Tema 5 (Parte II): Principios del Diseño de Paquetes (Componentes)

Evolución y Adaptación de Software

Carlos E. Cuesta, ETSII, URJC

Considerando el Diseño a Alto Nivel

- Las clases son un mecanismo (abstracción) útil pero insuficiente para organizar un diseño
- Organizaciones de granularidad superior a la clase:
  - Paquetes, a nivel de código (tiempo de diseño / compilación)
  - Componentes, a nivel de distribución (tiempo de ejecución)
  - En UML, resumimos ambas en el concepto de **paquetes**
- Objetivo:
  - Establecer la partición de las clases en una aplicación siguiendo un criterio y asignar estas particiones a paquetes.
- Consideraciones
  - ¿Cuál es el mejor criterio de partición?
  - ¿Qué principios guían el diseño de paquetes?
    - **Creación y dependencias entre paquetes**
  - Utilizar los principios que guían el diseño de paquetes
    - Creación, interrelaciones y utilización de paquetes.

Principios RCC + ASS (3C/3A)

- Son “los otros principios SOLID”
  - Mismo autor, Robert Cecil Martin (“Uncle Bob”)
  - A diferencia de los otros, él sí inventa varios de éstos
- No son tan conocidos como los 5 SOLID “famosos”
  - Se dan en contextos donde las aplicaciones software son grandes.
- Sus consecuencias son tanto o más importantes que la aplicación de los otros
  - Pueden desencadenar situaciones complejas
  - Un pequeño cambio puede provocar consecuencias a lo largo de muchos dominios complejos
  - Ese mismo cambio puede afectar a múltiples equipos de desarrollo
- Esta situación también se da en las relaciones entre dominios cuando estamos en una organización basada en DDD
  - Cuando distintos contextos comparten definiciones

Principios del Diseño de Componentes

Principios de Cohesión

- Principio de Equivalencia Reutilización/Revisión (REP)
- Principio de Reutilización Común (CRP)
- Principio de Cierre Común (CCP)

Principios de Acoplamiento

- Principio de Dependencias Acíclicas (ADP)
- Principio de Dependencias Estables (SDP)
- Principio de Abstracciones Estables (SAP)

Principio de Equivalencia Reutilización/Revisión
REP: *Reuse-Release Equivalence Principle*

> La granularidad de la reutilización debe ser la granularidad de la revisión
>
> *Robert C. Martin, 1996*

- Normalmente cada clase tiene un conjunto de clases con las que colabora
  - No es reutilizable por sí sola
  - Debe reutilizarse en conjunto con las clases con las que colabora
- Los desarrolladores que reutilizan código no desean tener que mantener los cambios de las clases que reutilizan
  - Se requiere un mecanismo de distribución de las nuevas revisiones
- Un elemento reutilizable (componente, clase, *cluster* de clases) no puede ser reutilizado a menos que sea gestionado por un sistema de distribución
  - Indirectamente: en este siglo esto se "lee" como control de versiones
- Por tanto: un criterio para agrupar clases en paquetes es la reutilización
  - Los paquetes son la unidad de distribución → también la unidad de reutilización

Principio de Reutilización Común
CRP: *Common Reuse Principle*

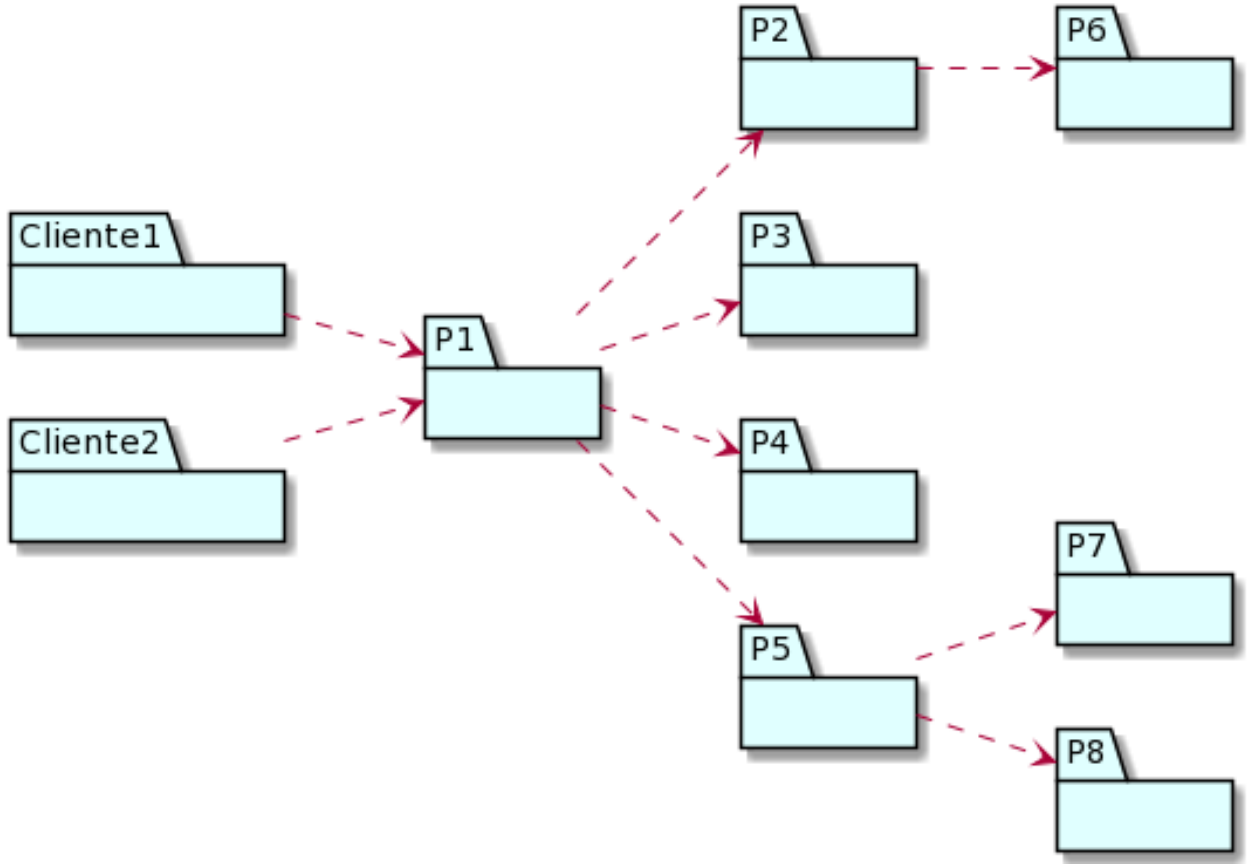
Todas las clases en un paquete deben ser reutilizadas juntas.
Si se reutiliza una de las clases del paquete, se reutilizan todas.

Robert C. Martin, 1996

- Una dependencia de un paquete es una dependencia de *todo* lo que está dentro del paquete
  - Cuando un paquete cambia, su número de revisión se incrementa
  - Todos los clientes de ese paquete han de verificar que funcionan con el nuevo paquete.
- Los paquetes de componentes reutilizables deben agruparse por la utilización esperada
  - Si no se va a reutilizar a la vez, no debería estar en el mismo paquete
- Las clases normalmente se reutilizan en grupos, en base a las colaboraciones entre las bibliotecas de clases

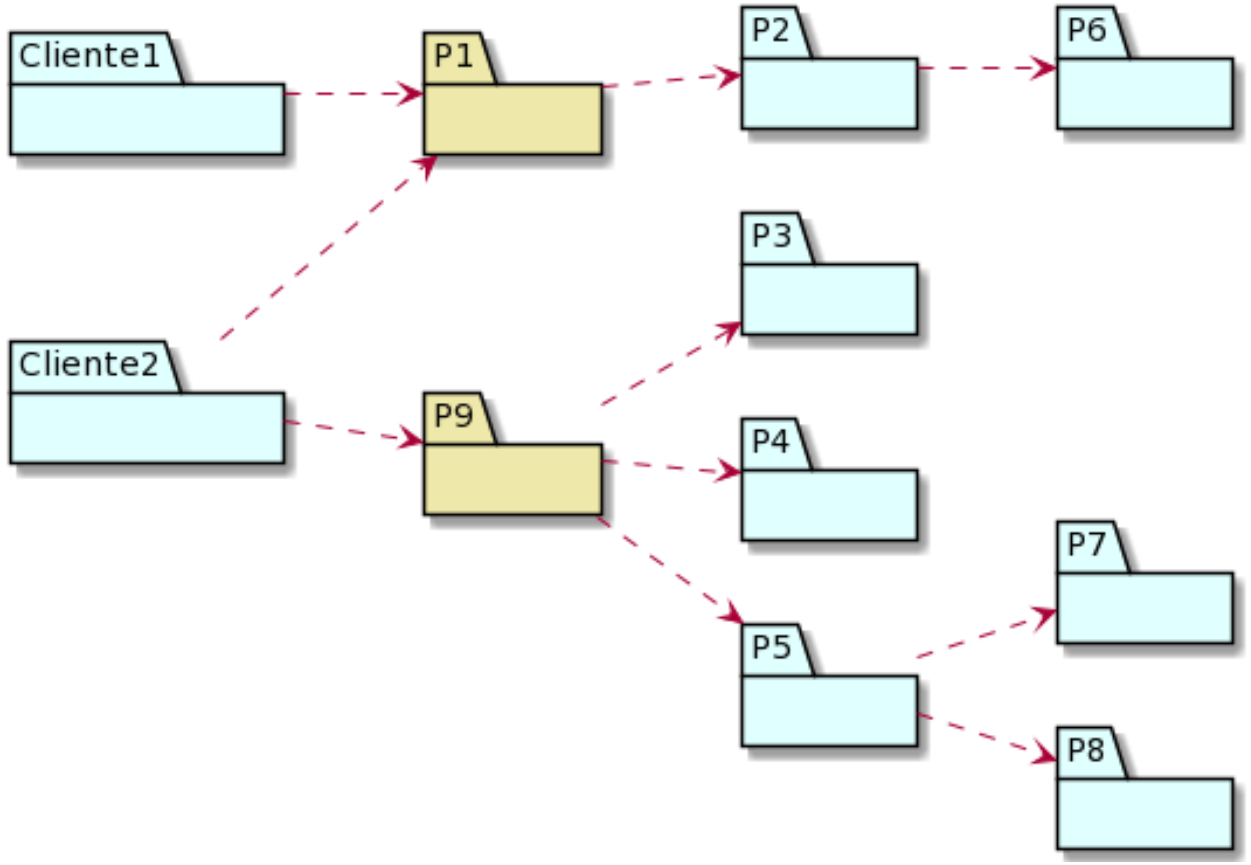
[CRP] Ejemplo

- Cuando un paquete concentra demasiadas funciones, conviene dividirlo



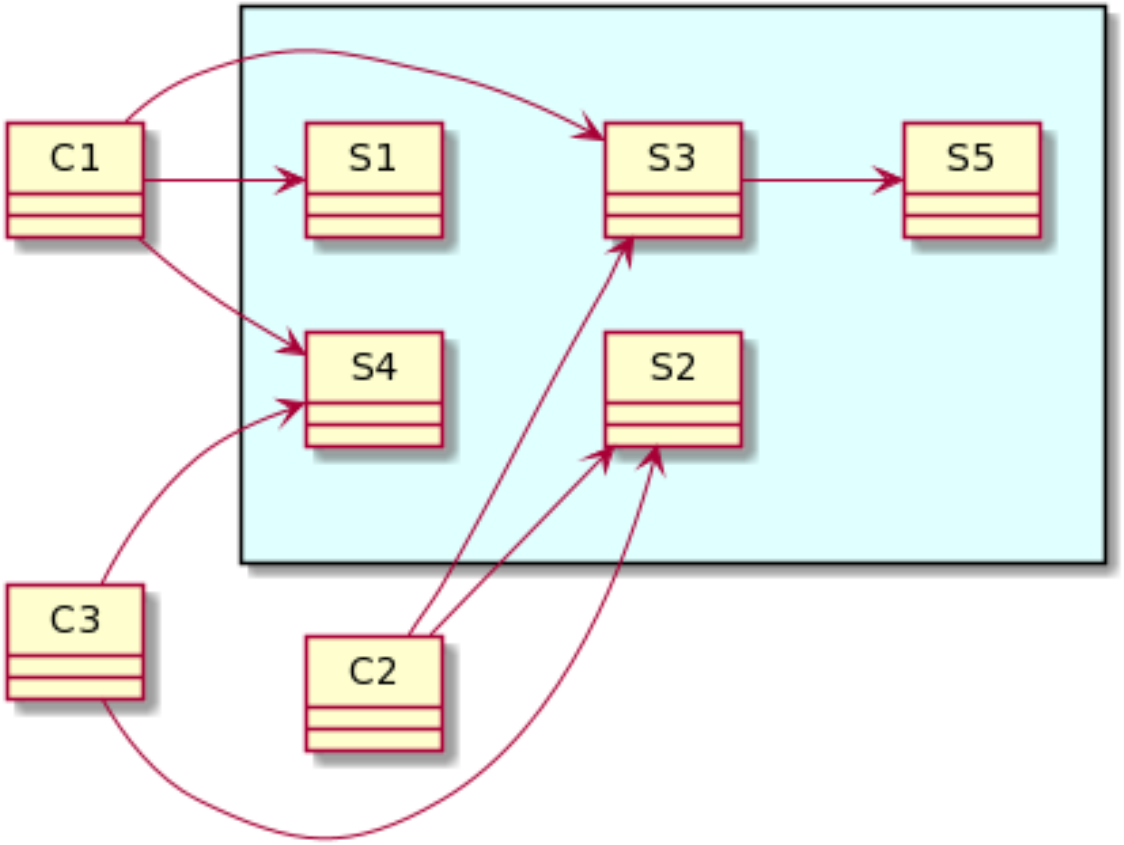
[CRP] Ejemplo

- En este caso, P1 se divide en dos: un nuevo P1 y un P9 adicional



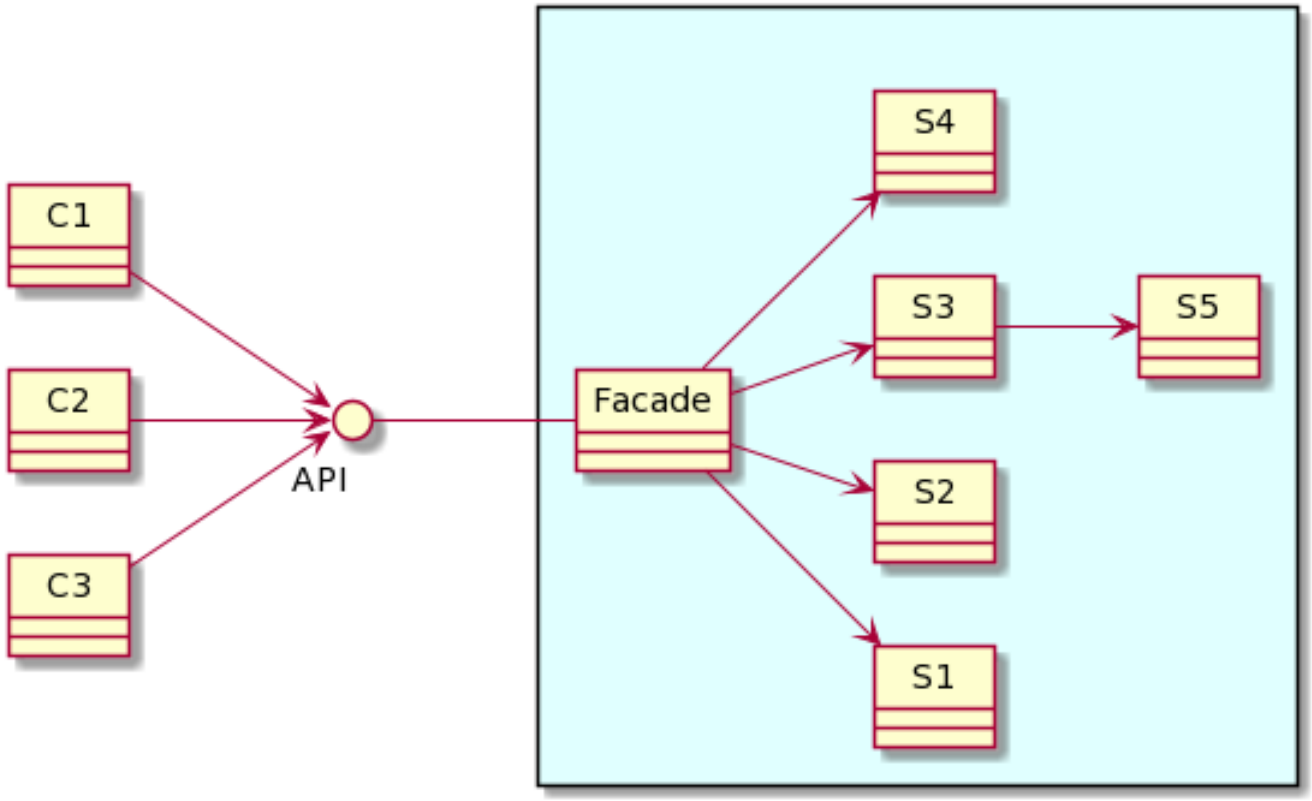
[CRP] Relación con el patrón Fachada

- Cuando el nivel de reutilización se convierte en un objetivo primordial, el patrón Fachada es fundamental
  - Supongamos que se parte de esta situación:



[CRP] Relación con el patrón Fachada

- Aplicando el Fachada se está aplicando el CRP, y el resultado es un diseño mucho más claro:



Principio de Cierre Común
CCP: *Common Closure Principle*

Las clases que cambian a la vez, deben estar juntas.
Classes that change together, belong together

Robert C. Martin, 1996

- Si una parte de la aplicación tiene que cambiar, lo mejor es que los cambios se centren en un único paquete
  - En lugar de dispersarse por el conjunto completo de paquetes
- Las clases en una agrupación deben tener un *cierre* común
  - Si una necesita ser cambiada, todas ellas pueden requerir cambios
  - Lo que afecta a una afecta a todas
- Las clases de un paquete han de ser cohesivas
- Dado un tipo particular de cambio, o se necesita tener que modificar *todas* las clases o *ninguna* en un componente/paquete

Principios de Cohesión (3C/RCC) Consideraciones

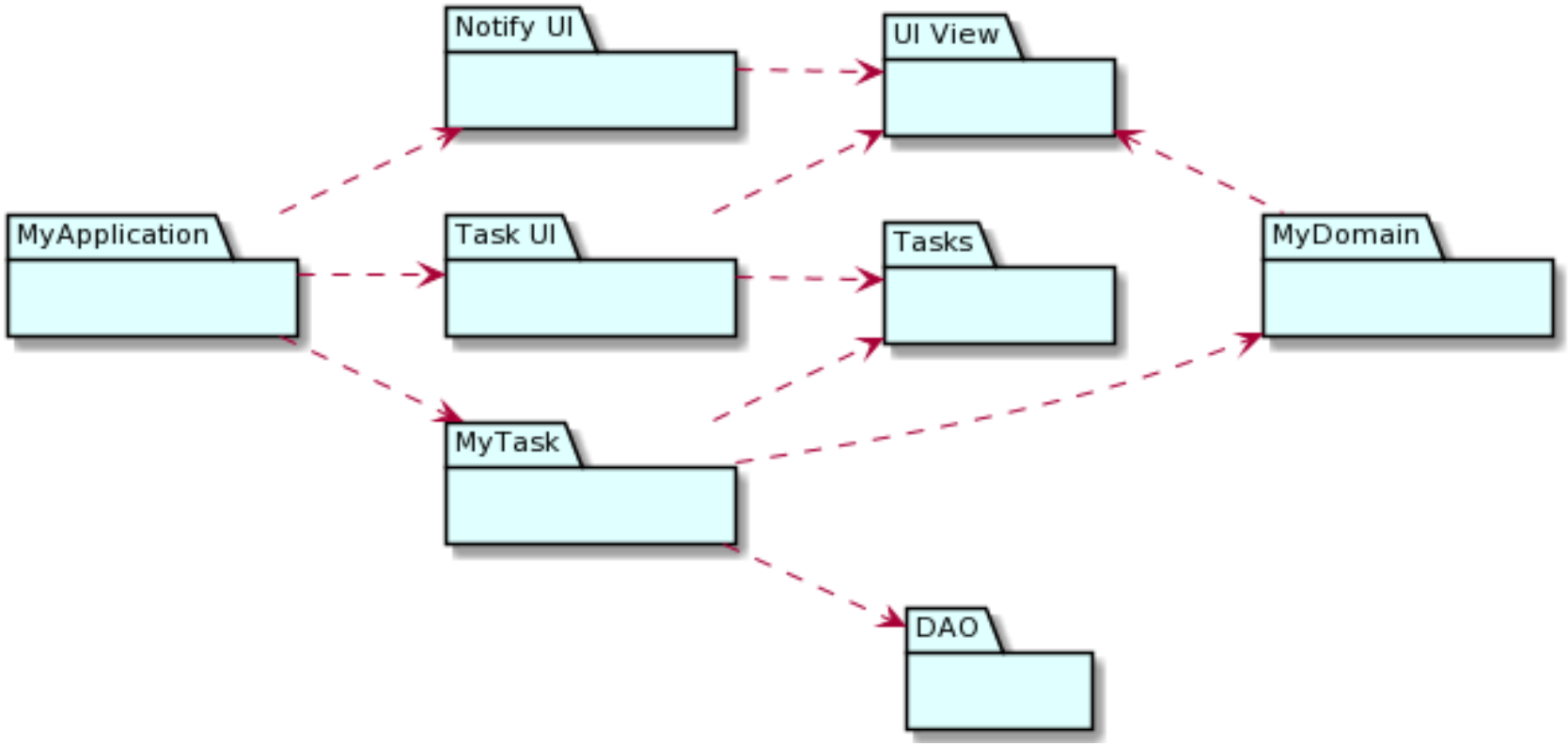
- Los tres principios REP, CRP y CCP no pueden ser satisfechos de forma simultánea
  - Los principios REP y CRP facilitan la vida a los reutilizadores
  - Mientras que CCP se la facilita al equipo de mantenimiento
- El CCP fuerza a que los paquetes sean tan grandes como sea posible
  - Después de todo, si todas las clases viven en un único paquete, solamente habrá que cambiar ese paquete
  - Por el contrario, el CRP intenta que los paquetes sean muy pequeños
- En las fases iniciales, cuando se establece la estructura del paquete, domina el principio CCP
  - Para facilitar el desarrollo y mantenimiento
- En las fases posteriores, *cuando la arquitectura se estabiliza*, se puede refactorizar la estructura del paquete
  - Para maximizar los principios REP y CRP de cara a los reutilizadores externos

Principio de Dependencias Acíclicas
ADP: *Acyclic Dependencies Principle*

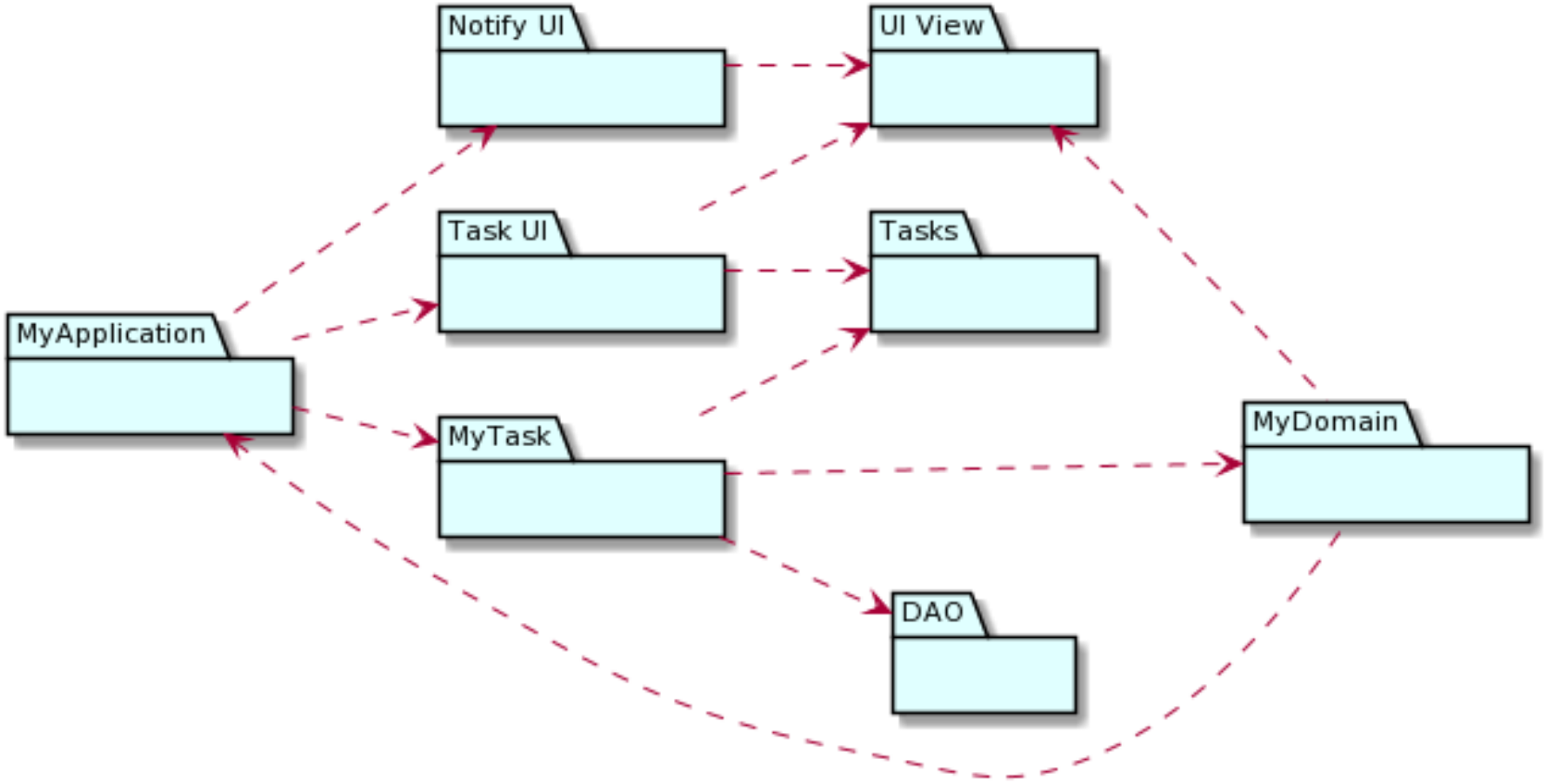
> Las dependencias entre paquetes no deben formar ciclos.
>
> *Robert C. Martin, 1996*

- Los paquetes son las unidades de trabajo
- Tendencia a que los desarrolladores trabajen sobre un único paquete, favorecida por los principios de cohesión de los paquetes
  - Consecuencia (una vez más) de la Ley de Conway
- El paquete es la unidad de revisión: es el que se libera
  - Como ya quedó establecido en el REP
  - Antes de liberar el paquete (unidad de revisión) ha de ser compilado y ensamblado con el resto de los paquetes de los que depende
- Una dependencia cíclica tiene graves consecuencias
  - Fuera de control puede establecer una lista de dependencias muy extensa
- Es necesario romper los ciclos

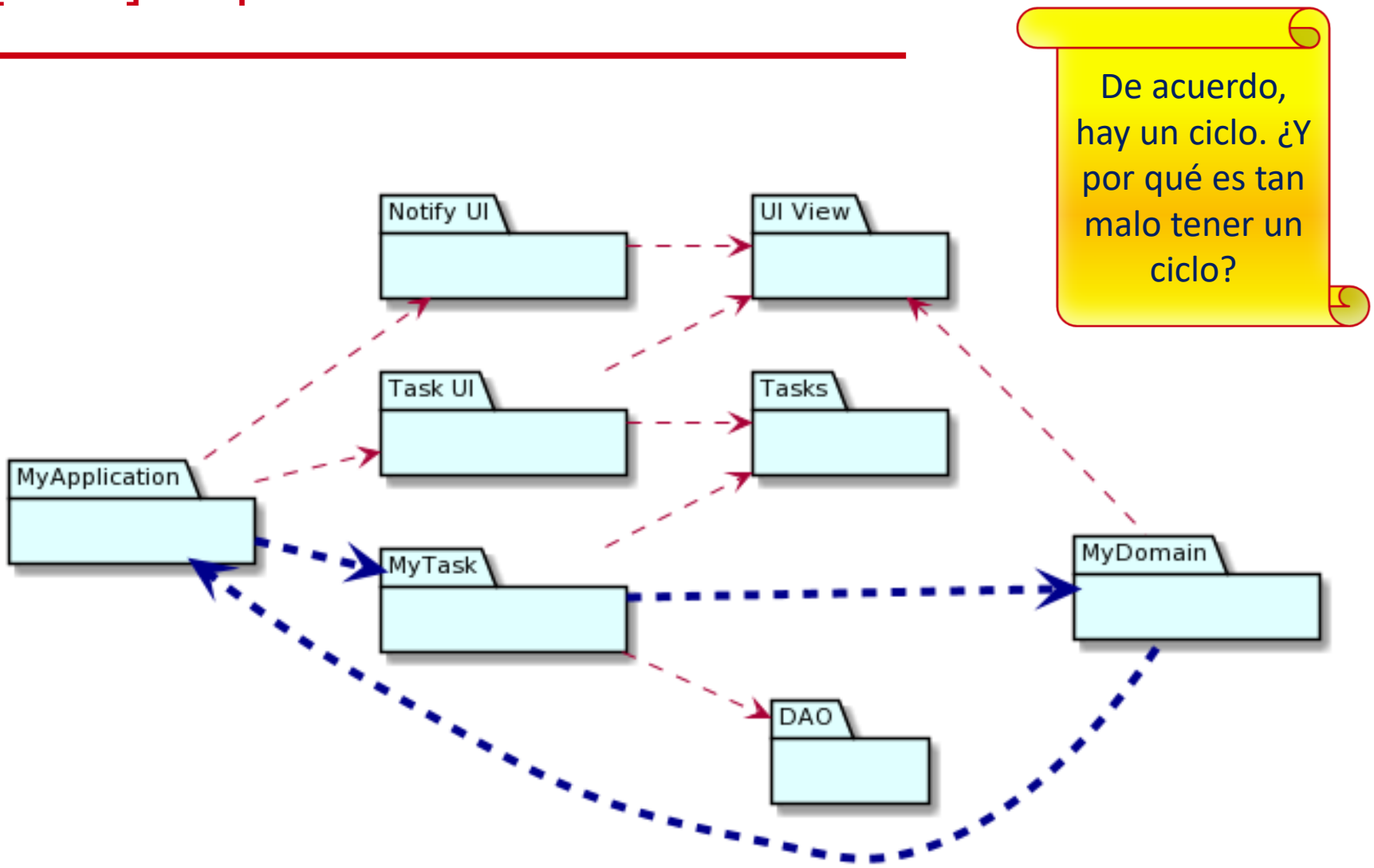
[ADP] Arquitectura sin Ciclos



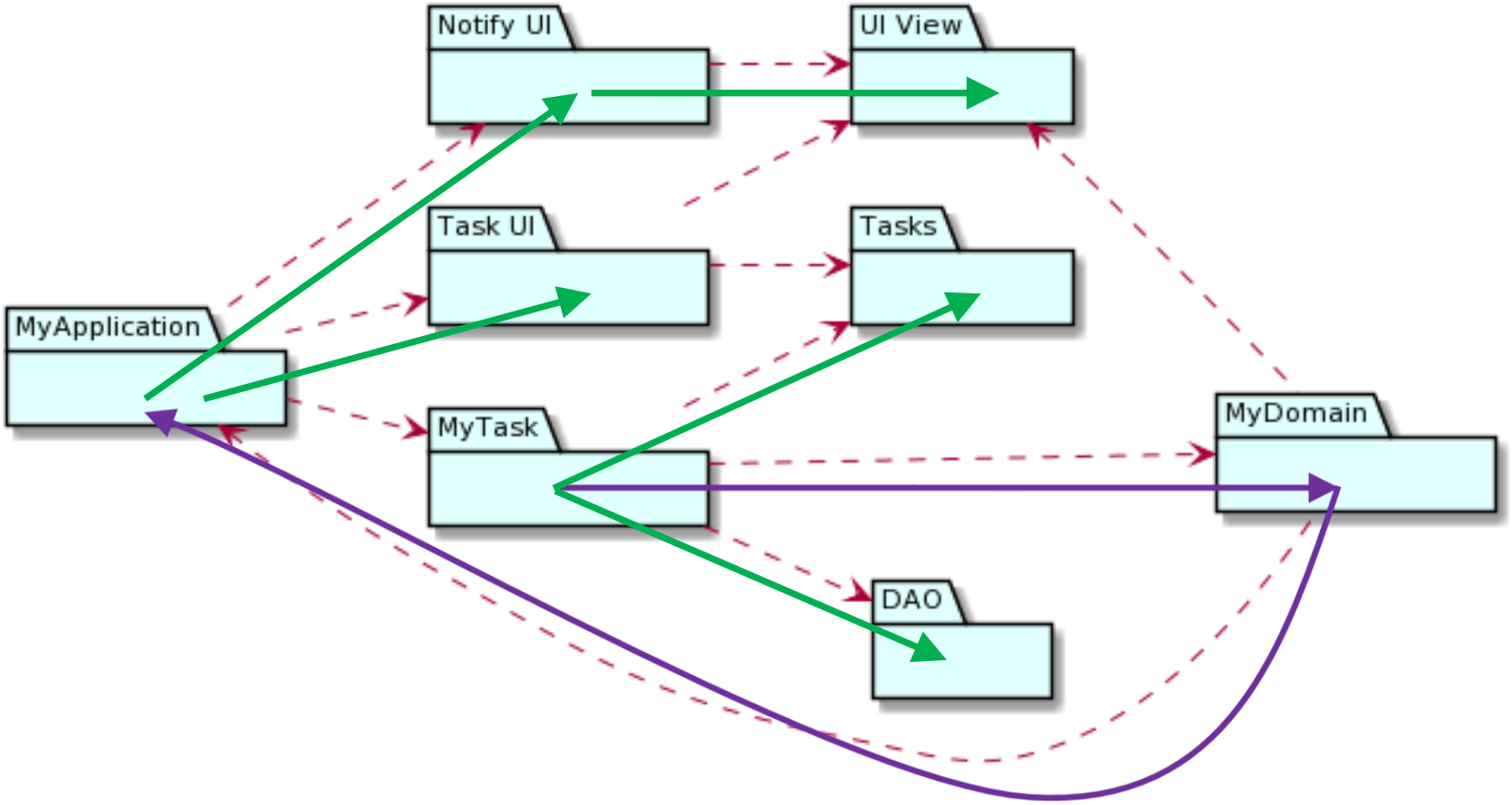
[ADP] Arquitectura Con Ciclos



[ADP] Arquitectura Con Ciclos



[ADP] Arquitectura Con Ciclos



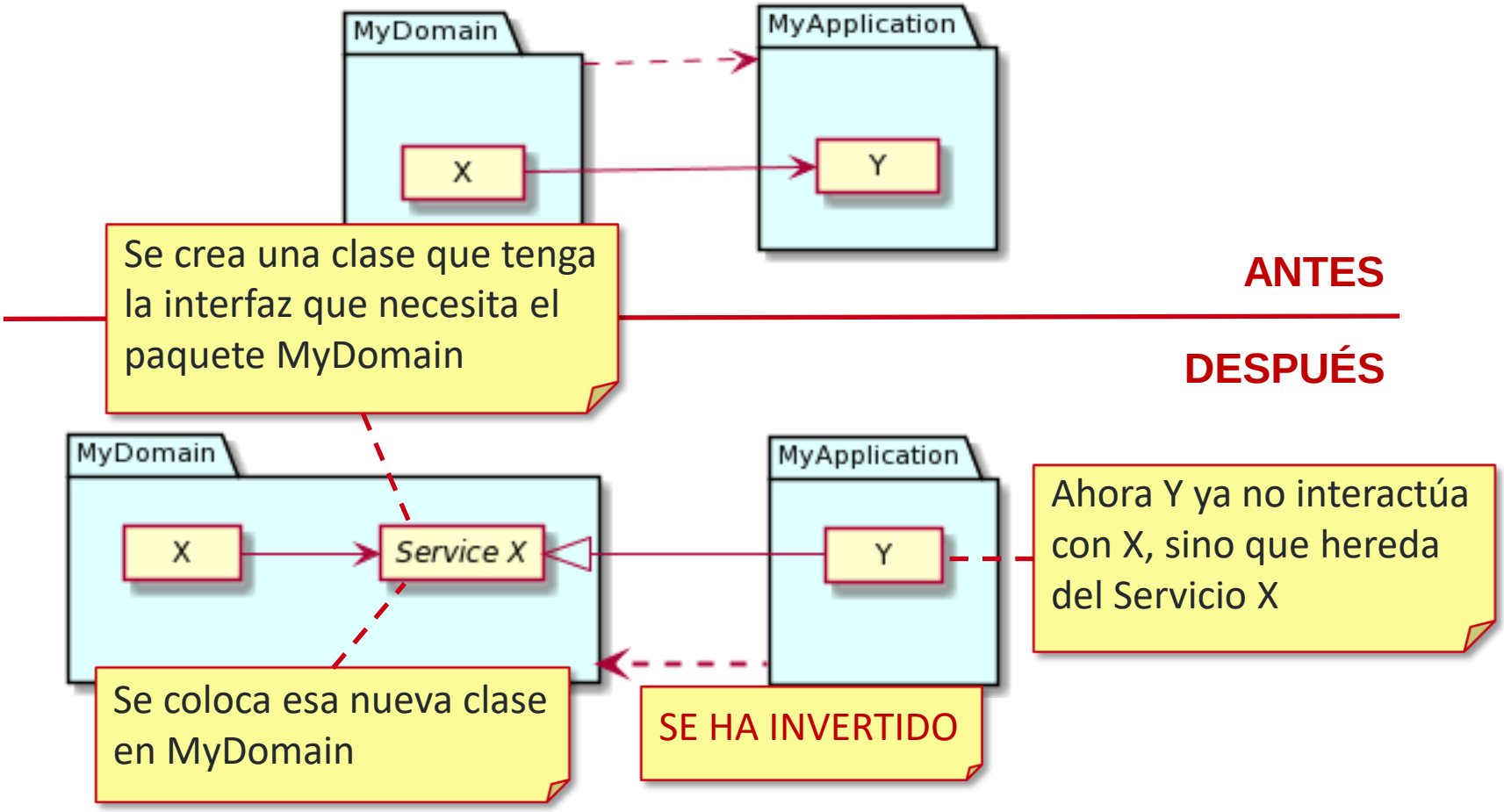
VORTICE

[ADP] Arquitectura Con Ciclos. Ejemplo

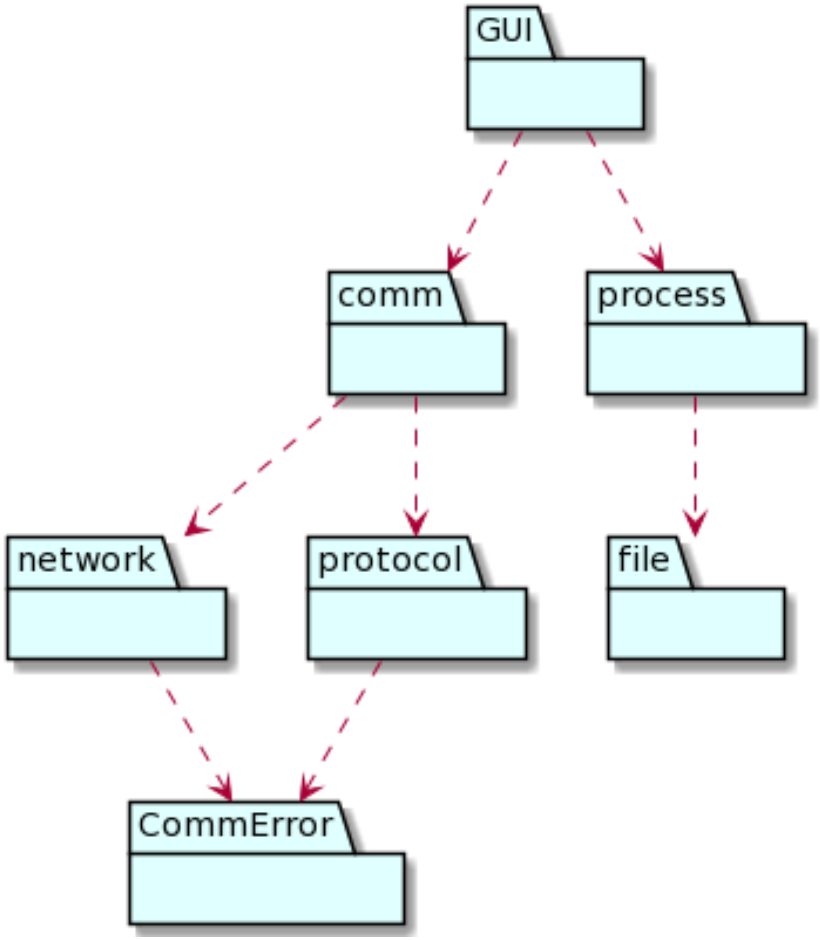
- El ciclo implica transitividad de dependencias
  - MyTask ya dependía directamente de Tasks y de DAO
  - MyTask ya dependía de MyDomain, que comienza el ciclo
  - Por tanto, ahora también depende MyApplication
  - Pero MyApplication depende de Notify UI
  - Pero, a su vez, Notify UI depende de UI View
  - Por otra parte, MyApplication también depende de Task UI
  - Es decir: MyTask ahora depende de **todos los paquetes**
  - Desarrollar MyTask se ha vuelto mucho más difícil
- Pero *exactamente lo mismo* le ocurre a MyDomain
  - De hecho, *exactamente lo mismo* también a MyApplication
- En resumen: MyTask, MyDomain y MyApplication se tienen que desarrollar **a la vez**
  - Esto en un ciclo pasa **siempre** (¡¡¡sorpresa!!!)

[ADP] Eliminación de Ciclos

- Se usa, de nuevo, el DIP

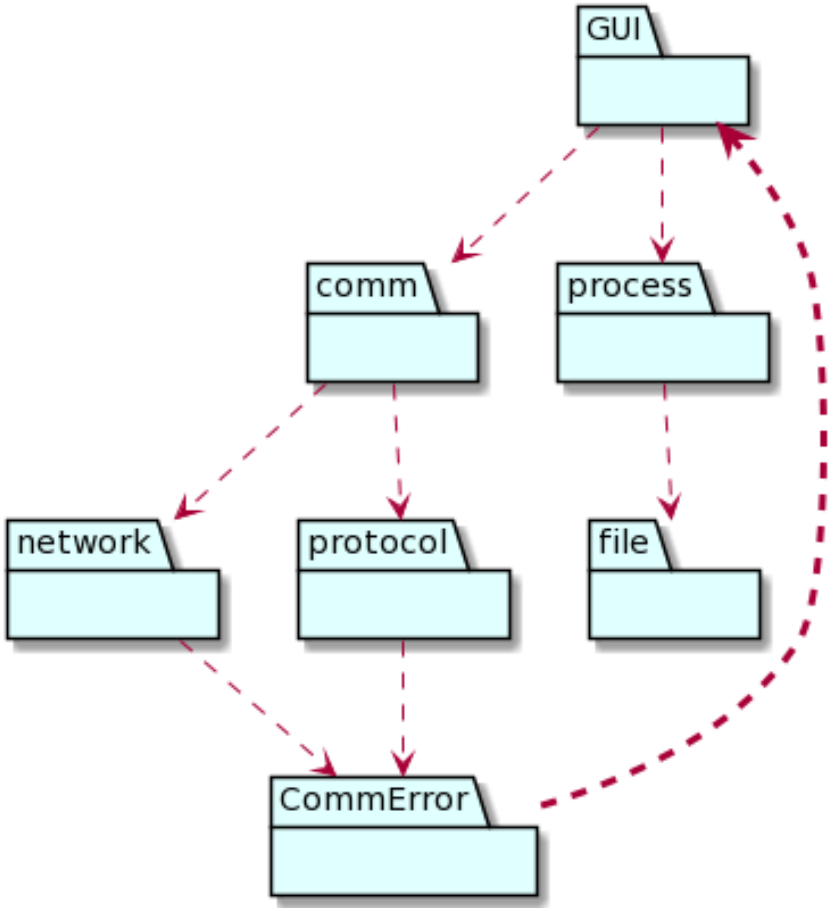


[ADP] Otro ejemplo (I)



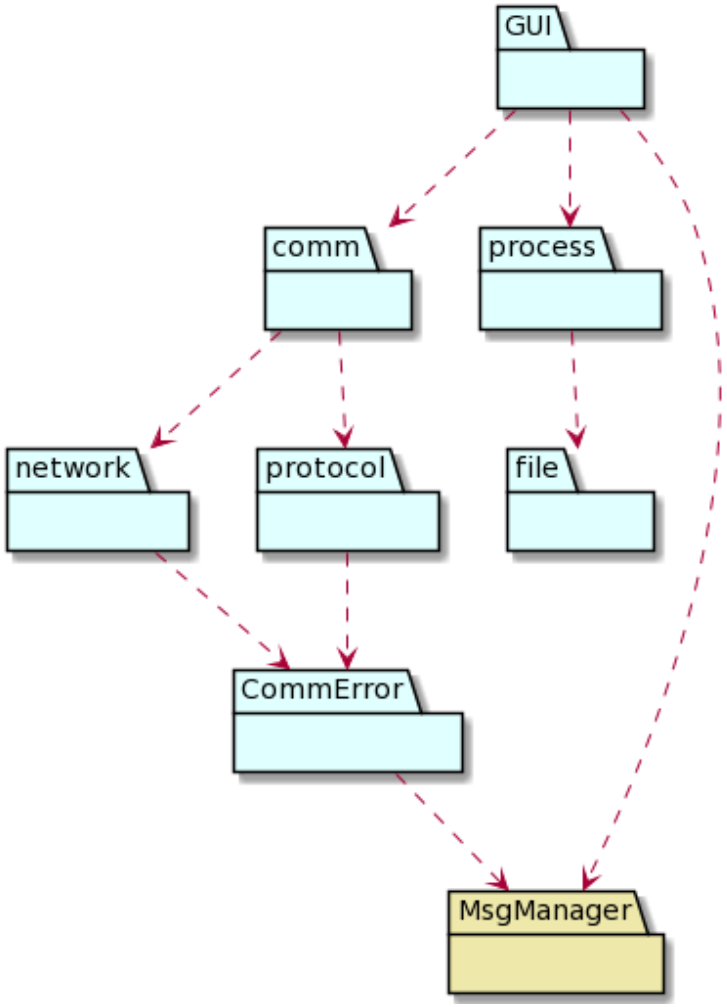
- El desarrollador que trabaja en el paquete CommError decide que tiene que mostrar un aviso por pantalla cuando hay un error
- Como la pantalla está controlada por GUI, envía un mensaje a uno de los objetos de GUI
  - De este modo, se muestra el aviso de error por pantalla
- Es decir, hace que CommError ahora dependa de GUI

[ADP] Otro ejemplo (II)

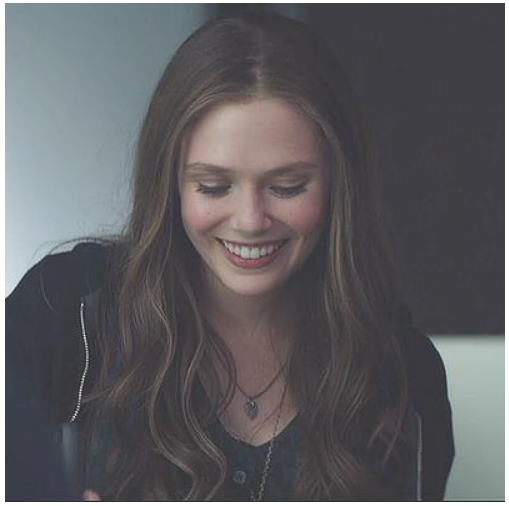


- **Ya hemos añadido un ciclo**
  - Ocurre constantemente
  - No es un “error extraño”

[ADP] Otro ejemplo (III)



- **Lo podemos romper añadiendo un nuevo paquete**
  - Las clases que CommError necesita se sacan de GUI y se colocan en el nuevo paquete MsgManager
  - GUI y CommError dependen ahora de MsgManager. Ya no hay ciclos

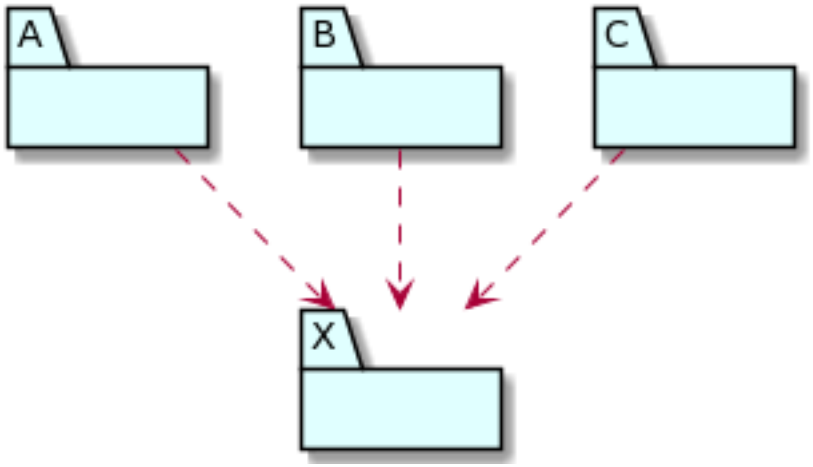


Principio de las Dependencias Estables
SDP: *Stable Dependencies Principle*

> Las dependencias entre paquetes en un diseño debe hacerse buscando la dirección de la estabilidad de los paquetes. Un paquete debe depender sólo de los paquetes que son más estables que él
>
> *Robert C. Martin, 1996*

- Introduce el concepto de buena dependencia
  - Es una dependencia de algo que **no** es volátil
  - Que no tiene interdependencias
  - La volatilidad depende de muchos factores, y es difícil de entender
- Un factor que influye en la volatilidad y es más fácil de medir es la estabilidad
  - Estabilidad: algo *estable* es algo que no es fácilmente cambiable
  - Estabilidad: una medida de la dificultad de cambiar un módulo.

[SDP] Responsabilidad e Independencia

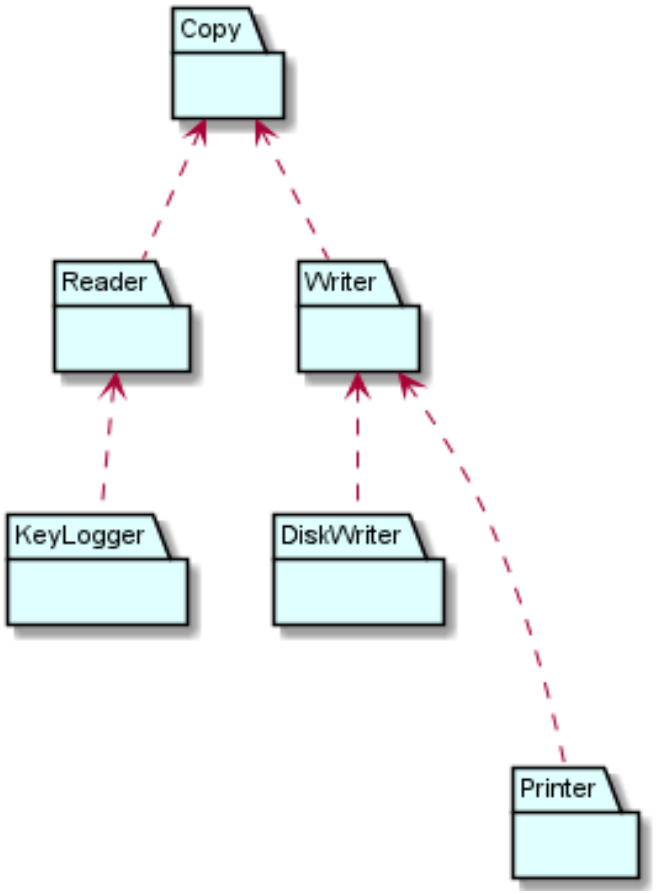


ESTABILIDAD = RESPONSABILIDAD + INDEPENDENCIA

- Los módulos estables tienden a ser poco volátiles

- Un paquete del cual dependen muchos paquetes es muy estable
  - Necesita mucho trabajo para reconciliar los cambios con todos los paquetes dependientes.
- El paquete X tiene tres paquetes que dependen de él.
  - Se dice que es **responsable** de esos tres paquetes.
  - Son tres buenas razones para no cambiar
- Por otra parte X no depende de nadie
  - Se dice que **independiente**

[SDP] Dependencias Estables: Ejemplo



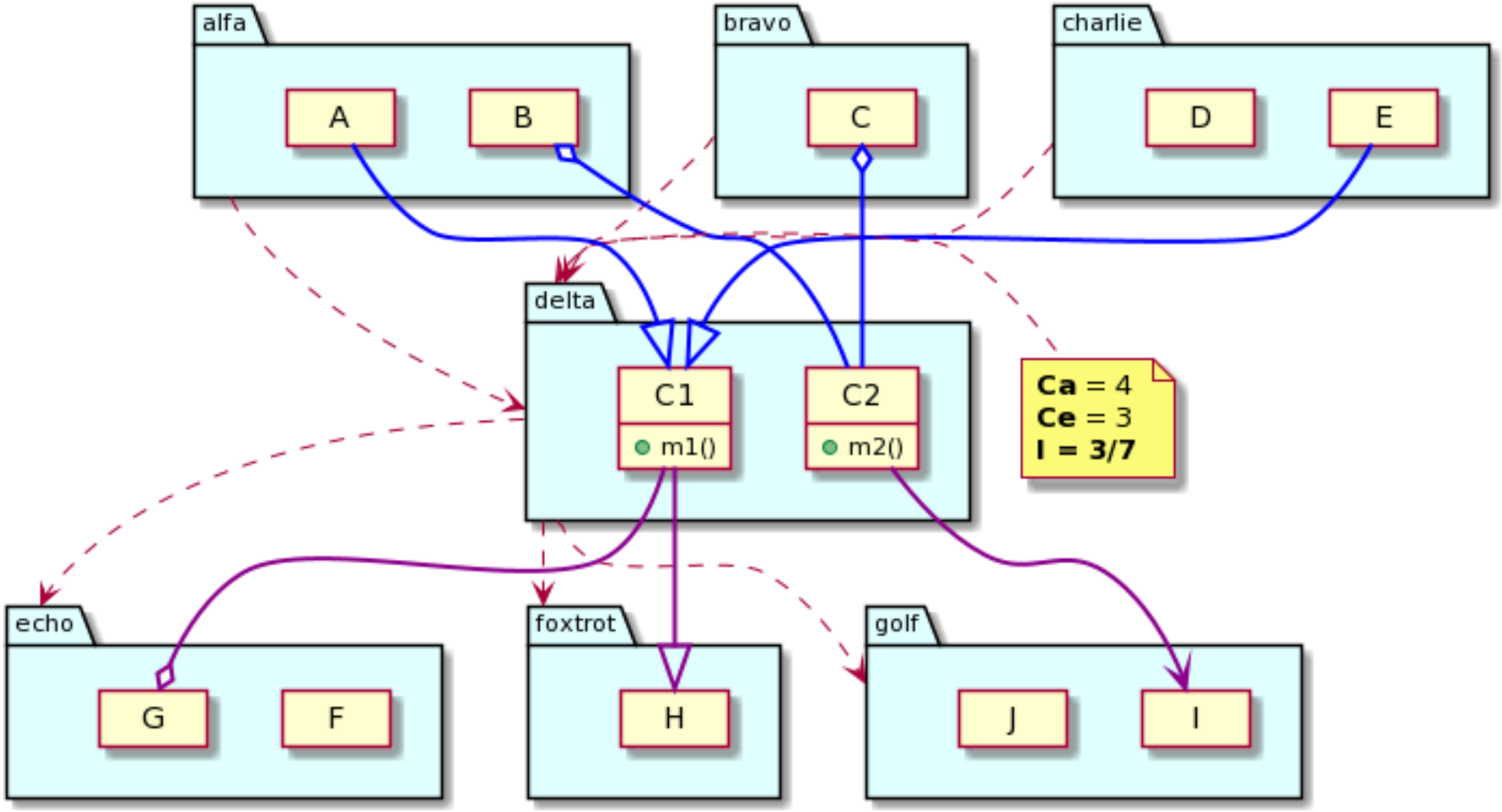
- El clásico ejemplo del Copista, trasladado a paquetes
- No hay interdependencias
- Copy es independiente; es además responsable de Reader y Writer
- Será estable: por tanto, robusto, mantenible y reutilizable
- Se considera inmune a los cambios
- Además, Reader y Writer son paquetes con poca volatilidad
- Ellos mismos son paquetes estables.

Medidas (métricas) de Estabilidad

- Acoplamientos Aferentes (**C_a**)
  - Número de clases fuera del paquete que dependen de las clases contenidas en el paquete.
    - ¿Cómo de responsable soy?
- Acoplamientos Eferentes (**C_e**)
  - Número de clases dentro del paquete que dependen de clases externas al paquete.
    - ¿Cómo de dependiente soy?
- Factor de Inestabilidad (**I**)
  - $I \in [0, 1]$
    - 0 = totalmente estable
    - 1 = totalmente inestable.

$$I = \frac{C_e}{C_a + C_e}$$

[SDP] Cálculo de estabilidad. Ejemplo



Principio de las Abstracciones Estables
SAP/ *Stable Abstractions Principle*

La abstracción de un paquete debe ser proporcional a su estabilidad.
Los paquetes con máxima estabilidad deben ser abstractos.
Los paquetes inestables deben ser concretos.

Robert C. Martin, 1996

- Establece la relación entre estabilidad y abstracción
  - Como podrá verse, es una reformulación del DIP a otra escala
- Las decisiones de diseño (arquitectura) de alto nivel no suelen cambiar
  - No suelen ser volátiles → situarlas en paquetes estables
- ¿Cómo podría un paquete totalmente estable (I=0) ser lo suficientemente flexible como para soportar los cambios?
  - No se puede aspirar a que *todo* sea I=0, o nada podría cambiar
  - Queremos *mejorarlo* sin cambiarlo, ¿esto es posible?
- Respuesta: Principio Abierto-Cerrado
  - Clases que pueden ser extendidas sin modificarlas: Clases Abstractas

Medidas (métricas) de Abstracción (I)

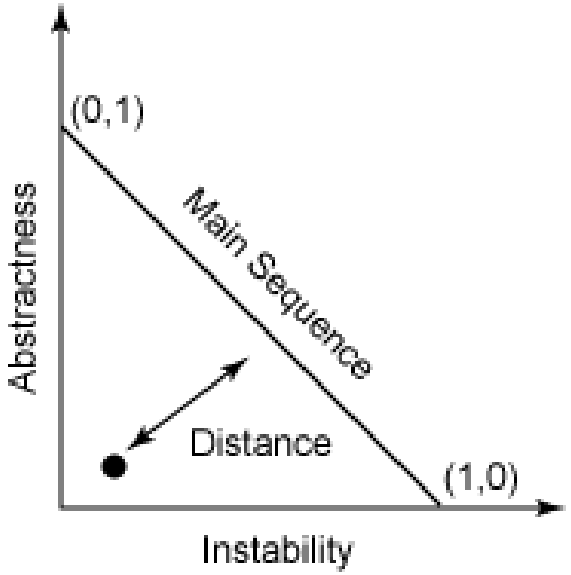
- Abstractividad (*abstractness*, A)
  - Ratio del número de clases abstractas (o interfaces) en el paquete, entre el número total de clases del paquete
  - $A \in [0, 1]$

    - A = 0 totalmente concreto
    - A = 1 totalmente abstracto

  - No es tanto cuestión de que se *declare* todo como abstracto, sino cuál es su verdadera naturaleza
    - Las clases abstractas e interfaces son mecanismos de reutilización que se combinan con la herencia
    - Los paquetes se vuelven “abstractos” a pesar de que sus mecanismos son composición y delegación

Medidas (métricas) de Abstracción (II)

- Distancia a la secuencia principal (D)
  - Distancia *perpendicular* del paquete respecto de la línea ideal, o *secuencia principal*: (A + I = 1)
  - Equilibrio entre estabilidad y abstracción
    - $D = | A + I - 1 |$
    - $D \in [0, 1]$



  - D=0 : está en la secuencia principal
    - Tot. abstracto y estable (A=1, I=0)
    - Tot. concreto e inestable (A=0, I=1)
  - D=1 : totalmente desequilibrado
    - Equilibrado: en el medio entre ambos extremos
      - Idealmente: en la secuencia principal

[SAP] Arquitectura ideal