Clase 10 - DA2 Teórico

18/10/2021

Proxy

Clasificación: Estructural

Intención: Proveer una clase (el proxy) que limite el acceso a otra clase.

Motivación: Se necesita una clase que limite el acceso a otra clase. Esto lo podríamos hacer por razones de seguridad o porque un objetos es muy caro de crearlo siempre (y entonces quiero crearlo una vez sola cuando se necesite) o porque tengo un objeto remoto al que quiero acceder.

La idea fundamental es usar un nivel de indirección para realizar un control de acceso sobre cierta clase. Y quien realiza esto es el proxy. Básicamente el proxy es un intermediario que agrega cierta lógica adicional a la hora de interactuar con cierta clase.

Aplicabilidad:

1. **Remote Proxy:** provee un representante local para un objeto que se encuentre en un espacio o maquina diferente (e.g. cuando desde angular creamos objetos para interactuar con nuestra API, o si quisiéramos hacer que nuestra API interactúe con otra API.
2. **Virtual Proxy:** crea objetos costosos a demanda.
3. **Protection Proxy:** controla los permisos de acceso a un objeto original. Sería un clase que actúa como intermediario permitiendo o no el acceso de una clase a otra.
4. **Smart Proxy / Reference:** reemplaza una referencia simple por algún objeto que realiza operaciones adicionales cuando es accedido.

———————————————

Remote proxy:

Web API:
ProjectC#
Name: Cardiex

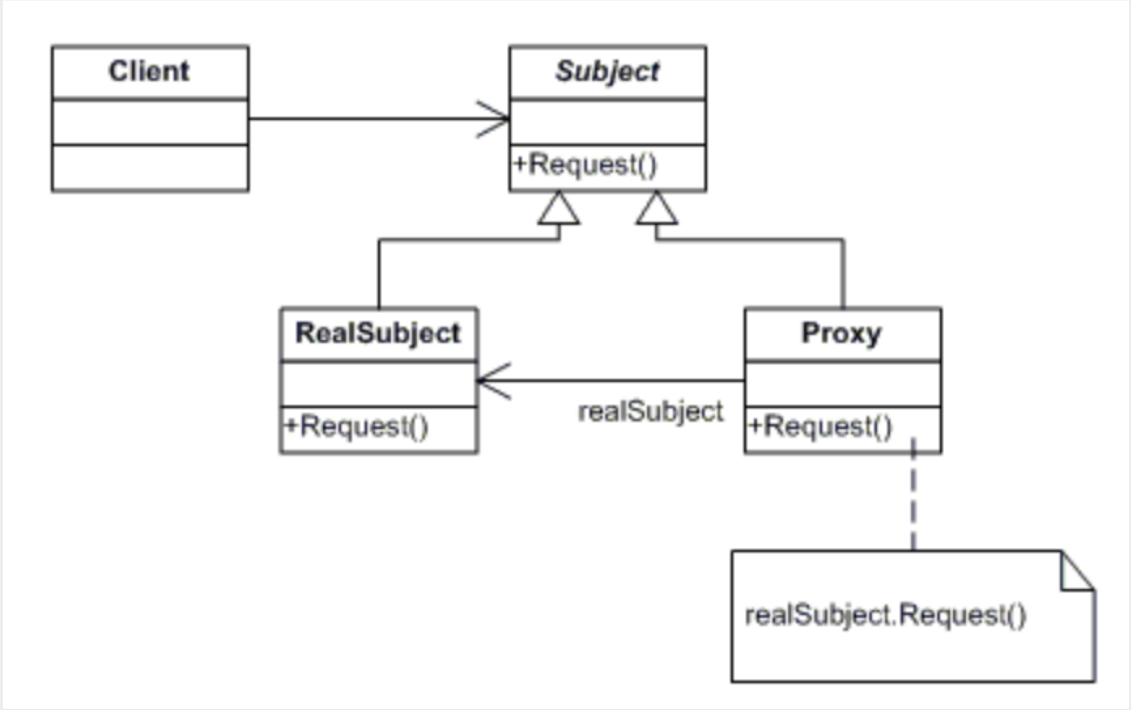
NET CORE - C# Nodejs - Typescript
Web API <—> Angular

Angular:
ProjectTS
Name: Cardiex

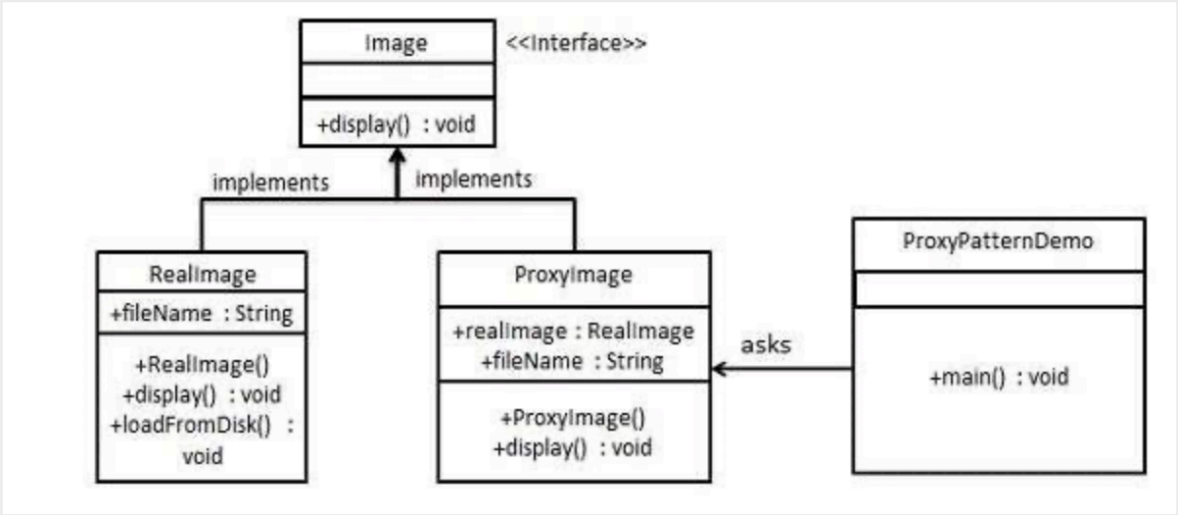
En angular voy a manipular el objeto ProjectTS como si fuera el de ProjectC#.

———————————————

Estructura



Ejemplo de aplicación:



Código:

```
public interface Image {
   void display();
}

public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}

public class ProxyImage implements Image{
```

```
  private List<RealImage> realImages;
  private String fileName;

  public ProxyImage(String fileName){
    this.fileName = fileName;
  }

  @Override
  public void display() {
        var realImage = realImages.FirstOrDefault(ri => ri.fileName
= fileName);
    if(realImage == null){
      realImage = new RealImage(fileName);
      this.realImages.add(realImage);
    }
    realImage.display();
  }
}
```

Ventajas:

- La implementación del proxy decide cuándo llamar a la operación real y cuando llamar o agregar su propio comportamiento.
- OCP: el cliente le es indistinto saber como se esta trabajando (si usa el proxy o el objeto real) y eso se da gracias a la clase abstracta/interfaz de arriba dado que la misma permite que el cliente pueda trabajar con las dos implementaciones indistintamente. Con el objeto real llamaría directamente al comportamiento deseado y con el objeto proxy podemos realizar optimizaciones o decidir cuando llamar al objeto real en cuestión.

Desventajas:

- El código se puede volver más complicado porque tenemos que agregar nuevas clases.
- La lógica adicional que el proxy agregue, puede llegar a retrasar la respuesta.

Composite

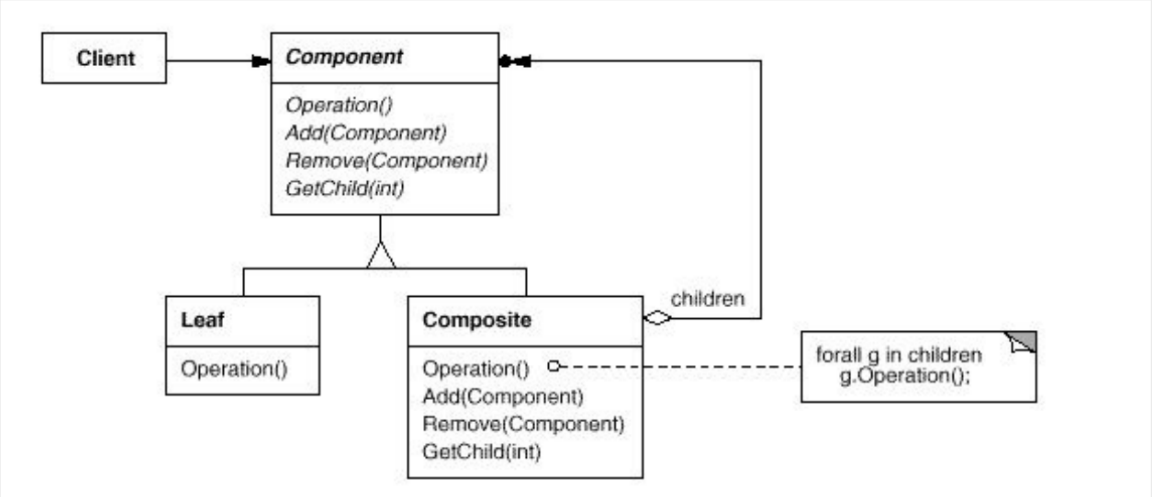
Clasificación: Estructural

Intención: Queremos mantener y tratar objetos simples y compuestos de forma uniforme. Ejemplo: carpetas pueden contener archivos u otras carpetas.

Aplicación:

- Nuestra aplicación necesita manipular una colección jerárquicas de objetos “primitivos” (archivos) y objetos compuestos (carpetas)
- Cuando cierto procesamiento sobre un objeto primitivo se realiza diferente de un objeto compuesto y no me interesa saber el tipo de cada objeto antes de llevar a cabo tal procesamiento (NO RTTI). Queremos tratarlos de forma uniforme.

Estructura:



Los objetos se deben componer de forma recursiva y no hay distinción (o si la hay es muy poca) entre los objetos compuestos y los simples. Esto es debido a que si me empiezan a aparecer muchas diferencias entre las interfaces de cada elemento (compuesto y componente) no vale la pena forzar el composite.

Detalles de implementación: seguridad vs transparencia:

- Seguridad: Si definimos estas operaciones en la clase Composite, obtenemos seguridad. Porque cualquier intento de borrar o agregar objetos sobre un Leaf sera atrapado en tiempo de compilación.
- Transparencia: Si definimos todas las operaciones en la raíz de la jerarquía, obtenemos transparencia. Porque podemos tratar a todos los componentes de forma uniforme, sin embargo nos cuesta en “Seguridad”.

Ventajas:

- Permite tratar todos los componentes de igual manera independientemente de su complejidad (para los clientes les da igual si están trabajando con un Leaf o con un Composite)
- Permite agregar nuevos tipos de componentes de forma sencilla: los clientes no se enteran ni cambian

Desventajas:

- El diseño puede ser a veces muy general y no todo componente puede ser generalizado: es sencillo agregar nuevos componentes pero es más difícil restringir a los componentes, ya que la interfaz/clase abstracta de arriba siempre obliga a tener todo el comportamiento implementado. Esto nos lleva a que no nos podamos confiar en el type system para forzar esas limitaciones para nosotros, tenemos que hacer dichos chequeos en runtime (RTTI).