Principios de paquetes (cohesión)

Principio equivalencia de reúso/liberación (REP)

Principio de Reúso
Agrupar las clases reusables en paquetes que se puedan administrar y reusar.

Separo lo que voy a reusar de lo que no, y lo libero en un ensamblado. En .NET la unidad de release es el assembly, entonces la unidad de reúso también lo es.

Por ejemplo, si quiero separar una solución en componentes siguiendo este principio, separo cada paquete de clases en un componente (así lo que uso junto lo libero junto).

Principio de clausura común (CCP)

Principio de mantenimiento
Agrupar las clases que cambian juntas. Las clases pertenecientes a un paquete deben de cambiar por el mismo tipo de cambio. Si un cambio afecta a un paquete, entonces afecta a todas las clases del mismo y a ningún otro paquete.

Principio de reúso común (CRP)

Principio de Reúso
Las clases que pertenecen a un paquete se reúsan juntas.

Principios de paquetes (Acoplamiento)

Principio de dependencias acíclicas (ADP)

- El grafo de dependencias entre paquetes debe ser dirigido y acíclico.
- Como solucionar si se crea un ciclo:
    - Con inversión de dependencias (DIP)
    - Crear un nuevo paquete que agrupe clases que ambas dependen.

Principio de dependencias estables (SDP)

- Las dependencias entre paquetes deben ir en el sentido de la estabilidad.
- Un paquete debe depender solamente de paquetes que sean más estables que él.

Principio de abstracciones estables (SAP)

Un paquete debe ser tan abstracto como es su estabilidad. Los paquetes más estables (dependen muchos de él y él no depende) deben tender a ser abstractos (con interfaces o clases abstractas). Paquetes inestables deben ser concretos. Un paquete estable debe ser abierto a la extensión y paquetes inestables deben ser fácilmente modificables. Por ejemplo, mediante OCP un paquete estable puede ser fácilmente extensible. Con DIP a nivel de paquete hago los paquetes estables abstractos y fácilmente extensibles por paquetes menos estables

- Un paquete debe ser tan abstracto como es su estabilidad.
- Los paquetes más estables deben tender a ser más abstractos mientras que los mas inestables deben ser más concretos.

Tensión

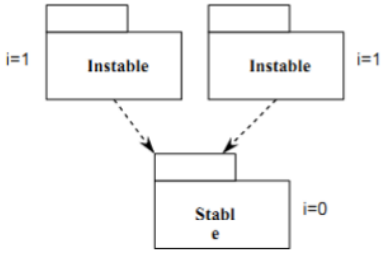
REP (Liberación y reúso) **CCP** (Clausura común) **CRP** (Reúso común)

Los tres principios son excluyentes.

CCP quiere paquetes grandes así cambian por lo mismo, mientras que CRP quiere paquetes chicos.

CCP es de mantenimiento, mientras que CRP y REP son para reúso. Se comienza favoreciendo al mantenimiento y después se ve si se puede reusar.

Dependencias de paquetes



Métricas

- **Atributo**: Una propiedad mesurable, física o abstracta, que comparten todas las entidades de una misma categoría.
- **Medida**: Es una indicación cuantitativa de extensión, cantidad, dimensiones, capacidad de algunos atributos de un proceso o producto. [Pressman]
  Resultado de una medición. Ejemplo: Número de defectos.
- **Métrica**: Una forma de medir y una escala definidas para realizar mediciones de uno a varios atributos. Ejemplos: Promedio de defectos por revisión, promedio de defectos por persona.

Se podrían dividir en tres categorías:

1. Orientadas al paquete (assembly)
   a. Ejemplo: Afferent Couplings (Ca), que indica el número de clases fuera del paquete que dependen de clases del paquete analizado.
2. Orientadas a la clase
   a. Ejemplo: Número de subclases de una clase.
3. Orientadas a los métodos
   a. Ejemplo: Complejidad Ciclomática, que indica la complejidad del método.

Cohesión Relacional →CCP

Indica cuán fuertemente relacionadas están las clases dentro de un paquete. Debería estar entre 1,5 y 4.

$$H = \frac{R + 1}{N}$$

R: Número de relaciones entre clases dentro de un paquete. Solo se cuentan las relaciones de asociación, composición y agregación. Las relaciones bidireccionales se cuentan por dos.
N: Número de clases e interfaces en un paquete.

Inestabilidad →SDP

Indica cuán inestable es un paquete. Refiriéndose a cuán fácil es cambiarlo dependiendo de cuántos paquetes lo utilizan.

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

Ca: Dependencias entrantes.
Ce: Dependencia salientes.

Resultado:
Cerca de 0: Estable (se debe intentar EXTENDER haciéndolo abstracto mediante DIP, OpenClose)
Cerca de 1: Inestable (depende de pocos paquetes. Este paquete es fácil de cambiar debido a que impacta en pocos paquetes)

Abstracción:

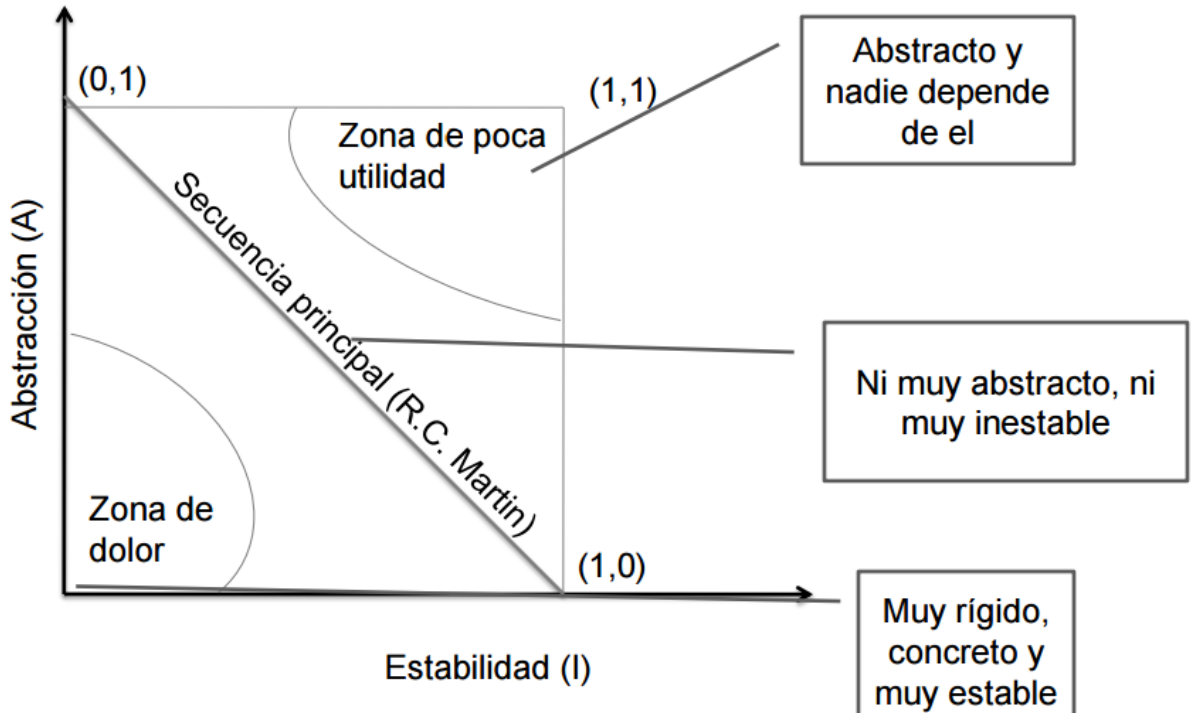
Indica cuán abstracto es un paquete.

$$A = \frac{N_a}{N_c}$$

Na: Cantidad de clases abstractas e interfaces.
Nc: Cantidad de clases, incluyendo las abstractas e interfaces.

Resultado:
Cerca de 0: Paquete concreto (deben crearse interfaces y hacer que las clases que dependen de él, pasen a depender de las interfaces. Los paquetes más concretos deberían ser los más cercanos a la lógica de negocio.)
Cerca de 1: Paquete abstracto (son interfaces. Este paquete es muy poco concreto. Se debe evitar que los paquetes abstractos sean muy inestables, ya que si no será trabajo sin sentido porque serian interfaces que nadie usa)

Abstracción vs estabilidad →SAP



NO SON BUENOS VALORES CERCA DE CERO NI DE UNO.

Normalizada de D:

$$D' = \left| A + I - 1 \right| = \left[0 \, , \, 1 \right]$$

Cuanto más cerca de 0, más cerca de la secuencia principal, mientras cerca de uno quiere decir que esta lejos de la secuencia principal.

Valores cercanos a 0,0 (zona de dolor) indican que el paquete es concreto y estable (responsable). Estos paquetes no son buenos porque no son extensibles y si cambian impactan en otros.

Valores cercanos a 1,1 (zona de poca utilidad) indican que el paquete es abstracto e inestable. El paquete es extensible, pero tiene pocos paquetes que dependan de él.

Síntomas de un diseño en deterioro

Rigidez

Es la tendencia del software a ser difícil de cambiar, incluso en las cosas más sencillas. Cada cambio produce una cascada de cambios en módulos dependientes.

Fragilidad

Es la tendencia que tiene el software a generar fallas cuando sufre un cambio. Es común que las fallas ocurran en sitios o módulos que no están relacionados conceptualmente con el área que se está cambiando.

Inmovilidad

Es la resistencia del software a ser reutilizado en otros proyectos o parte de otros proyectos. Pasa muchas veces que un programador descubre que necesita un módulo que es muy parecido a otro que ha desarrollado otro programador. Sin embargo, también pasa muchas veces que el módulo en cuestión depende demasiado de la aplicación en la que está integrado.

Viscosidad

Es la dificultad de mantener la filosofía del diseño original. Cuando se afronta un cambio los programadores encuentran normalmente más de una manera de realizarlo. Algunas de estas formas preservan la filosofía del diseño y otras no. Cuando las formas de implementar el cambio preservando el diseño son más difíciles de realizar que las que no lo preservan, entonces la viscosidad del diseño es alta.