9. Strukturális tervezési minták II.

## Facade (Structural pattern) („Homlokzat”)

* **Egységes interfészt definiál egy alrendszer interfészeinek halmazához.**
* **Probléma**
  + Kód széleskörű objektumokkal rendelkezik, amik egy library-hez vagy keretrendszerhez tartozik.
  + Normális esetben az összes objektumot inicializálni kellene, nyomon követni a függőségeket, a metódusokat a megfelelő sorrendben végrehajtani és így tovább.
  + Ennek eredményeként az osztályok üzleti logikája szorosan összekapcsolódik a harmadik féltől származó osztályok megvalósítási részleteivel, ami megnehezíti a megértést és a karbantartást.
* **Megoldás**
  + Csak a tényleges funkciókat tartalmazza.
  + Praktikus, ha integrálni kell az alkalmazást egy library-vel, ami sok funkcióval rendelkezik, de csak egy kis részére van szükség.

### Facade használati esetek

* Akkor használjuk, ha egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé.
* Akkor használjuk, ha számos függőség van a kliens és az alrendszerek osztályai között.
* Rétegezéskor

### Facade implementációja

1. Nézzük meg, hogy lehet-e egyszerűbb interfészt biztosítani, mint amit egy meglévő alrendszer biztosít.
2. Interfész implementálása az új facade osztályban.
3. A facade-nek át kell irányítania a kódból érkező hívásokat az alrendszer megfelelő objektumaihoz.
4. Innentől kezdve a kódban csak a facade-en keresztül kommunikáljon az alrendszer.
   1. Mostantól a kód védve van az alrendszer kódjának bármilyen változásától.
   2. Ha egy alrendszer új verzióra frissül, csak a facade kódot kell módosítani.

### Facade előnye és hátránya

* **Előny**
  + Elszigetelhető a kód az alrendszer komplexitásától.
* **Hátrány**
  + „god object” lehet belőle

## Proxy (Structural pattern)

* **Objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést.**
* **Probléma:** Miért akarjuk ellenőrizni az objektumhoz való hozzáférést?
  + Van egy hatalmas objektum, ami rengeteg rendszererőforrást fogyaszt és időnként szükség van rá, de nem mindig.
* **Megoldás**
  + Hozzunk létre egy új proxy osztályt, aminek interfésze megegyezik az eredeti service objektummal.
  + Ezután frissíti az alkalmazást, hogy átadja a proxy objektumot az eredeti objektum összes kliensének.
  + A klienstől érkező kérés fogadásakor a proxy létrehoz egy valódi service objektumot és mindent átad neki.
* **Haszna**
  + Ha valamit az osztály alapvető logikája előtt vagy után kell végrehajtani, a proxy lehetővé teszi, hogy ezt az osztály megváltoztatása nélkül tegye.
  + Mivel a proxy ugyanazt az interfészt valósítja meg, mint az eredeti osztály, átadható bármely olyan kliensek, ami valódi szolgáltatásobjektumot vár.

### Proxy típusok

* **Távoli Proxy:** Távoli objektumok lokális megjelenítése „átlátszó” módon, tehát a kliens nem is érzékeli, hogy a tényleges objektum egy másik címtartományban vagy egy másik gépen van.
* **Virtuális Proxy:** Nagy erőforrás igényű objektumok szerinti létrehozása, például egy kép.
* **Védelmi Proxy:** A hozzáférést szabályozza különböző jogoknál.
* **Smart Pointer:** Egy pointer egységbezárása, hogy bizonyos esetekben automatikus műveleteket hajtson végre, például lockolás.

### Proxy implementációja

1. Service interfész létrehozása vagy a proxy a service osztály alosztálya lesz és így örökli a service interfészét.
2. Proxy osztály létrehozása és egy field-et deklarálni kell, hogy lehessen hivatkozni a service-re.
3. Proxy metódusok implementálása.
4. Meg kell fontolni egy olyan létrehozási módszer bevezetését, ami eldönti, hogy a kliens proxy vagy valódi service-t kap-e. (Ez lehet egy statikus vagy factory metódus is.)
5. Service objektum inicializálása.

### Proxy előnyei és hátrányai

* **Előnyök**
  + A service objektumot a kliensek tudta nélkül is lehet vezérelni.
  + Akkor is működik a proxy, ha a service objektum nem áll készen vagy nem elérhető.
  + Open/Closed elv alapján működik, tehát a service vagy a kliensek módosítása nélkül új proxy-kat lehet bevezetni.
* **Hátrányai**
  + Bonyolult kód sok új osztálynál.
  + A service válasza késhet.

## Decorator (Structural pattern)

* **Objektumok funkciójának dinamikus kiterjesztése, vagyis rugalmas alternatívája a leszármaztatásnak.**
* **Probléma**
  + Van egy notification library, amit más program arra használnak, hogy fontos eseményekről küldjön értesítést.
  + Használatkor kiderül, hogy csak email-eket lehet vele küldeni, és a programban pedig SMS-eket szeretne küldeni és így tovább.
  + Így alosztályokat hozunk létre, amik több értesítési módszert kombinálnak egy osztályon belül, de ez azért **nem jó**, mert a könyvtári és a kliens kódot is megnöveli nagy mértékben.
* **Megoldás**
  + Decorator-öket kell csinálni a különböző metódusokból, például az értesítő módszereknél, csinálunk SMS, Facebook, stb decorator-öket.
  + A decorator-ök ugyanazokat az interfészeket használják.
  + Példaként ha fázom, akkor felveszek egy pulóvert és ha még mindig fázok, akkor egy kabátot is felveszek.

### Decorator használati esetek

* Akkor használjuk, ha dinamikusan szeretnénk funkcionalitást/viselkedést hozzárendelni az egyes objektumokhoz.
* Akkor használjuk, ha a funkcionalitást a kliens számára átlátszó módon szeretnénk az objektumhoz rendelni.
* Akkor használjuk, amikor a származtatás nem praktikus.

### Decorator implementálása

1. Absztrakt Component osztály létrehozása az elvégzendő művelet metódussal.
2. Absztrakt Decorator osztály létrehozása és Component implementálása.
   1. Rendelkeznie kell egy Component privát field-el, amivel meghívjuk a műveletet.
3. A konkrét Component osztályban Component implementálása és itt hajtjuk végre a tényleges elvégzendő „műveleteket”.
4. A konkrét Decorator osztályokban implementáljuk a Decorator-t.

### Decorator előnyei és hátrányai

* **Előnyei**
  + Sokkal rugalmasabb, mint a statikus öröklődés.
  + Több testreszabható osztály határozza meg a tulajdonságokat.
* **Hátrányai**
  + Bonyolultabb, mint az egyszerű öröklés, mert több osztály szerepel.
  + A decorator és a dekorált komponens interfésze azonos, de maga az osztály nem ugyanaz.