Clase 12 - DA2 Teórico
01/02/2021

Requerimiento de importadores:

1. El administrador inicia sesión correctamente
2. Se va a dirigir a una sección de “Importar Bugs”
3. Se va a desplegar un lista de importadores
    1. Van a tener que realizar un endpoint que recorra todas las dll que esta en la carpeta de importadores. Crear instancias mediante Reflection y retornar la lista de importadores con su path, nombre y lista de parámetros.
4. El usuario va a tener que llenar los parámetros y enviarlos junto con el path del importador
    1. Un endpoint que reciba el path de importador seleccionado y la lista de los parámetros con sus valores correspondientes.
    2. Crear una instancia de este importador mediante Reflection y ejecutar el método de obtener los bugs con estos parámetros.
    3. Una vez que tengo la lista de bugs, simplemente los persisto en mi aplicación (base de datos).

Código anexado

Patrón Observer

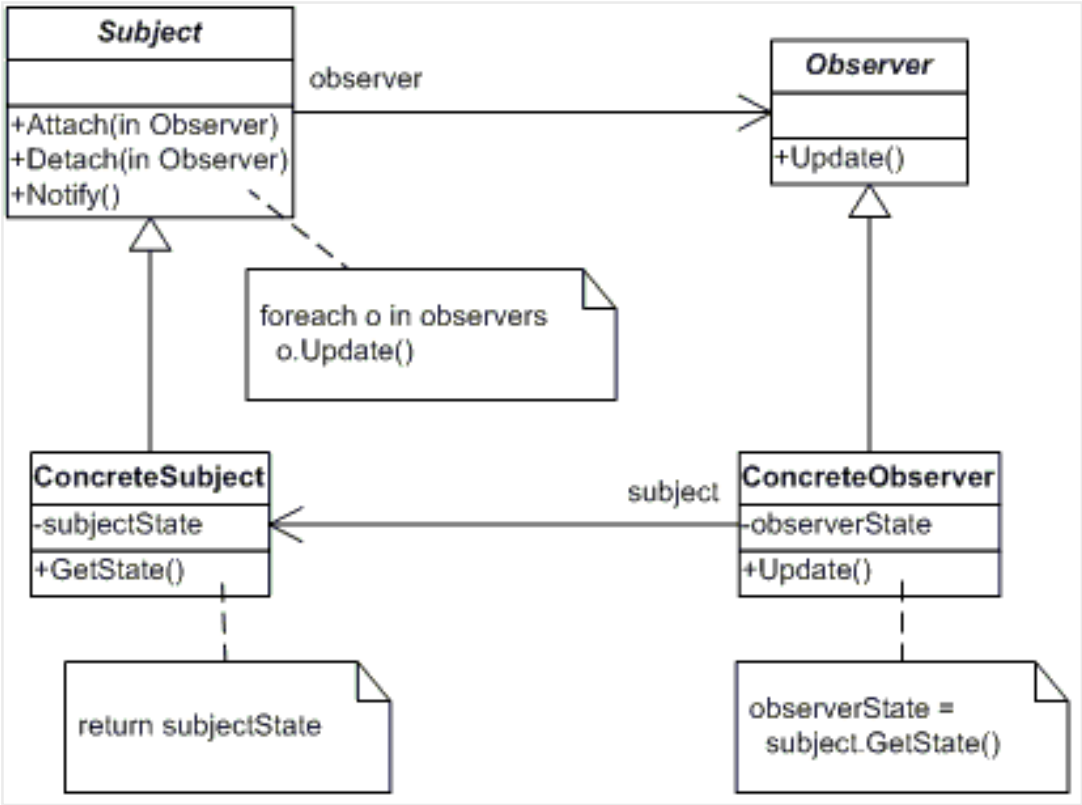
Clasificación: Comportamiento

Intención:
Define una relación 1 a N entre objetos de forma que cuando un objeto cambia su estado, todos sus objetos dependientes son avisados y se actualizan automáticamente.

Al objeto observado también se lo denomina **SUJETO** y mantiene

una lista de sus dependientes denominados **OBSERVADORES**.

Motivación: Dado un cierto sujeto, este puede tener un número de observadores. Todos los observadores son avisados cuando el sujeto pasa por un cambio en su estado. En respuesta, cada observador cambiará su estado de acuerdo al estado del sujeto que observa, "se sincronizan".



Aplicabilidad:

- Usamos este patrón cuando cambios en el estado de un objeto (observado o el publicador) puede llegar a requerir cambiar en el estado de otros objetos (observadores o los suscriptores) y el conjunto real de dichos objetos observadores es desconocido de antemano o cambia dinámicamente.
- Cuando un cambio en un objeto debe provocar cambios en otros, pero no se sabe cuántos necesitan cambiar. Por ejemplo: tengo un stock de un supermercado que es enorme y es necesario que cada vez que el stock se reduzca o cambie, cada sistema/objeto particular sea avisado.

Ventajas:

- Bajo acoplamiento. Estamos logrando una comunicación "asíncrona" dado que el sujeto (publicador) no necesita conocer los detalles de implementación de los observadores (suscriptores).
- Cumple con OCP. Podemos agregar nuevos observadores sin tener que cambiar el código de el sujeto (y viceversa si tenemos una interfaz para los sujetos).
- Podemos establecer relaciones entre objetos en runtime.

Desventajas:

- El sujeto (publicador) puede llegar a enviar actualizaciones (llamar al Update()) que no le interesen a alguno de los Observadores (suscriptores).
- Esto obliga a que cada observador tenga que distinguir según el evento que se esta notificando e identificar si quiere hacer algo con él o no.
- Las notificaciones para los observadores se realizan en orden aleatorio.

Principios a nivel de paquetes

Tenemos dos tipos de principios:

- A nivel de cohesión: estos principios van a decir cómo ubicamos clases dentro de uno o varios paquetes apuntando a una **alta cohesión.**
- A nivel de acoplamiento: estos principios nos van a decir cómo relacionamos paquetes entre sí y como tienen que ser sus dependencias, apuntando a un **bajo acoplamiento.**

Principios a nivel de cohesión:

1. **REP (Principio de Equivalenca Reuso/Liberación - Reuse/ Release Equivalence Principle)**

"La granularidad del reuso debe ser la misma que la granularidad de la liberación"

Reuso: La unidad de reuso es el paquete. Lal forma en que nosotros reusamos código es a través de importar un cierto paquete (using).
Liberación: En .NET la unidad de liberación (o de release) es el assembly (.dll o .exe)

Lo que entonces termina formulando este principio es la siguiente equivalencia: **1 paquete = 1 assembly**

Este principio nos indica entonces que el paquete es la unidad de reuso y por ende el sistema debe estar particionado en paquetes reusables. No debería pasar que para poder utilizar el paquete A, yo tengo que referenciar a los ensamblados A1, A2 y A3. y viceversa, si yo quiero usar únicamente A1, no debería pasar por el ensamblado A2, A3.
Este principio apunta al reuso, y cada vez que reusamos una clase no queremos tener que recompilar toda la solución si dicha clase cambia.

Nuestro criterio de agrupación para mantener clases dentro de un paquete es entonces el reuso.

2. **CCP (Principio de Clausura Común o Common Closure Principle)**

“Las clases pertenecientes a un paquete, deben cambiar por el mismo tipo de cambio. Si un cambio afecta a un paquete, afecta a todas las clases del mismo y a ningún otro paquete”

Clases que cambian juntas, deben pertenecer juntas (dentro del mismo paquete)

Plantea una idea muy similar al SRP pero a nivel de paquetes.

Aquellas clases que cambien o que son candidatas a cambiar por el mismo motivo de cambio se deben agrupar en un mismo paquete, de esa forma podemos minimizar el impacto de cambio. De lo contrario, si las clases que cambian juntas estuvieran

esparcidas en diferentes paquetes, el impacto de cambio sería mucho mayor.

Este principio apunto al mantenimiento (por eso hablamos de impacto de cambio)

3. **CRP (Principio de Reuso Común o Common Reuse Principle)**

“Las clases que pertenecen a un paquete se reusan juntas. Si se reusan clases del paquete entonces se reusan todas.”

Si se cambia una clase de un paquete, dicho cambio nos obliga a liberar (recompliar) el componente nuevamente.

Por ende, como queremos minimizar la cantidad de veces que liberamos un nuevo componente, las clases que no se reusan juntas, no deberían pertenecer juntas.

Esta última idea contradice el principio CCP y nos lleva a introducir la noción de “Tensión entre principios”. Esta nos sugiere que no vamos a poder cumplir con todos estos principios a la vez, dado que proponen argumentos diferentes y contradictorios. Para resolver esta tensión, Robert Martin plantea que apliquemos los principios según la etapa y madurez del software al cual nos estamos enfrentando:

1. Si el software se encuentra evolucionando(es inmaduro, el diseño y requerimientos cambian, estamos en una etapa temprana del desarrollo, etc) debemos favorecer al CCP (2) dado que me interesa ganar en **mantenibilidad** y minimizar el impacto de todos los cambios.
2. Si el software y la arquitectura es estable favorecemos el **reuso** y aplicando REP(1) y CRP(3).