

Gang-of-Four tervezési minták 6

Struktúrális minták

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 kell egy library-vel, ami sok funkcióval rendelkezik, de csak egy kis részére van szükség.
 - **Példa:** Egy alkalmazás, ami rövid videókat tölt fel egy platformra, ami egy összetett videókonvertáló könyvtárat használ.
 - Tehát azon az osztályon belül eléri azt a metódust, amivel lehet konvertálni és azt hozzátcsoljuk a konvertáló library-vel, akkor megvan az első facade.

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.
- Rétegeléskor

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.
 - a. Mostantól a kód védve van az alrendszer kódjának bármilyen változásától.
 - b. 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?
 - o Van egy hatalmas objektum, ami rengeteg rendszererőforrást fogyaszt és időnként szükség van rá, de nem mindig.
- **Lusta megoldás**
 - o Csak akkor hozzuk létre az objektumot, amikor tényleg szükség van rá.
 - o Végre kellene hajtani néhány késleltetett inicializálási kódot, de ez kód duplikációt okozna.
- **Megoldás**
 - o Hozzunk létre egy új proxy osztályt, aminek interfésze megegyezik az eredeti service objektummal.
 - o Ezután frissíti az alkalmazást, hogy átadja a proxy objektumot az eredeti objektum összes kliensének.
 - o A kienstől érkező kérés fogadásakor a proxy létrehoz egy valódi service objektumot és mindent átad neki.
- **Haszna**
 - o Ha valamit az osztály alapvető logikája előtt vagy után kell végrehajtani, a proxy lehetővé teszi, hogy ezt az osztály megváltoztatása nélkül tegye.
 - o Mivel a proxy ugyanazt az interfészt valósítja meg, mint az eredeti osztály, átadható bármely olyan kliensek, ami valódi szolgáltatásobjektumot vár.

Proxy struktúra

- **Subject:** Közös interfészt biztosít a Subject és a Proxy számára (így tud a minta működni).
- **Realsubject:** A valódi objektum, amit a proxy elrejt.
- **Proxy:** Helyettesítő objektum, ami tartalmaz egy referenciát a tényleges objektumra, hogy el tudja azt érni. Szabályozza a hozzáférést a tényleges objektumhoz, feladata lehet a tényleges objektum létrehozása, törlése.

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

Decorator (Structural pattern)

- **Objektumok funkciójának dinamikus kiterjesztése, vagyis rugalmas alternatívája a leszármaztatásnak.**
- **Probléma**
 - o Van egy notification library, amit más program arra használnak, hogy fontos eseményekről küldjön értesítést.
 - o Használatkor kiderül, hogy csak email-eket lehet vele küldeni, és a programban pedig SMS-eket szeretne küldeni és így tovább.
 - o 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**
 - o Decorator-öket kell csinálni a különböző metódusokból, például az értesítő módszereknél, csinálunk SMS, Facebook, stb decorator-öket.
 - o A decorator-ök ugyanazokat az interfészeket használják.
 - o Példaként ha fázom, akkor felveszem egy pulóvert és ha még mindig fázom, akkor egy kabátot is felveszek.

Decorator használati esetek

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

Decorator előnyei és hátrányai

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

Flyweight (Structural pattern) trükkök

- Nincs konkrét megoldás, sok trükköt biztosít a Flyweight minta.

On-the-fly property-k

- A memóriában nem foglalnak helyet ezek a property-k.
- Amikor az adott property-t lekérjük, akkor lazy loading elven akkor hajtódik végre, amikor szükség van rá.
- Amikor a főprogram elkéri az adott property-t, akkor hajtódik végre a „levegőben”, emiatt nevezzük on-the-fly property-nek.
- El kell dönteni, hogy mikor akarjuk használni, mert például ha rengeteg adat van és például azokon akarunk átlagolni, akkor az sokáig is eltarthat.
- Ha nem használjuk, akkor pedig használjunk külön szálakat, aszinkron metódusokat például.

Objektumok közös részeinek eltárolása egyszer

- Példány szintjén is megnézhetjük az adott tulajdonságot.
- Felesleges tárolást lehet vele kiváltani, mert olyan jellemzőket teszünk bele, amiket nem szeretnénk módosítani.
- Mivel ez egy megosztott objektum és ha átírunk valamit, akkor az összes többi példányra kihatással van.
- Így érdemes védeni az írás ellen, tehát olvashatóként kell definiálni.

Újrahasznosított objektumok

- Lényege, hogy ne hozzunk létre újabb objektumot például egy törlés után, hanem használjuk fel újra a már meglévőt.
- Memóriát és CPU időt is megtakaríthatunk vele, mert mindig ugyanazt az objektumot használjuk fel.

Flyweight a .NET osztályokban (String, Type)

- **String**-ek .NET-ben immutable-ek, vagyis nem lehet létrehozás után módosítani.
 - o Gyorsítótárba helyezi újrafelhasználás céljából.
 - o Tehát megnézi, hogy van-e már egy ugyanilyen értékű létező String a String pool-ban, ha van, akkor nem jön létre új String, hanem a meglévő String-re való hivatkozás kerül vissza.
- A **Type** osztály egy objektum típusát reprezentálja és minden típusnak egyedi identitása van egy AppDomain-en belül.
 - o Típusokat metaadatokból tölti be és a típus metaadatai az újrafelhasználás miatt gyorsítótárba kerülnek.
 - o Ezek a metaadatok tartalmazzák a típus nevére, névterére, attribúumaira és member-ekre vonatkozó információkat.
 - o Tehát a gyorsítótárazott metaadatokat adja vissza, ahelyett, hogy a metaadatokat újratöltené a lemeről.