicamente** separable de un programa.
Es una unidad **discreta** e **identificable**.

Criterios para seleccionar módulos que soporten abstracciones bien definidas: acoplamiento y cohesión.

Acoplamiento

Definición

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

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

El nivel de acoplamiento se define a nivel de diseño arquitectónico y de alto nivel. Y no puede reducirse durante la implementación.

Ventajas de la independencia

Los módulos se pueden implementar, modificar y testear separadamente.

No es necesario comprender todos los módulos para comprender uno en particular: Cuanto más conexiones hay entre dos módulos, más dependientes son uno del otro, es decir, se requiere más conocimiento de un módulo para comprender el otro.

Objetivo

Los módulos deben estar tan *débilmente acoplados* como sea posible.

En un sistema *no existe la independencia entre todos los módulos* pues deben cooperar entre sí. Pero queremos que la dependencia sea mínima.

Factores que influyen en el acoplamiento

1. **Tipo de conexiones entre módulos**: La *complejidad* y *oscuridad* de lasinterfaces. Ejemplo: ¿Utilizo solo las interfaces especificadas o también uso datos compartidos?
2. **Complejidad de las interfaces**: ¿Estoy pasando solo los parámetros necesarios? De todas maneras, cierto nivel de complejidad en las interfaces es necesario para soportar la comunicación requerida con el módulo.

3. **Tipo de flujo de información entre módulos**: ¿Estoy pasando parámetros de control (flags)? si se pasan flags que determinan el un caso de la función, es probable que se pueda dividir en dos funciones. Transferencia de información de control permite que las acciones de los módulos dependan de la información; y hace que los módulos sean más difíciles de comprender.

El acoplamiento disminuye si:

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

El acoplamiento se incrementa si:

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

Cohesión

Definición

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

Consiste en minimizar las relaciones entre los elementos de los distintos módulos y maximizando las relaciones entre los elementos del mismo módulo.

Objetivo

Alta cohesión. Usualmente, a mayor cohesión de los módulos, menor acoplamiento.

Tipos de cohesión

1. **Casual**: La relación entre los elementos del módulo no tiene significado.
2. **Lógica**: Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.
3. **Temporal**: Los elementos están relacionados en el tiempo y se ejecutan juntos. Ejemplo: inicialización, clean-up, finalización.
4. **Procedural**: Contiene elementos que pertenecen a una misma unidad procedural. Ejemplo: un ciclo o secuencia de decisiones.

5. **Comunicacional**: Tiene elementos que están relacionados por una referencia al mismo datos. Ejemplo: pedir los datos de una cuenta personal y devolver todos los datos del registro.
6. **Secuencial**: Los elementos están juntos porque la salida de uno corresponde a la entrada del otro. Es relativamente buena cohesión y relativamente fácil de mantener, pero difícil de reusar.
7. **Funcional**: Todos los elementos del módulo están relacionados para llevar a cabo una sola función. Ejemplo: Calcular el seno de un ángulo.

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** (ejemplo: 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**.

Notación y especificación del diseño

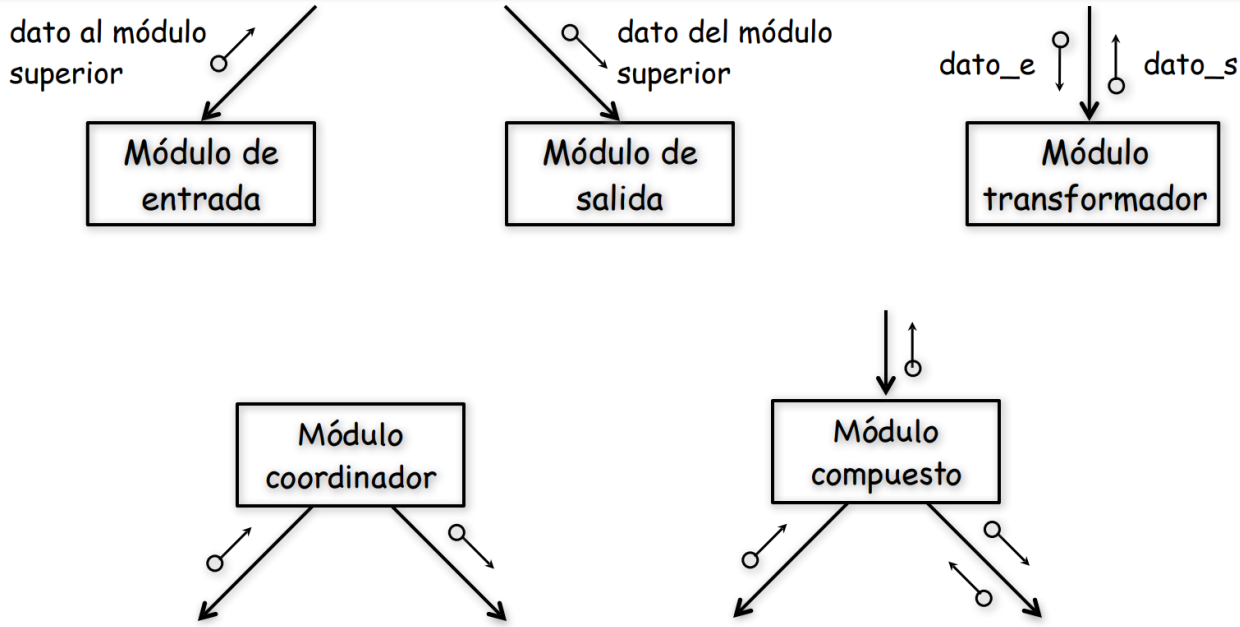
Nos interesan el **diseño del sistema** (el producto de la fase) y el **proceso** que lleva a cabo el diseño.

Diagramas de estructura

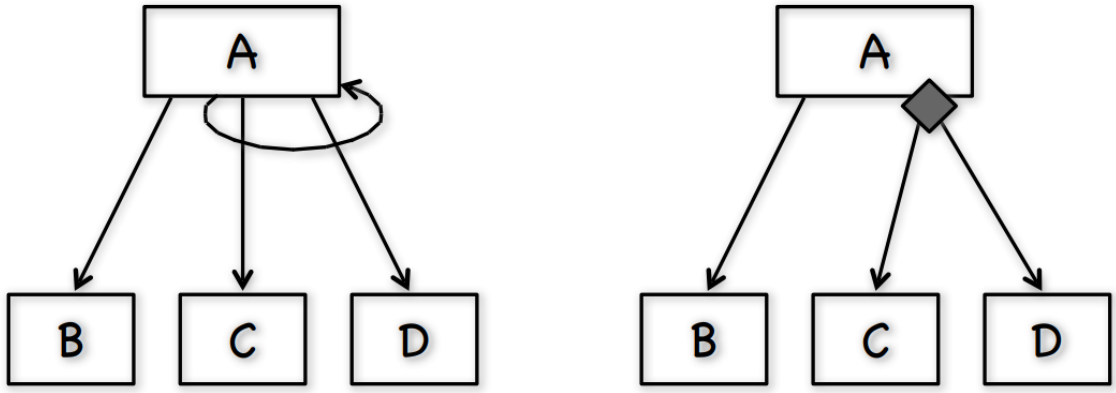
Presenta una notación gráfica para la *estructura de un programa*.
Representa módulos y sus interconexiones. Se puede realizar una correlación entre código y diagramas de estructura.

La invocación de A a B se representa con una flecha; cada flecha se etiqueta con los ítems que se pasan.

Tipos de módulos:



Iteración y decisión



No es intención de los diagramas de estructura mostrar la lógica del programa. Solo se indican las más importantes.

Metodología del diseño estructurado

La estructura se decide durante el diseño y la implementación NO debe cambiar la estructura. La metodología de diseño estructurado (SDM: *Structured Design Method*) apunta a controlar la estructura y proveer pautas para auxiliar al diseñador en el proceso de diseño. SDM es una metodología orientada a funciones.

Pautas

- Los módulos *subordinados* son los que realizan la mayoría de la computación. El procesamiento real se realiza en los módulos *atómicos* del nivel más bajo.
- 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 realice en módulos subordinados.

Pasos principales

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

Reformular el problema como un DFD

Se toma el *flujo de datos* de todo el sistema propuesto. Y se plantea un DFD, que proveerá una visión de alto nivel del sistema.

Si bien *se ignoran aspectos procedurales* y la *notación es la misma*, el *propósito* del DFD es *diferente* al del de análisis de requerimientos.
(No nos interesa entender el problema, sino empezar a desarrollar la solución).

Identificar las entradas y salidas abstractas

MAI (*Most Abstract Input*)

- La *MAI* consiste en la *entrada* y todas las *transformaciones* que se le aplican a la misma para que esté en un *formato adecuado*.
- 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.

MAO (*Most Abstract Output*)

- Es dual a la *MAI*.
- Elementos de datos en el DFD que están más distantes de la salida real, pero que aún puede considerarse como salida.

Determinar las *MAI* o las *MAO* es subjetivo, es decir, representan un juicio de valor.

Las **transformaciones centrales** entre la *MAI* y la *MAO* es donde el sistema realmente "hace algo". Y estos se concentran en la transformación sin importar el formato/validación/etc de las entradas y salidas.

Entonces, el primer diseño básico consiste en:

MAI	→	Sistema intermedio	→	MAO
entrada	→	transformaciones	→	salida

Realizar el primer nivel de factorización

Hay que subdividir en:

1. *Modulo principal*: Módulo coordinador
2. *Módulos subordinados*
    1. De entrada: responsables de entregar las entradas lógicas
    2. Transformadores: consumen las entradas lógicas y obtienen las salidas lógicas
    3. De salida: Consumen las salidas lógicas
    - Cada uno de los tres tipos de módulos pueden diseñarse separadamente y son independientes.

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.
- Usualmente no debería haber módulos de salida.

Factorizar los módulos de salida

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

Factorizar los transformadores centrales

- Utilizar un proceso de refinamiento top-down.
- El objetivo es determinar los sub-transformadores que compuestos conforman el transformador.
- Graficar DFD.
- Repetir hasta alcanzar los módulos atómicos.

Mejorar la estructura

Mejorar la estructura (heurísticas, análisis de transacciones)

Esta etapa consiste en hacer las *modificaciones finales*.
Dada una serie de heurísticas se determina si la estructura está bien hecha.
Y si no lo es, modificar.

El objetivo es tener el menor grado de acoplamiento y tener mucha cohesión en los módulos.

Heurísticas "Rules of thumb"

1. *Tamaño del módulo*: Indicador de la complejidad del módulo. Examinar los módulos con muy pocas líneas o con más de 100 líneas.

2. *Cantidad de flechas de salida y entrada*: La primera no debería exceder las 5 o 6 flechas; la segunda debería maximizarse.
3. *Alcance del efecto de módulo*: Los módulos afectados por una decisión en este módulo (pueden ser más profundo que solo los inmediatos).
4. *Alcance del control de un módulo*: A dónde van las flechas. Todos los subordinados.

Lo ideal es que `efecto == control`. Mientras menos se propaguen los problemas, mejor. Una decisión debe tener efecto solamente en los módulos subordinados.

Verificación

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

Hacer análisis de desempeño, eficiencia, etc.

Si se usan lenguajes formales para representar el diseño => existen *herramientas* que asisten para el análisis... Se usan también *listas de control*.

La calidad del diseño se completa con una buena modularidad (*bajo acoplamiento y alta cohesión*).

Métricas

Objetivo: proveer una evaluación cuantitativa del diseño (así el producto final puede mejorarse).

Se aplican dentro de UN MISMO proyecto y siempre se debe ser consistente.

1. **Tamaño estimado:** `cantidad de módulos + tamaño estimado de c/u`
2. **Complejidad de módulos:** cuáles son los que van a tomar más tiempo testear.
3. **Métricas de red:** enfocado en la estructura del diagrama.
    1. Se considera un buen diagrama aquel en el cual cada módulo tiene sólo un módulo invocador (reduce acoplamiento).
    2. Impureza del grafo = `nodos_del_grafo - aristas_del_grafo - 1` si impureza = 0 tenemos un árbol.
4. **Métricas de estabilidad:** Trata de capturar el *impacto de los cambios* en el diseño. La estabilidad es la cantidad de suposiciones por otros módulos sobre uno específico.
5. **Métricas de flujo de información:** Computar la complejidad inter-módulo que se estima con inflow y outflow (el flujo de info que entra o sale del módulo).
    - `DC = tamaño * (inflow * outflow)²`
    - `DC = fan_in * fan_out + inflow * outflow` Donde fan_in representa la cantidad de módulos que llaman a C y fan_out los llamados por C.
    - Propenso a error si `DC > complejidad_media + desviación_estándar`
    - Complejo si `complejidad_media < DC < complejidad_media + desviación_estándar.`

Diseño orientado a objetos

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

Análisis OO: Primer paso (previo a la SRS).
Existe métodos que combinan análisis y diseño (ADOO)

Análisis	Diseño
Dominio del problema	Dominio de la solución
El objetivo es entender el sistema	El objetivo es modelar una solución
Los objetos representan un concepto del problemas. Son objetos semámticos	* Produce objetos semánticos, de interfaces, de aplicaciones y de utilidad
	Hace incapié en el comportamiento **dinámico** del sistema

*

- Objetos de interfaces: se encargan de la interfaz con el usuario
- Objetos de aplicaciones: Especifican los mecanismos de control para la solución propuesta.
- Objetos de utilidad: Son los necesarios para soportar los servicios se los objetos semánticos (ejemplo: pilas, árboles, diccionarios, etc).

Conceptos de la Orientación a Objetos

Clases

Es una "plantilla". Las clases definen un tipo, los objetos son sus instancias.

Una clase tiene interfaz, cuerpo y variables:

Componente	Descripción
Interfaz	Cuáles partes de un objeto puede accederse desde el exterior
Cuerpo	Implementa las operaciones
Variables de instancia	Sirven para retener el estado del objeto

Las operaciones pueden ser:

- **públicas:** accesibles del exterior
- **privadas:** accesibles sólo dentro de la clase.
- **protegidas:** accesibles desde dentro de la clase y desde sus subclases.

Objetos

La propiedad básica de los objetos es el **encapsulamiento**. Los objetos encapsulan datos e información y proveen interfaces para accederlos y modificarlos. El encapsulamiento brinda abstracción y ocultamiento de información.

Los objetos, a diferencia de las funciones, tienen

- **estado persistente** tracking del estado.
- **identidad:** cada objeto puede ser identificado y tratado como una entidad.

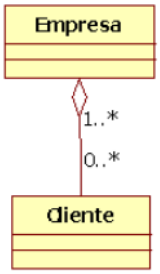
El comportamiento del objeto queda definido conjuntamente por los servicios y el estado.

Relaciones entre objetos

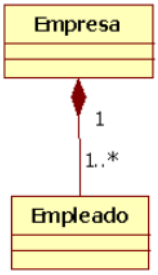
Asociación	Agregación	Composición
Un objeto está vinculado durante un tiempo con otro	Un objeto es parte de otra clase	Un objeto está compuesto por otros, no existe sin la presencia ellos.
Se envían mensajes o se solicitan servicios	Relación derivada de una asociación, en general es una colección. Sus ciclos de vida no están relacionados (relación unidireccional)	El ciclo de vida de ambos objetos están muy relacionados.

Asociación:

Agregación:



Composición:



Herencia

La **herencia** es una relación entre clases que permite la definición e implementación de una clase basada en la definición de una clase existente.

Si una clase *Y* hereda de una clase *X*, *Y* toma implícitamente todos los atributos y operaciones de *X*.

- *X*: superclase o clase base
- *Y*: subclase o clase derivada. Tiene una parte derivada (heredada de *X*) y una parte incremental (nueva).

Herencia múltiple: puede generar conflictos. El polimorfismo es incluso más complejo. Mejor no usarlas a menos que sea MUY necesario.

- **Herencia estricta:** No se redefinen métodos. Solo se agregan características para especializarla. (Es suficiente con mantener el invariante de clase y las pre/post condiciones en las interfaces).
- **Herencia no estricta:** Se reescriben métodos. Es mala costumbre.

Forma una jerarquía entre clases; y crea una relación "es un" (un objeto de una subclase es un objeto de la superclase).

`herencia => polimorfismo` pues un objeto puede ser de distintos tipos (pertenecer a distintas clases).

Polimorfismo

Hay **polimorfismo** si un objeto de tipo *Y* es también un objeto de tipo *X* (si *Y* es subclase de *X*).

El polimorfismo produce **vinculación dinámica** de operaciones (*dynamic binding*), es decir, el código asociado con una operación se conoce sólo durante la ejecución.

Conceptos de diseño

El diseño se puede evaluar usando:

- **Acoplamiento**
- **Cohesión**
- **Principio abierto-cerrado**

Acoplamiento

Tipos de acoplamiento

Interacción:

- Métodos de una clase *invocan* a métodos de otra clase.
- No se puede eliminar del todo.
- *Menor acoplamiento*: si se comunican a través de la menor cantidad de parámetros pasando menos información y nada de control.
- *Mayor acoplamiento*: Los métodos acceden a partes internas de otros métodos o variables. O la información se pasa a través de variables temporales.

Componentes:

- Una clase *A* tiene variables de otra clase *C*. Si *A* tiene variables de instancia, parámetros o métodos con variables locales de tipo *C*.
- Cuando *A* está acoplada con *C*, también está acoplada con todas sus subclase.
- Mejor Si las variables de la clase C en A son, o bien atributos o parámetros en un método. Es decir, son *visibles*.

Herencia:

- Si una es subclase de otra.
- Lo malo es si se modifican la *signatura* de un método o *eliminan* un método (herencia no estricta). O si cambia la pre y post condición (*especificación*).
- Menor acoplamiento si la subclase solo agrega variables de instancia y métodos pero no modifica los existentes en la superclase.

Cohesión

Tipos de cohesión

Cohesión de método

- Es mayor si cada *método* implementa una *única función* claramente definida con todos sus elementos contribuyendo a implementar esta función.
- Se debería poder escribir una oración simple de lo que hace un método.

Cohesión de clase

- Es mayor si una *clase* representa un *único concepto* con todos sus elementos contribuyendo a este concepto.
- Se pueden detectar múltiples conceptos si los métodos se pueden separar en diversos grupos, cada grupo accediendo a distintos subconjuntos de atributos.

Cohesión de Herencia:

- Es mayor si la jerarquía se produce como consecuencia de la *generalización-especificación*.
- Cohesión por reuso no esta tan bueno.

Principio abierto-cerrado

"Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas."

El comportamiento puede extenderse para adaptar el sistema a nuevos requerimientos, pero el código existente no debería modificarse. Es decir, permitir *agregar* código pero no modificar el existente.

Este principio se satisface 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.

Evitar la herencia no estricta.

Si se cumple el principio de Sustitución de Liskov se cumple el principio abierto-cerrado:

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.

En general `Liskov => abierto-cerrado`.

Metodología de diseño

Pasos de la metodología OMT (*Object modeling technique*).
El punto de partida es el modelo obtenido durante el análisis OO.
El producto final debe ser un plano de la implementación (incluyendo algoritmos y optimización).

1. **Producir el diagrama de clases**
   - Básicamente el diagrama obtenido en el análisis. Explica qué pasa en el sistema. *Estructura estática*.
2. **Producir el modelo dinámico y usarlo para definir operaciones de las clases.**
   - Describe la *interacción* entre objetos. *Estructura de control*.
   - Apunta a especificar cómo cambia el estado de los distintos objetos cuando ocurren un evento (solicitud de operación).
   - Los escenarios son la secuencia de eventos que ocurren en una ejecución particular del sistema; permiten identificar los eventos. Todos los escenarios juntos permiten caracterizar el comportamiento completo del sistema.
   - Para el diagrama de secuencia: empezar por escenarios iniciados por eventos externos → escenarios exitosos → escenarios excepcionales.
   - Una vez reconocidos los eventos de los objetos, se expande 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.
3. **Producir el modelo funcional y usarlo para definir operaciones de las clases**
   - Define la *transformación* de los datos. *Estructura de cómputo*.
   - Describe las operaciones que toman lugar en el sistema; y especifica cómo computar los valores de salida a partir de los valores de la entrada.
   - No considera los aspectos de control (usar DFD).
4. **Identificar las clases internas y sus operaciones**
   - Considera cuestiones de implementación.
   - Evaluar críticamente cada *clase* para ver si es necesaria en su forma actual.
   - Considerar luego las *implementaciones* de las operaciones de cada clase.
5. **Optimizar y empaquetar**
   - Agregar asociaciones redundantes (optimizar el acceso a datos).
   - Guardar atributos derivados (evitar cálculos complejos repetidos y asegurar consistencia).
   - Usar tipos genéricos (permite reusabilidad de código).
   - Ajustar la herencia (considerar subir en la jerarquía operaciones comunes, considerar la generación de clases abstractas para mejorar la reusabilidad).

Métricas

Sirven para repensar el diseño o identificar que componentes necesitarán más atención en testing.

Métodos pesados por clases (WMC)

La complejidad de la clase depende de la **cantidad de métodos en la misma y su complejidad**.
`M1, ..., Mn` son métodos de `C` y `C(Mi)` su complejidad (ej.: la longitud estimada, complejidad de la interfaz, complejidad del flujo de datos, etc)

$$WMC = \sum_{i=1}^{n} C(M_i)$$

Si WMC es alto, la clase es más propensa a errores.

Profundidad del árbol de herencia (DIT)

Una clase muy por debajo en la jerarquía de clases puede heredar muchos métodos y dificulta la predicción de su comportamiento.
`mayor DIT => mayor probabilidad de errores`

Cantidad de hijos (NOC)

Cantidad de subclases inmediatas de C.
`mayor NOC => mayor reuso`
`mayor NOC => mayor influencia => mayor importancia en la corrección del diseño de esta clase`

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.

`menor CBC => mayor independencia => más modificable`
`mayor CBC => mayor probabilidad de error`

Respuesta para una clase (RFC)

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 probable que sea más difícil testear clases con RFC más alto.
Muy significativo en la predicción de clases propensas a errores.

Practico