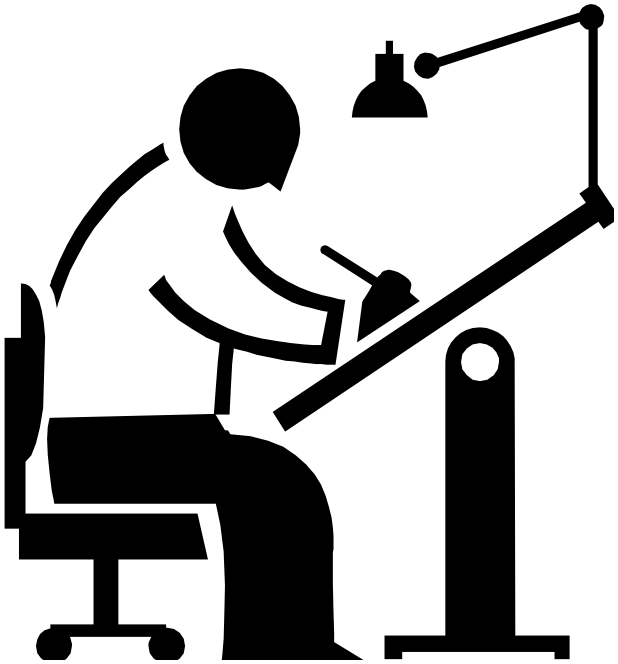
Universidad
Rey Juan Carlos

Diseño y Arquitectura de Software
Tema 4: Principios del Diseño del Software

Carlos E. Cuesta

Depto. de Lenguajes y Sistemas Informáticos II
Universidad Rey Juan Carlos

¿Qué es diseño?

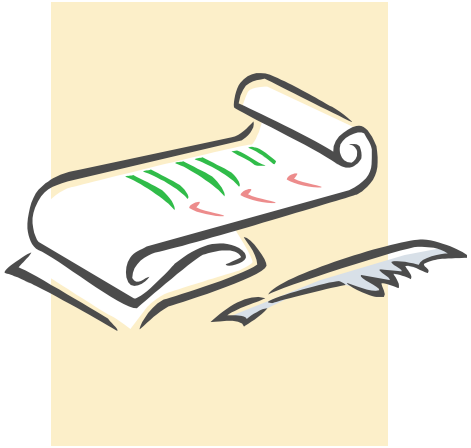


- Primer paso en la fase de desarrollo de un sistema
- Es el proceso de aplicar un conjunto de técnicas y principios con el propósito de definir un dispositivo, proceso o sistema con el suficiente detalle como para determinar su realización física.

Principios Generales de Diseño

- Modularidad
  - Descomposición del sistema en módulos o componentes
    - Divide y vencerás
  - Determinar relaciones
  - Especificar interfaces
  - Describir funcionalidades de cada módulo
- Diseño flexible y extensible
  - Abstracción
  - Gestión de la complejidad
- Reusabilidad
  - Fomentar y utilizar.
- Portabilidad
- Anticiparse a la obsolescencia

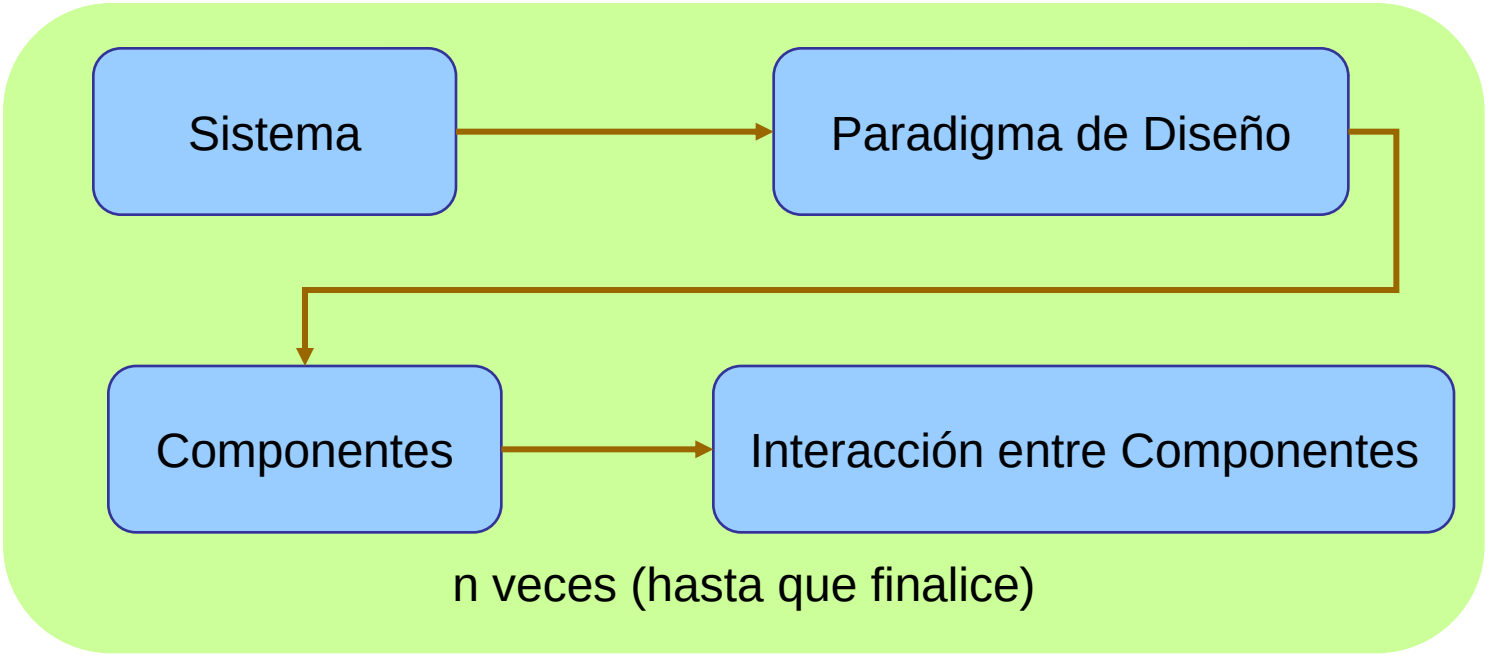
Reglas Generales de Diseño



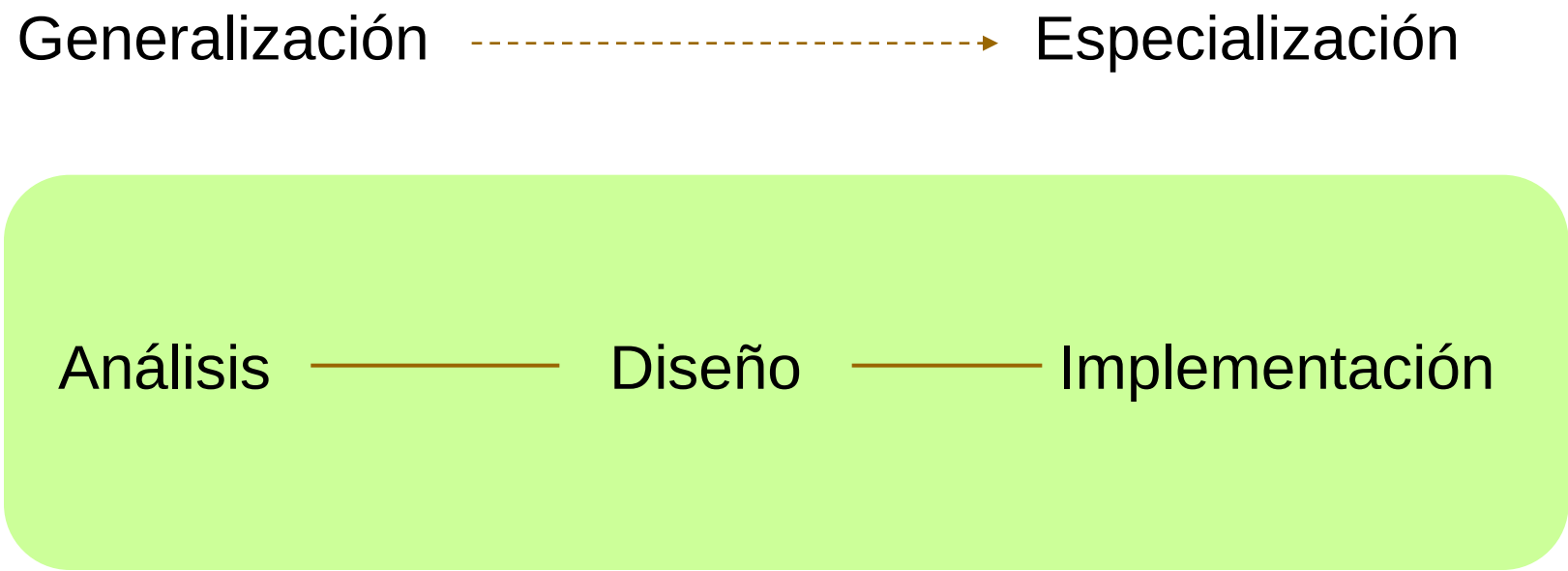
- Un buen diseñador mira alternativas
- Navegación entre el diseño y el análisis
- Reutilizar
- Acercamiento a la estructura del dominio del problema
- Diseño con uniformidad e integración
- Estructurarse para admitir cambios
- Degradación gradual
- Diseñar no es codificar, ni codificar es diseñar
- Hay que valorar la calidad del diseño mientras se crea, no cuando ya se ha terminado

Descomposición

- Gestión de la complejidad
- Divide problemas grandes en problemas pequeños



Abstracción



Abstracción

- En DOO, las clases se pueden definir como abstracciones.
- Los constructores de clases siguen el principio de ocultamiento
- Los objetos no pueden, al menos no deberían, cambiar el estado de otros objetos de forma inesperada
- Alternativas para fomentar la abstracción
  - Variables privadas
  - Pocos métodos públicos
  - Uso de superclases e interfaces
  - Los métodos como abstracciones de procedimiento.

Ocultación de Información

Comunicar solo la información necesaria

ABSTRACCIÓN

Define las entidades procedimentales

OCULTACIÓN

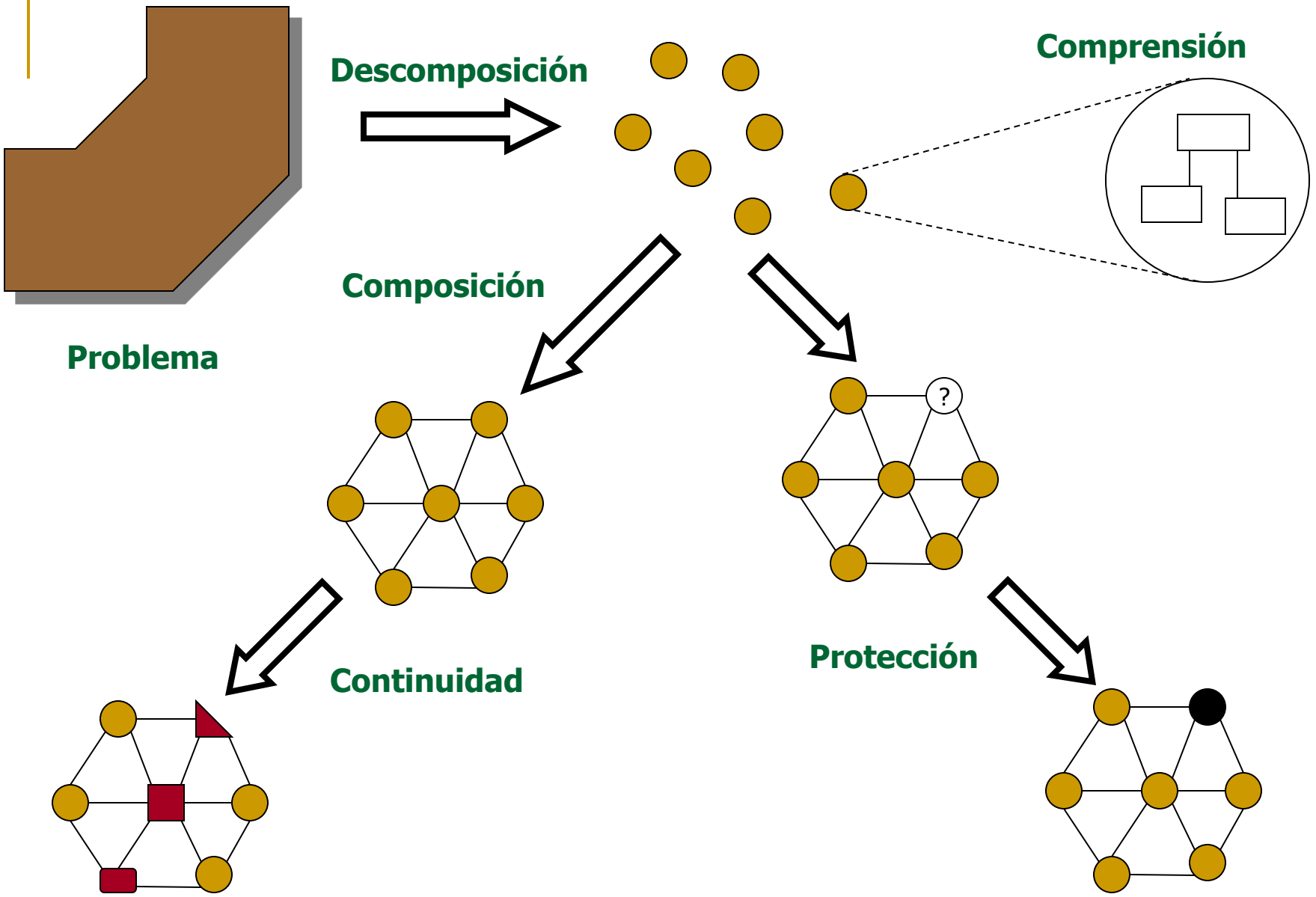
Define y refuerza las restricciones de acceso

Modularidad

- Divide el software en componentes identificables y tratables por separado.
- Módulo: Componente bien definido de un sistema de software. Autónomo, auto-contenido.
- Sistema modular = Σ módulos
- Beneficios
  - Facilita los factores de calidad del software
  - Calidad en los diseños de software

Modularidad

- Criterios de Meyer para evaluar la Modularidad
  - Descomposición
    - ¿Los componentes grandes están descompuestos en componentes pequeños?
  - Comprensión
    - ¿Los componentes son comprensibles individualmente?
  - Composición
    - ¿Los componentes grandes se componen de componentes pequeños?
  - Continuidad
    - ¿Pequeños cambios en la especificación afectan un número limitado y localizado de componentes?
  - Protección
    - ¿Están los efectos de las anomalías de ejecución confinados a un número pequeño de componentes relacionados?

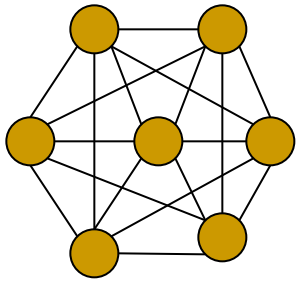

Descomposición
Comprensión
Problema
Composición
?
Continuidad
Protección

Reglas de Meyer para la modularidad

- Correspondencia directa (direct mapping)
- Pocos interfaces
- Interfaces pequeños
- Interfaces explícitos
- Ocultamiento de la información.

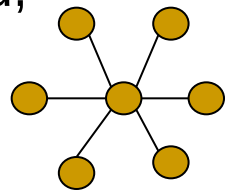
Reglas de Meyer para la modularidad

- Correspondencia directa (direct mapping)
  - Debe existir una relación consistente entre el modelo del problema y la estructura de la solución.
  - Mantener la estructura de la solución compatible con la estructura del dominio del problema modelado
  - Afecta a la continuidad y a la descomposición:
- Interfaces pequeños
  - Si dos componentes se comunican, deben intercambiar la menor información posible.
  - Afecta a la Continuidad y la Protección.

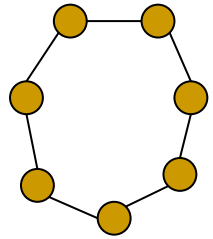


Reglas de Meyer para la modularidad

- Interfaces explícitos
  - Cuando dos componentes se comunican, esto debe ser evidente en la especificación de al menos uno de ellos.
  - Afecta a la Descomposición, Composición, Continuidad, Comprensión.



  - Pocos interfaces
  - Cada componente debe comunicarse con el menor número posible de componentes.
  - Afecta a la Continuidad, Protección, Composición, Comprensión.



Diseño modular efectivo

- Independencia funcional
  - Modularidad + Abstracción + Ocultación de la información.

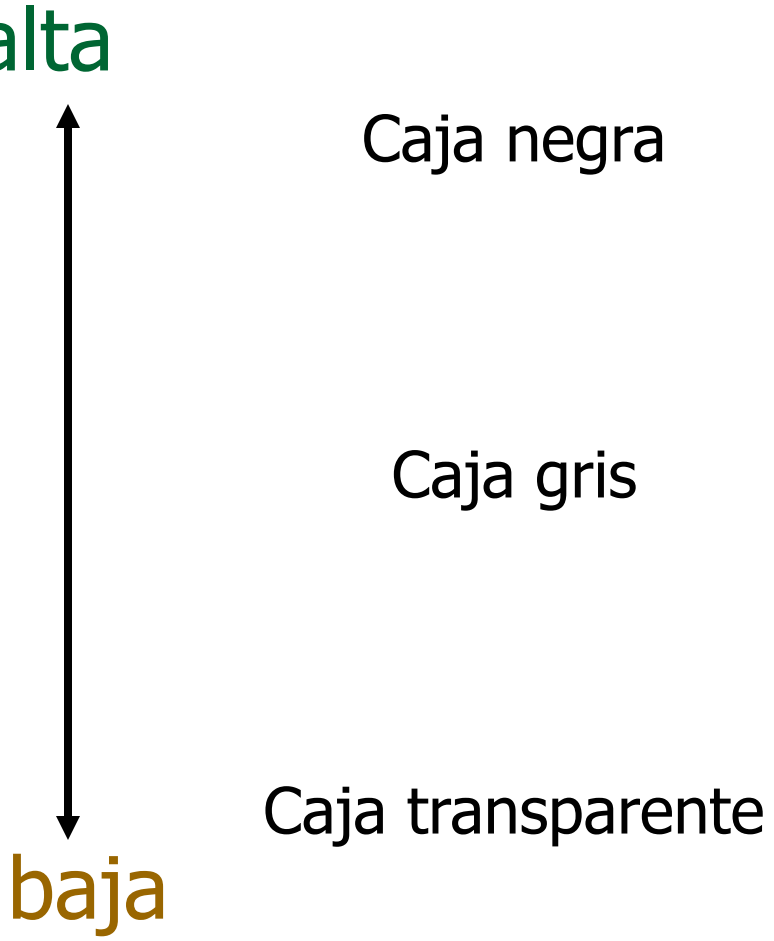
- Cómo lograrla?
  - Desarrollando módulos con una función “determinante”
  - “Aversión” a una interacción excesiva con otros módulos.
- Ventajas
  - Módulos más fáciles de desarrollar
  - Módulos más fáciles de mantener y probar
- La independencia se mide mediante dos criterios cualitativos
  - Cohesión
  - Acoplamiento.

Cohesión

- Un subsistema o módulo tiene un alto grado de cohesión si mantiene “unidas” cosas que están relacionadas entre ellas y mantiene fuera el resto.
- Un módulo cohesivo lleva a cabo una sola tarea dentro de un procedimiento software.
- Objetivo:
  - Diseñar servicios robustos y altamente cohesionados cuyos elementos estén fuerte y genuinamente relacionados entre si.
- Ventajas:
  - Favorece la comprensión y el cambio de los sistemas.

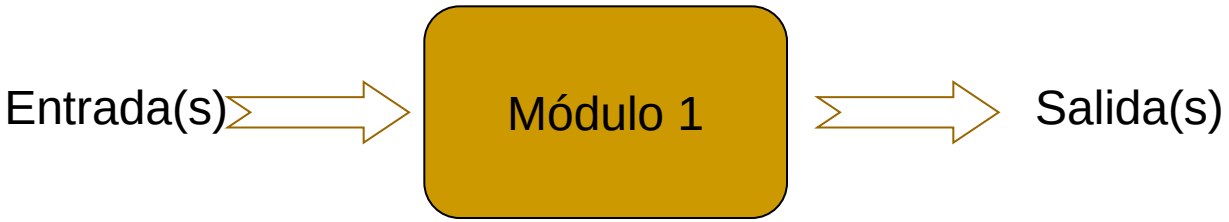
Niveles de Cohesión

1. Funcional
2. Secuencial
3. Comunicación
4. Procedimental
5. Temporal
6. Lógica
7. Coincidencia



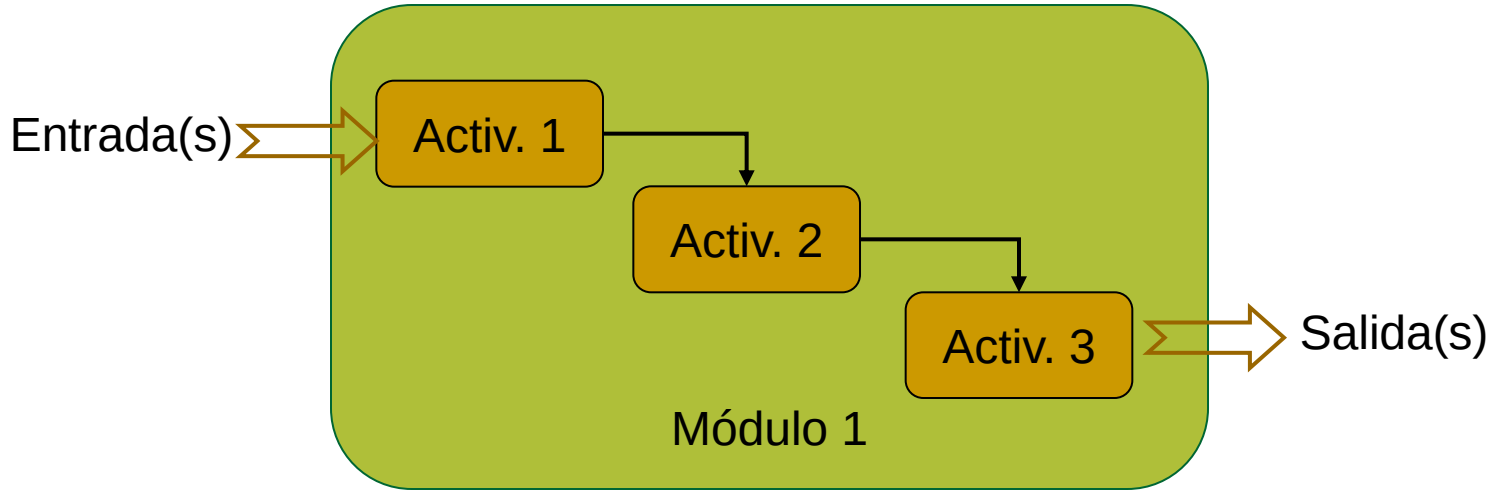
Cohesión Funcional

- Cada módulo realiza operaciones simples y sus acciones no tienen efectos colaterales.
- Ventajas:
  - Software fácil de comprender, reutilizable, fácilidad para reemplazar algún elemento.
- Los elementos contribuyen a la ejecución de una tarea relacionada con el problema.



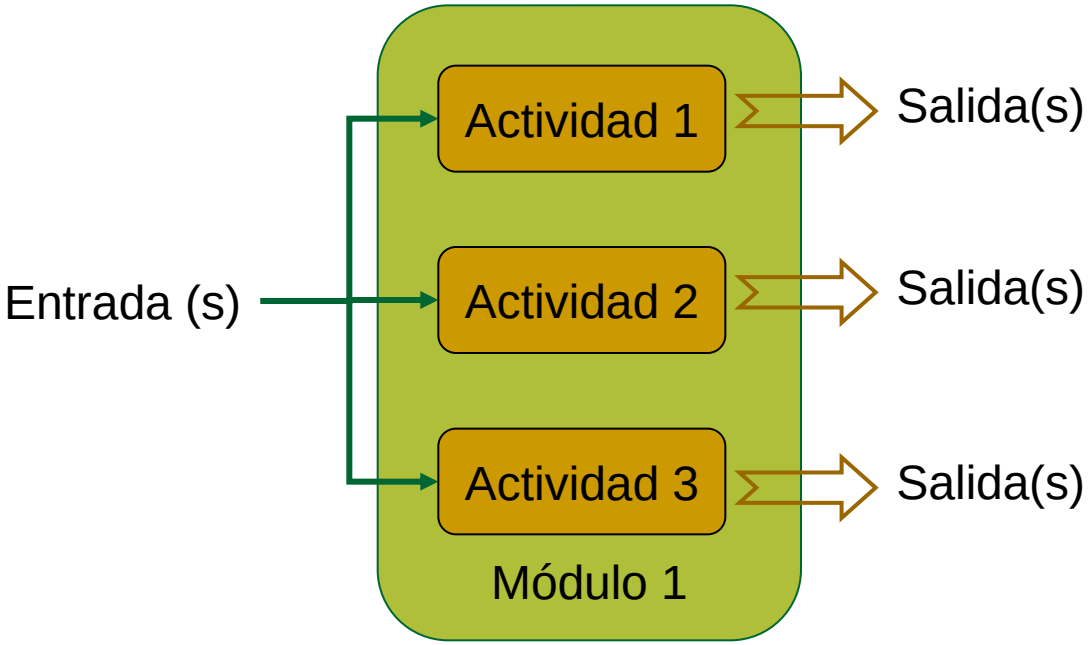
Cohesión Secuencial

- Agrupa procedimientos en los que uno proporciona la entrada al siguiente.
- Características:
  - Puede no ser bueno de cara a la reutilización.
  - Puede contener actividades que no se suelen utilizar juntas.



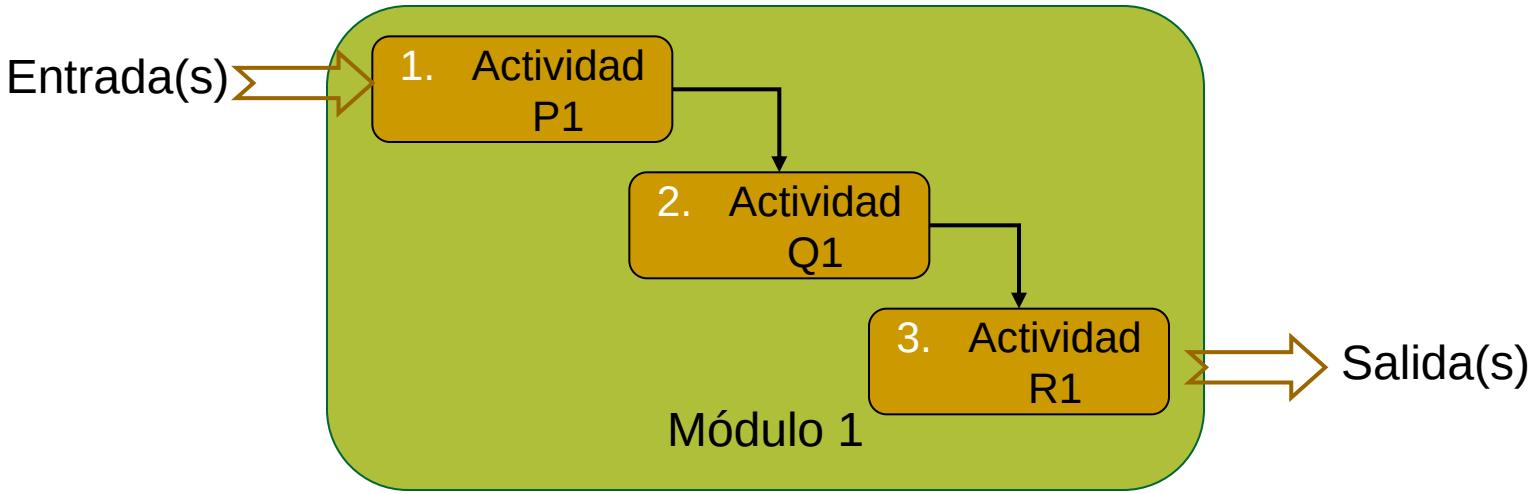
Cohesión Comunicación.

- El criterio para unir las actividades es si acceden o manipulan los mismos datos.
- El orden, a diferencia de la cohesión secuencial, no es importante.



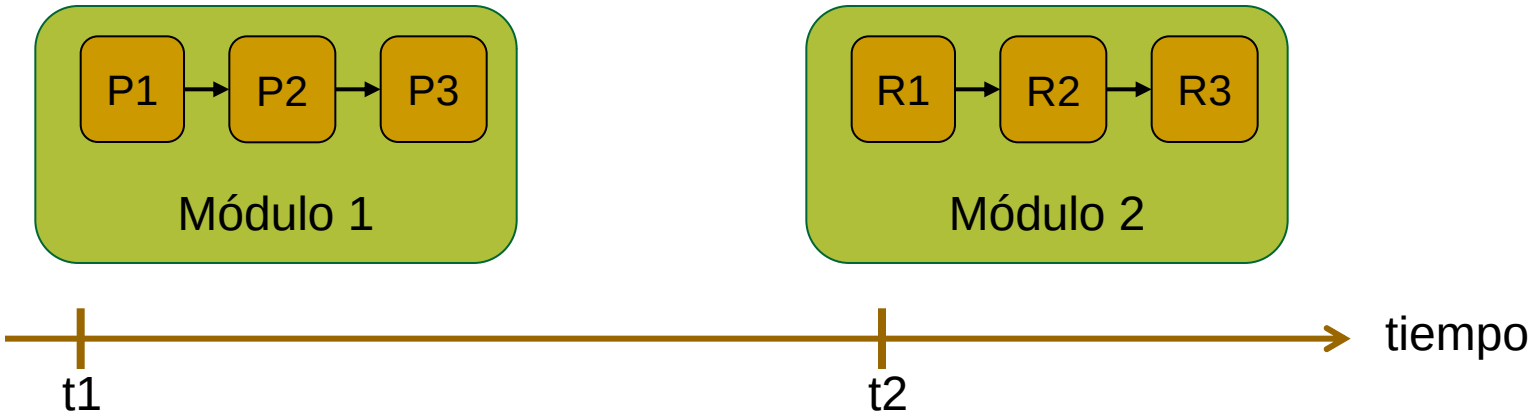
Cohesión Procedimental

- Composición de partes de funcionalidad que se organizan secuencialmente pero que, por otra parte, tienen poca relación entre si.
- Mal mantenimiento.



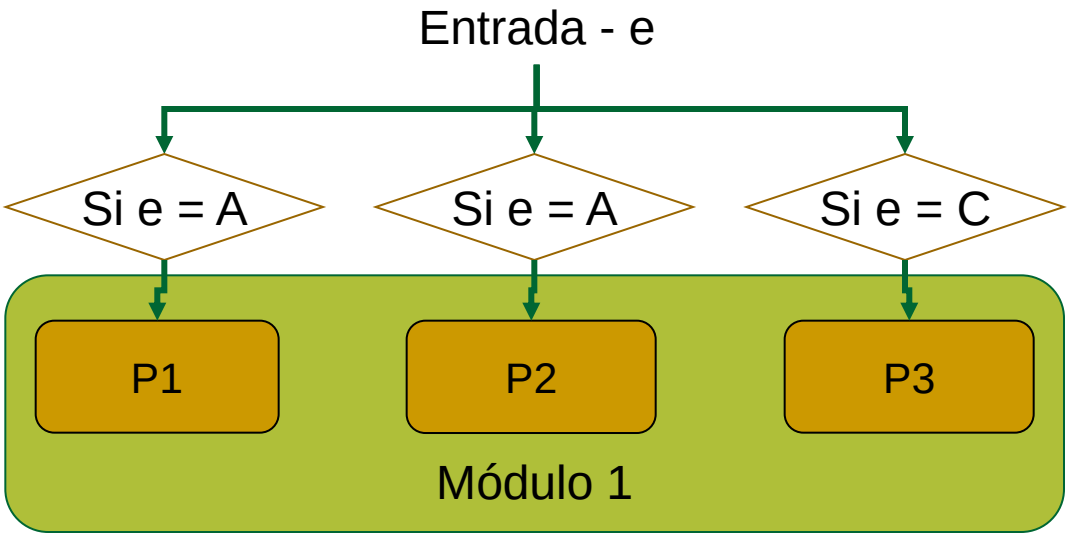
Cohesión Temporal

- Las operaciones que se realizan durante la misma fase de la ejecución del programa se mantienen unidas.
- Mal mantenimiento.



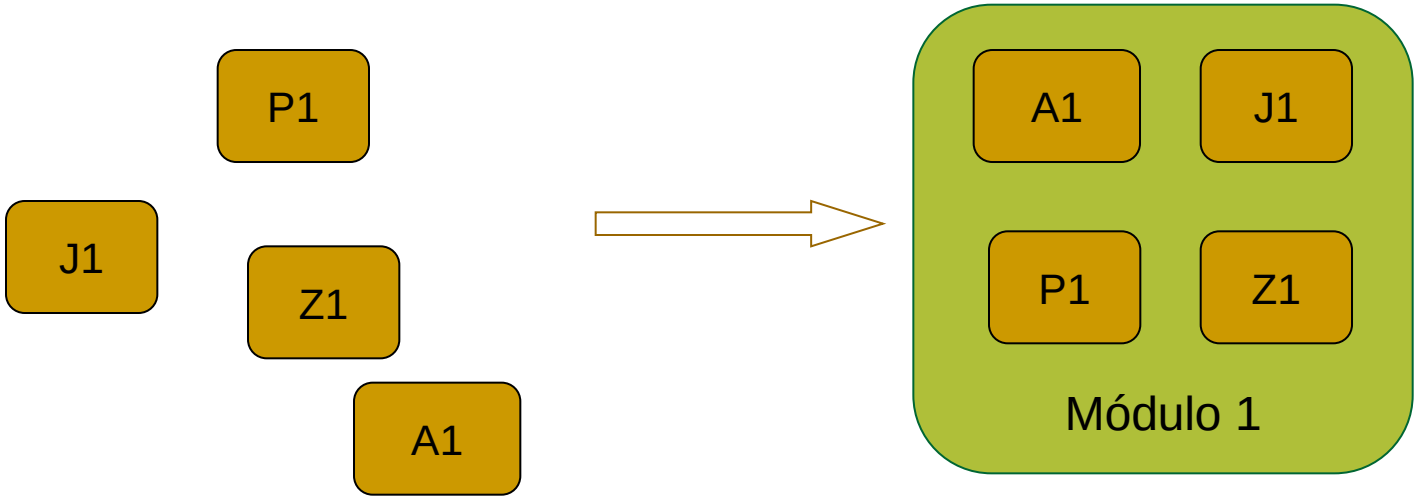
Cohesión Lógica

- Agrupa una serie de actividades en la misma categoría general.
- La actividad a ejecutar se determina normalmente por un parámetro de entrada.



Coincidencia o Cohesión por Utilidad

- Cuando se relacionan utilidades que no se pueden ser situadas de forma lógica en otras unidades cohesivas.



Acoplamiento.

- *"Acoplamiento es la medida de la fortaleza de la asociación establecida por una conexión entre módulos dentro de una estructura software"*
- Depende de la complejidad de interconexión entre los módulos, el punto donde se realiza una entrada o referencia a un módulo y los datos que se pasan a través del interfaz.
- Es una medida de la interconexión entre módulos dentro de una estructura del software.
- Un acoplamiento bajo indica un sistema bien dividido y puede conseguirse mediante la eliminación o reducción de relaciones innecesarias.

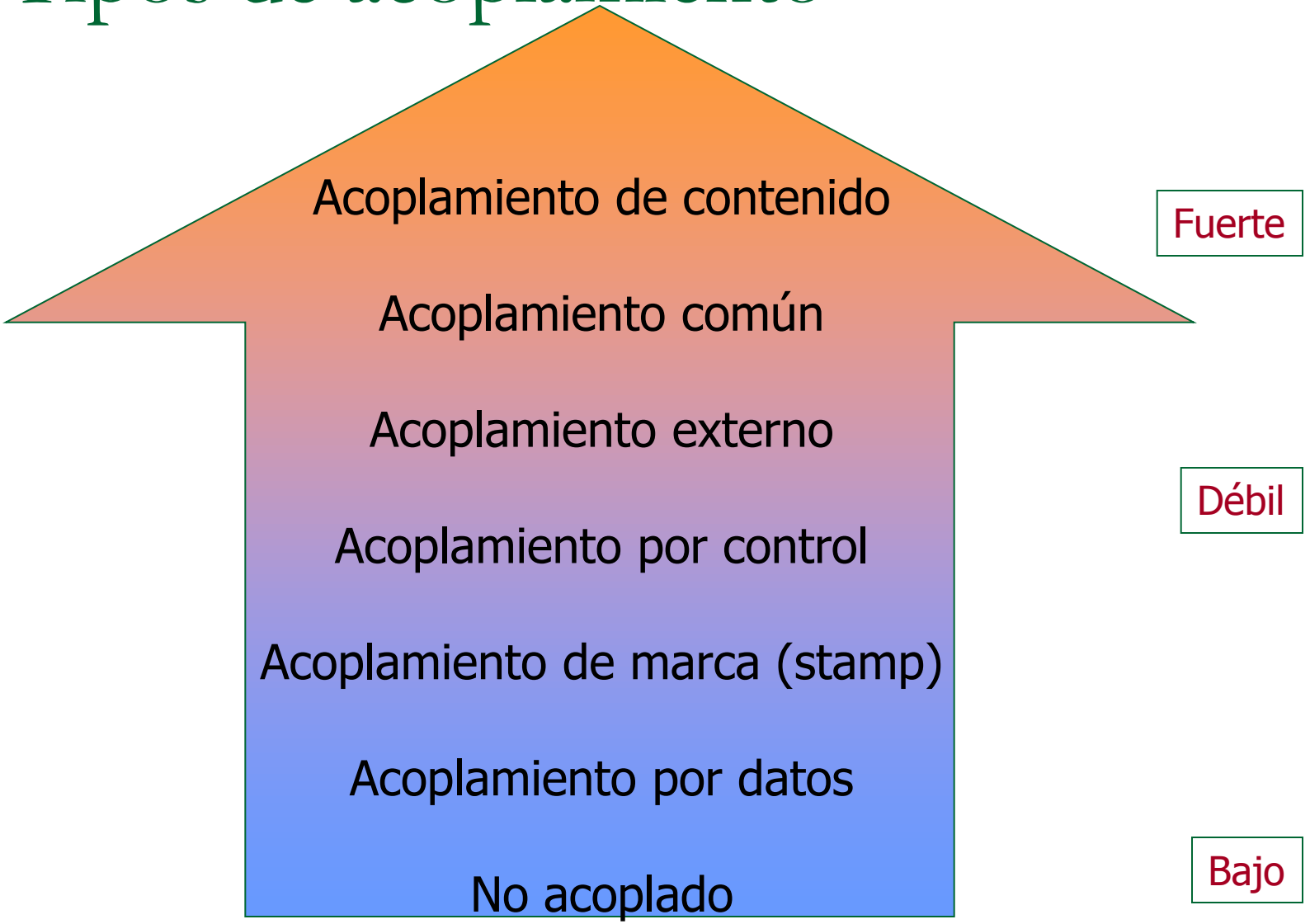
¿Qué buscamos con el Acoplamiento?

- Se produce una situación de acoplamiento cuando un elemento de un diseño depende de alguna forma de otro elemento del diseño.
- El objetivo es conseguir un acoplamiento lo más bajo posible. Conseguir que cada componente sea tan independiente como sea posible. (Acoplamiento bajo).
- Un bajo acoplamiento indica un sistema bien dividido y puede conseguirse mediante la eliminación o reducción de relaciones innecesarias.

Dependencia entre componentes

- El acoplamiento depende de varios factores:
  - Referencias hechas de un componente a otro (Invocaciones)
  - Cantidad de datos pasados de un componente a otro (parámetros)
  - El grado de control que un componente tiene sobre el otro (banderas de control)
  - El grado de complejidad de la interfaz entre los componentes (dependencia)

Tipos de acoplamiento



Niveles bajos de Acoplamiento

- Acoplamiento de datos
  - Solamente se pasan datos entre los módulos o componentes.
- Acoplamiento de marca (stamp coupling)
  - Se usa una estructura de datos para pasar información de un componente a otro, y es la estructura misma lo que se pasa a través de la interfaz.

Niveles moderados de Acoplamiento

- Acoplamiento de control
  - Se caracteriza por el paso de control entre módulos.
  - Dado P y Q. Si P invoca a Q:
    - Q contesta datos
    - Q contesta datos mas instrucción
- Acoplamiento externo
  - Los módulos están ligados a un entorno externo al software. Ej. La E/S acopla un módulo a dispositivos, formatos y protocolos de comunicación. Este acoplamiento es esencial pero deberá estar limitado a unos pocos módulos.

Niveles fuertes de acoplamiento

- Acoplamiento común:
  - Los módulos acceden a datos en un área de datos global.
  - Comparten una estructura de datos global.
    - No encapsulamiento y ni modularidad.
    - El diagnostico de problemas en estructuras con acoplamiento común es costoso en tiempo y difícil de realizar.
- Acoplamiento de contenido
  - Se da cuando un módulo hace uso de datos o de información de control mantenidos dentro de los límites de otro módulo. Este tipo de acoplamiento puede y deberá evitarse.

Síntomas de podredumbre del Diseño

- Rigidez
  - Software difícil de cambiar.
  - Gestores temen los cambios
- Fragilidad
  - Tendencia del software a “romperse” por muchos sitios cada vez que se produce un cambio.
  - El software “casca” en sitios inesperados.
- Inmovilidad
  - El software está tan “entrelazado” que es imposible reutilizarlo en otros proyectos.
- Viscosidad
  - Dificultad en utilizar métodos que preserven el diseño original cuando nos enfrentamos a un cambio.

Causas de podredumbre del Diseño

- Cambio de Requisitos
  - Es inevitable
  - Es necesario conseguir que los diseños sean resistentes a los cambios.
- Gestión de dependencias
  - Los cambios que provocan podredumbre son aquellos que introducen dependencias nuevas no planificadas.
  - Cohesión y acoplamiento
  - Deben controlarse
    - Creando “cortafuegos” para que no se propaguen.
- Existen numerosos principios y técnicas (patrones de diseño) para conseguir estos cortafuegos y gestionar las dependencias.

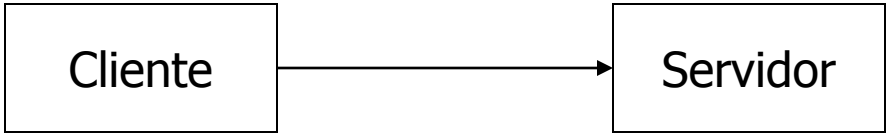
Principio Abierto-Cerrado (Open-Closed Principle. OCP)

> Las entidades software deben estar abiertas para su extensión pero cerradas para su modificación.
>
> *B. Meyer, 1988 / R. Martin, 1996*

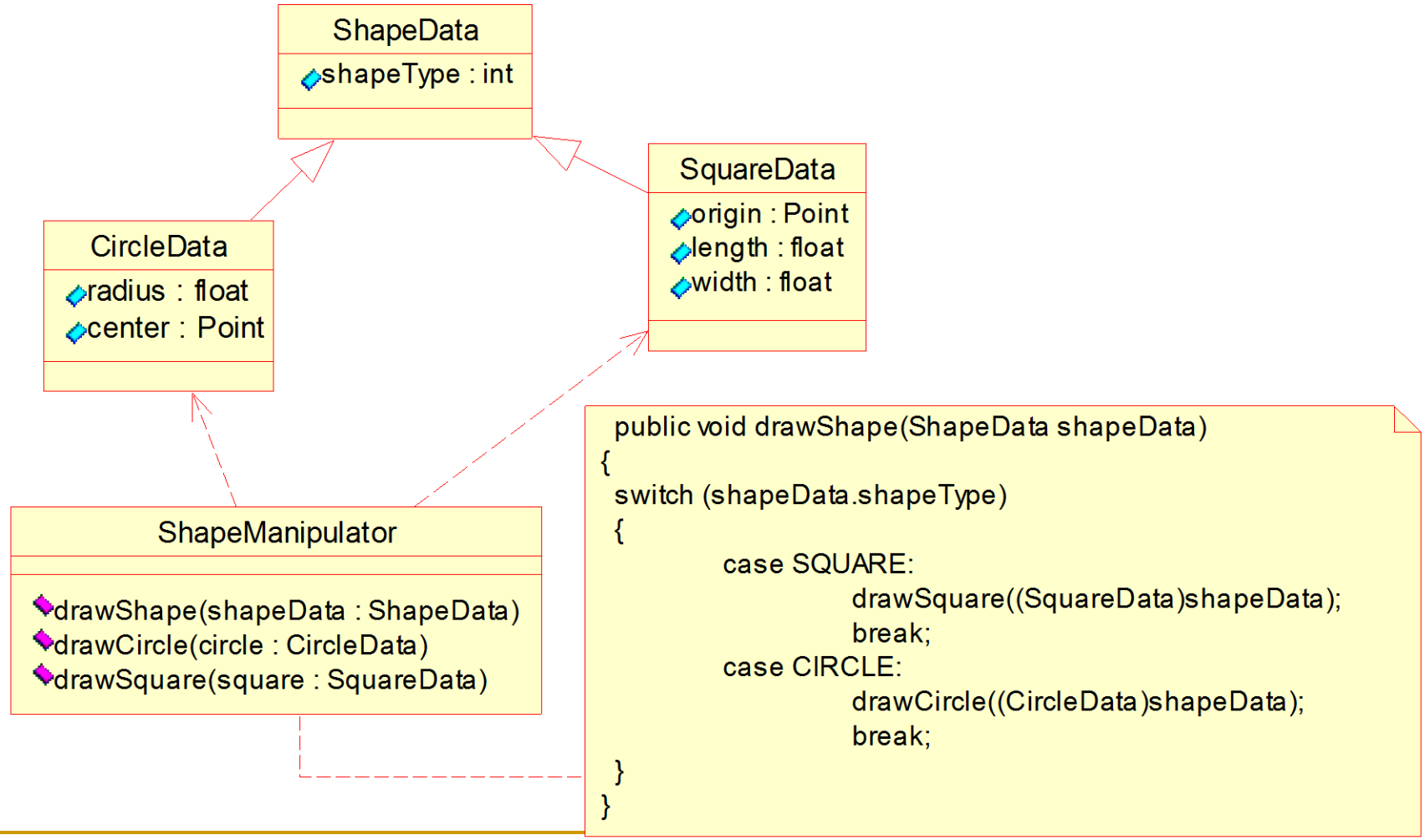
- Deben ser abiertos para extensión
  - El comportamiento del módulo puede ser extendido.
- Deben ser cerrados para modificación
  - El código fuente del módulo es inalterable.
- Los módulos han de ser escritos de forma que puedan extenderse sin que sea necesario modificarlos.
- ¿Cómo?
  - Abstracción y polimorfismo.

Principio Abierto Cerrado. (2)

- La clave del principio está en la abstracción.
- Se pueden crear abstracciones que sean fijas y que representen un número ilimitado de posibles comportamientos.
- Las abstracciones son un conjunto de clases base, y el grupo ilimitado de los posibles comportamientos viene representado por todas las posibles clases derivadas.
- Un módulo está cerrado para su modificación al depender de una abstracción que es fija. Pero el comportamiento del módulo puede ser extendido mediante la creación de clases derivadas.
- El siguiente diseño NO cumple el Principio Abierto/Cerrado.
  - Las clases cliente y servidor concretas. La clase cliente usa a la clase servidor.
  - Si se desea que un objeto cliente use un objeto servidor diferente, se debe cambiar la clase cliente para nombrar la nueva clase servidor.



Principio Abierto-Cerrado. Ejemplo

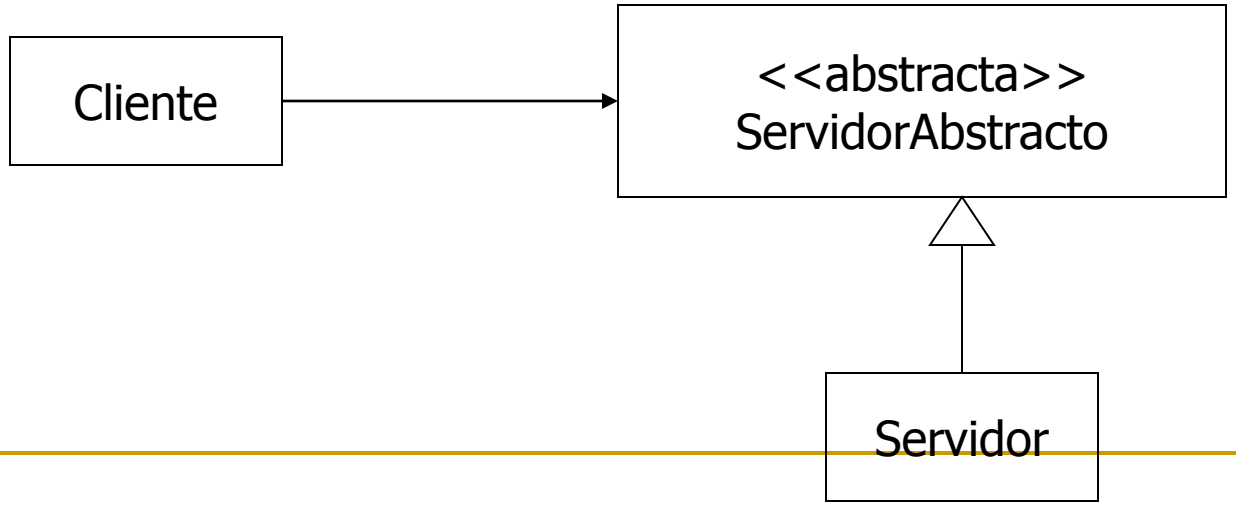


Principio Abierto-Cerrado. Ejemplo. Discusión

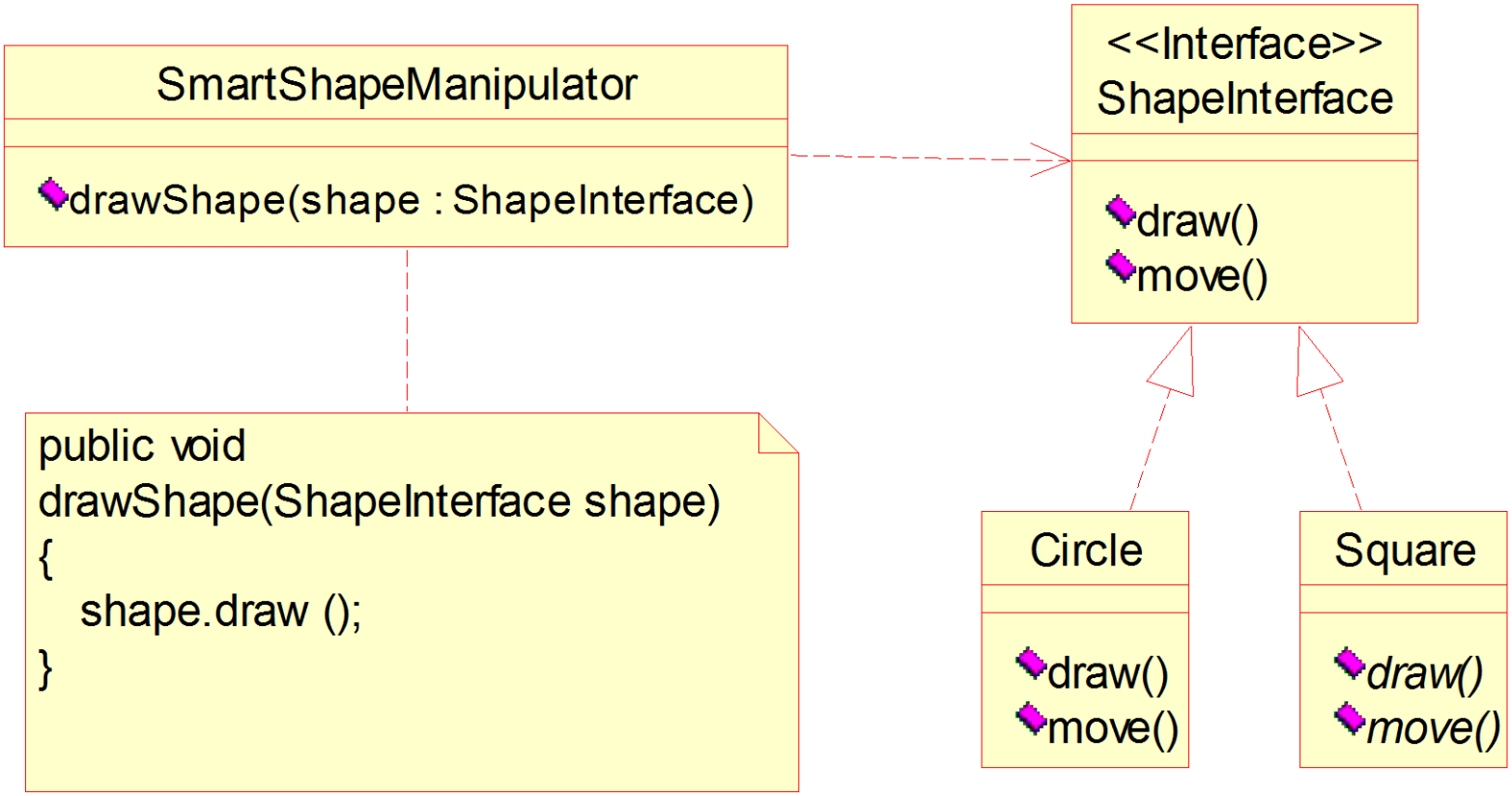
- Si se necesita crear una nueva figura, un Triángulo por ejemplo, necesitamos modificar la función: ‘drawShape()‘.
- En una aplicación compleja la sentencia switch/case se repite una y otra vez para cada tipo de operación que tenga que realizarse sobre una figura.
- Peor aún, cada módulo que contenga este tipo de sentencia switch/case retiene una dependencia sobre cada posible figura que haya que dibujar, por lo tanto, cuando se realice cualquier tipo de modificación sobre una de las figuras, los módulos necesitarán una nueva compilación y posiblemente una modificación.
- Si los módulos de una aplicación cumplen con el principio abierto/cerrado, las nuevas características pueden añadirse a la aplicación añadiendo nuevo código en lugar de cambiar el código en funcionamiento.

Principio Abierto-Cerrado. Ejemplo.

- Este diseño cumple el principio abierto/cerrado.
  - La clase ServidorAbstracto es una clase abstracta
  - La clase Cliente usa esa abstracción y, por lo tanto, los objetos de la clase Cliente utilizarán los objetos de las clases derivadas de la clase ServidorAbstracto.
  - Si se desea que los objetos de la clase Cliente utilicen diferentes clases Servidor, se deriva una nueva clase de la clase ServidorAbstracto y la clase Cliente permanece inalterada.



Principio Abierto-Cerrado. Ejemplo



Principio de Sustitución de Liskov (LSP).

- Los elementos clave del OCP son: Abstracción y Polimorfismo
  - Implementados mediante herencia
  - ¿Cómo medimos la calidad de la herencia?

La herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos supertipo también lo sea para los objetos subtipo.

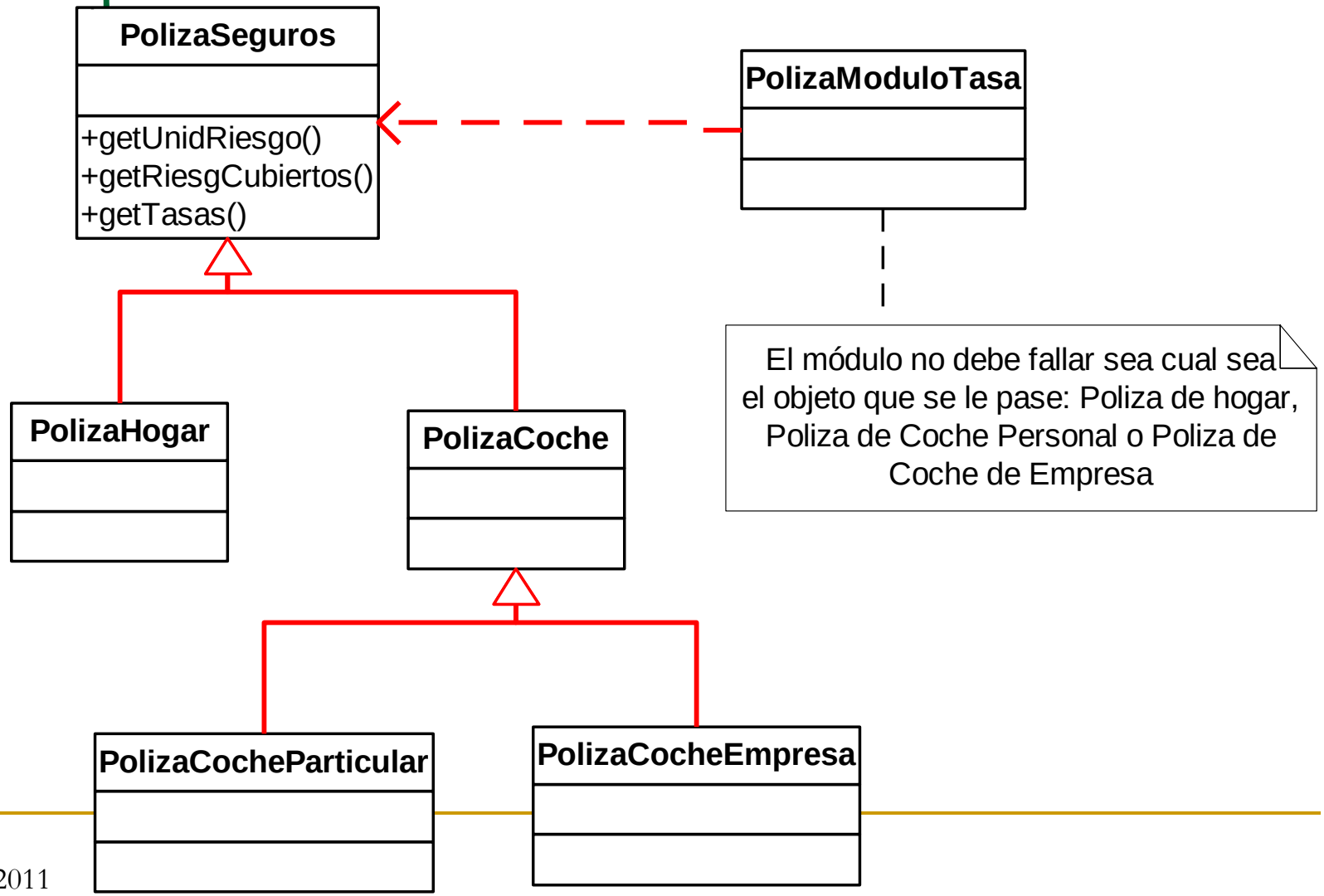
B. Liskov, 1987

Las clases derivadas deber ser utilizables a través de la interfaz de la clase base sin necesidad de que el usuario conozca la diferencia.

R. Martin, 1997

En resumen: Las subclases deben ser sustituibles por sus clases base

Principio de Sustitución de Liskov. Ejemplo



Principio de Sustitución de Liskov

- Un cliente de la clase base debe seguir funcionando adecuadamente si una clase derivada de esa clase base se le pasa a dicho cliente.
  - En otras palabras: Si alguna función toma un argumente del tipo Póliza, debería ser legal pasarle una instancia de PolizaCochePersonal ya que deriva directa o indirectamente de Póliza.
- El principio de Liskov establece que la relación IS_A en DOO se refiere al comportamiento externo público, que es comportamiento del que dependen los clientes.
  - Dilema círculo/elipse.
- Las violaciones del principio de Liskov son violaciones enmascaradas del principio abierto/cerrado.

Principio de Inversión de la Dependencia (DIP)

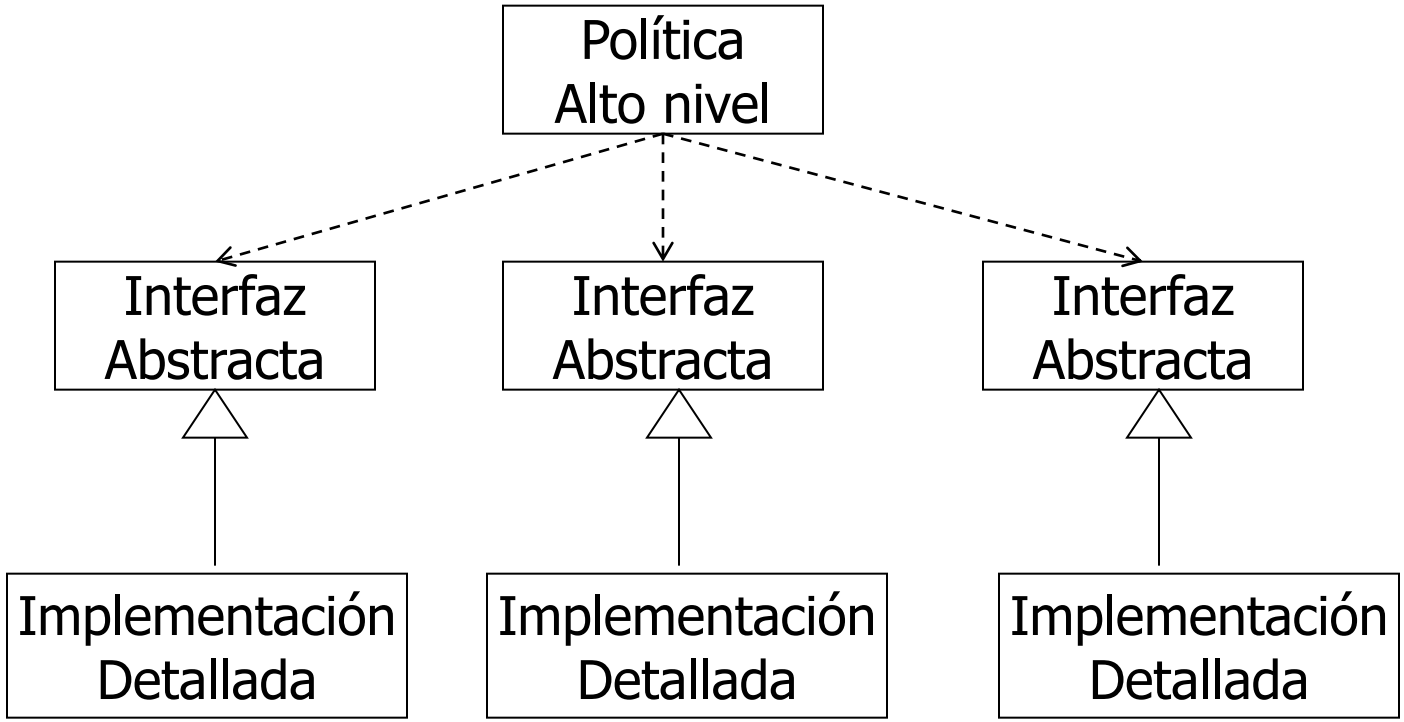
A. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
B. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

R. Martin, 1996

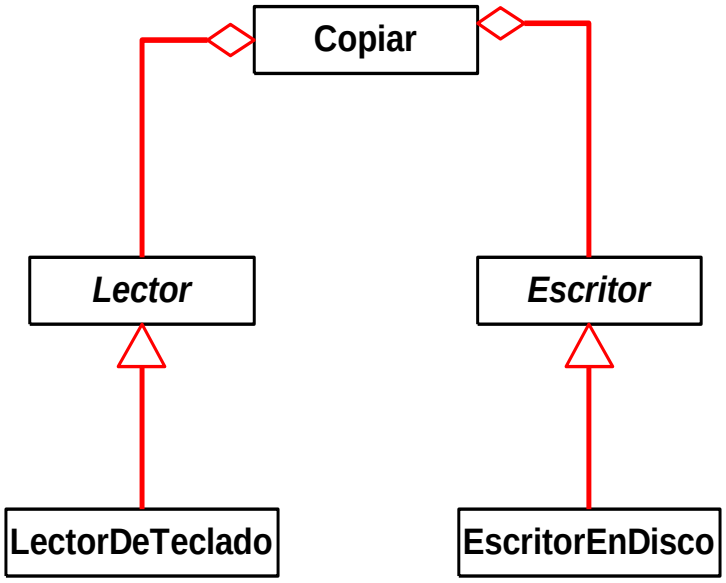
- El OCP establece el objetivo, el DIP establece el mecanismo.
- Indica la dirección que tienen que tomar todas las dependencias en un diseño orientado al objeto.
- La Inversión de Dependencia es la estrategia de depender de interfaces o funciones y clases abstractas, en lugar de depender de funciones y clases concretas.
- Cada dependencia en el diseño deberá tener como destino un interfaz o una clase abstracta. Ninguna dependencia debería tener como destino una clase concreta.

Principio de Inversión de Dependencia (DIP)

Estructura de las dependencias en una arquitectura Orientada al Objeto



Principio de Inversión de Dependencia (DIP). Ejemplo



- La clase Copiar obtiene un carácter del Lector y se la manda al Escritor independientemente de los módulos de bajo nivel.
- La clase Copiar depende de abstracciones y los Lectores y Escritores especializados dependen de las mismas abstracciones.
- Se puede reutilizar Copiar independientemente de los dispositivos físicos.
- Se pueden añadir nuevos tipos de *lectores* y *escritores* sin que la clase copiar dependa en absoluto de ellos.

Principio de Inversión de Dependencia. Consideraciones

- La motivación que subyace en el DIP es prevenir dependencias de módulos volátiles.
- Normalmente, las cosas concretas cambian con más frecuencia que las cosas abstractas.
- Las abstracciones son “puntos de enganche” que representan los “sitios” sobre los que el diseño puede ser aplicado o extendido, sin que se modifiquen (OCP)
- El DIP está relacionado con la heurística de DOO: “Diseñe interfaces, no implementaciones”

Principio de Inversión de Dependencia. Niveles.

"... todas las arquitecturas orientadas al objeto bien estructuradas tienen niveles claramente definidos, donde cada nivel ofrece algún conjunto de servicios coherentes a través de una interfaz bien definida y controlada".

G. Booch, 1996

- Heurística: Evitar dependencias transitivas.
  - Evitar estructuras en las que los niveles altos dependan de abstracciones de los niveles bajos.
- Solución
  - Utilizar herencia y clases abstractas ancestro para eliminar de forma efectiva las dependencias transitivas.

Principios DOO y Patrones

- Cuando se siguen los principios de diseño expuestos se descubre que las estructuras se repiten una y otra vez.
  - Estas estructuras repetitivas es lo que se conoce como patrón de diseño.

- Por otra parte, los patrones son soluciones probadas a los problemas de diseño más comunes con los que se enfrentan los diseñadores en orientación al objeto.
  - Gestión de dependencias
  - Diseño de sistemas que puedan evolucionar a medida que los requisitos cambien.
  - Maximizar la reutilización de diseños.
  - ....

Principios DOO y Patrones

Los patrones de diseño ayudan a evitar rediseños al asegurar que un sistema pueda cambiar de forma concreta

- Causas comunes de rediseño.
  1. Crear un objeto indicando la clase.
  2. Dependencia de operaciones específicas
  3. Dependencia de plataformas hardware o software
  4. Dependencia sobre representación de objetos.
  5. Dependencias de algoritmos
  6. Acoplamiento fuerte entre clases
  7. Extender funcionalidad mediante subclases
  8. Incapacidad de cambiar clases convenientemente

- Patrones
  1. Abstract factory, Method factory, Prototype
  2. Chain of Responsability, Command
  3. Abstract factory, Bridge
  4. Abstract factory, Bridge, Memento, Proxy,
  5. Builder, Iterator, Strategy, Template Method, Visitor
  6. Abstract factory, Bridge, Chain of Responsability, Command, Facade, Mediator, Observer
  7. Bridge, Chain of Responsability, Composite, Decorator, Observer, Strategy
  8. Adapter, Decorator, Visitor