

Gang-of-Four tervezési minták 5

Open/Closed elv

- Egy osztály legyen nyitott a bővítésre és a zárt módosításra, vagyis nem írhatunk bele, de származtathatunk tőle.
- **Ha nem követjük, akkor:**
 - o Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetők.
 - o Leszármazott megírásakor módosítani kell az őosztályt, ami tilos.
 - o Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában.

Liskov behelyettesíthetőség elve

- Őosztály helyett utódpéldány legyen mindig használható.
- Compiler supported, hiszen OOP elv (polimorfizmus)
- Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.

OOP virtuális metódusok

Késői kötés

- A metódus meghívásakor a referencia által hivatkozott objektum valódi típusának megfelelő osztály metódusa hívódik meg.
- A fordító nem tudhatja, hogy a meghívás pillanatában milyen típusú objektumra fog hivatkozni a referencia, így olyan kódot generál, ami futás közben dönti el, hogy melyik metódust hívja meg, ez a dinamikus kötés.
- **Előnyei:** Rugalmasság, polimorfizmus lehetősége
- **Hátrányai:** Teljesítményvesztés, tárigény, bonyolult

Virtuális metódus

- Minden meghívásakor késői kötetést fog használni (**virtual**).
- Leszármazottakban felül lehet definiálni (**override**), ilyenkor a késői kötésnek megfelelően fog a megfelelő metódus lefutni.

VMT – Virtuális Metódus Tábla

- Egy objektum VMT táblázata tartalmazza a dinamikusan kötött metódusok címeit.
- Egy metódus meghívásakor a VMT alapján dönthető el futásidőben, hogy melyik megvalósításnak kell lefutnia.
- Azonos osztályba tartozó objektumoknál ez mindig azonos, ezt megszokták osztani és minden objektum csak egy mutatót tárol erre a táblázatra.

Kompozíció

Aggregáció

- Az asszociáció speciális esete, tartalmazási kapcsolat.
- A tartalmazó osztály példányai magukba foglalják a tartalmazott osztály egy vagy több példányát, ez a rész-egész kapcsolat.
- A tartalmazó és a tartalmazott osztály egymástól függetlenül létezhetnek.
- A tartalmazás lehet, erős illetve gyenge.
- **Erős aggregáció:** A részek élettartalma szigorúan megegyezik az egészével, ez a kompozíció.
- **Gyenge aggregáció:** Egyszerű/általános aggregáció.



Kompozíció

- Másnéven **erős aggregáció**, tehát szigorú tartalmazási kapcsolat.
- Egy rész objektum csak egy egészhez tartozhat.
- **A tartalmazó és a tartalmazott élettiklusa közös:** Például van egy User objektum, aminek van egy Address tulajdonsága. Ha a User objektum megszűnik, akkor megszűnik az Address is, de nem létezhet User objektum Address nélkül. Ezért közös az élettiklusuk.



Strategy (Behavioral pattern)

- Algoritmusok egy csoportján belül az egyes algoritmusok egységbezárása és egymással kicserélhetővé tétele.
- A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetőek.
- **Probléma**
 - o Készítünk egy navigációs alkalmazást, ahol időről időre új funkciókat szeretnénk lefejleszteni.
 - o Útválasztási algoritmusokat akarunk fejleszteni autókhoz, sétához, stb.
 - o Átláthatatlan lett a kód, mert amikor hozzáadunk mindig egy új útválasztási algoritmust, megduplázódott a kód.
- **Megoldás**
 - o Adott algoritmust szervezzük ki egy külön osztályba, ezt nevezzük **stratégiának**.

Strategy használjuk, ha

- Egy objektumon belül egy algoritmus különböző változatait szeretnénk használni és képesnek kell lennie a változásra az egyik algoritmusról a másikra futás (runtime) közben.
- Sok hasonló osztály van, amik csak abban különböznek egymástól, hogy hogyan hajtanak végre bizonyos viselkedést.
- Business Logic elkülönítése

Strategy implementálása

1. Context osztályban azonosítsunk be egy olyan algoritmust, ami hajlamos a gyakori változásokra.
2. Strategy interfész deklarálása
3. Egyenként implementáljuk a Strategy interfészt a megfelelő algoritmussal a saját osztályába.
4. Context osztályban legyen egy field, ami tárolja a strategy objektum referenciáját tárolja.
 - a. Setter-t is állítsunk neki

Strategy előnyök és hátrányok

- **Előnyök**
 - o Objektumon belül algoritmust tudunk cserélni futásidőben. (runtime)
 - o Az algoritmus implementálását izolálhatjuk az azt használó kódtól.
 - o Öröklődést kicserélhetjük kompozícióval.
 - o Open/Closed elv
- **Hátrányok**
 - o Ha csak pár algoritmus van és azok is ritkán változnak, akkor nem érdemes túlbonyolítani új algoritmusokkal, osztályokkal és interfészekkel, amik a mintával együtt járnak.

Template (Behavioral)

- Egy műveleten belül algoritmus vázat definiál és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
- **Probléma**
 - o Készítünk egy olyan alkalmazást, amivel különböző dokumentumokból adatokat lehet kinyerni.
 - o Egy idővel rájövünk, hogy például a PDF, CSV fájlok között viszonylag hasonló műveletek hajtódnak végre, így kód duplikáció keletkezhet.
- **Megoldás**
 - o Magát az algoritmust bontsuk szét kisebb lépésekre, metódusokra.
 - o Ezeket fogjuk meghívni a template method-ban.

Template használjuk, ha

- Algoritmust kisebb lépésekre szeretnénk bontani.
- Logikai hasonlóság esetén

Template implementálása

1. Kisebb részekre bontás
2. Absztrakt osztály létrehozása, ahol deklaráljuk a template method-ot.
3. Hívjuk meg az alosztályokat, a lépéseket a template method-ban.

Template előnyök és hátrányok

- **Előnyök**
 - o Kód duplikáció elkerülhető vele, tehát a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg.
 - Leszármazott osztályban felül lehet definiálni.
- **Hátrányok**
 - o Megsérthetjük a Liskov behelyettesítési elvet, ha egy alosztályon keresztül elnyomja az alapértelmezett lépés implementációját.
 - o Template method-okat egy idő után nehéz karbantartani, ha sok kisebb lépést (metódusokat) tartalmaz.

Adapter (Structural pattern)

- Egy osztály interfészét olyan interfésszé konvertálja, amit a kliens vár.
- Lehetővé teszi olyan osztályok együttműködését, amik egyébként inkompatibilis interfészeik miatt nem tudnának együttműködni.
- **Probléma**
 - o Össze szeretnénk kötni két rendszert, amik nem kompatibilisek.
 - Például van egy alkalmazás, ami XML formátummal működik és szeretnénk használni egy másik csomagot, ami csak JSON formátummal működik.
- **Megoldás**
 - o *(Valós példa: Adapter kábelek: VGA -> HDMI és vissza)*
 - o Készítünk egy adapter-t, ami elrejt magát a konverziót.

Adapter használjuk, ha

- Egy olyan osztályt szeretnénk használni, aminek interfésze nem megfelelő Adapter.
- Egy újrafelhasználható osztályt szeretnénk készíteni, ami együttműködik előre nem látható vagy független szerkezetű osztályokkal. (pluggable adapters)

Adapter implementálása

1. Adapter osztály elkészítése
2. Az adapter osztályban adjunk hozzá egy field-et, ami referenciaként rámutat a service objektumra.
3. Kliens interfész metódusainak implementálása az adapter osztályban.
4. Hajtsuk végre magát a konverziót az adapter segítségével a két nem kompatibilis interfész között.

Adapter előnyök és hátrányok

- **Előnyök**
 - o Single Responsibility elv
 - o Open/Closed elv
- **Hátrányok**
 - o Komplexitás növekedhet minden egyes új osztálynál és interfésznél.

Bridge (Structural pattern)

- Különválasztja az absztrakciót (interfészt) az implementációtól, hogy egymástól függetlenül lehessen őket változtatni.
- **Probléma**
 - o Egy osztály két jellemzőtől is függ
 - o Például alakzatok, szín és forma
- **Megoldás**
 - o Szét kell bontani az osztályt
 - o A forma osztály várja interfészen keresztül a szín osztályt
 - o Kompozícióval lehessen összeépíteni őket

Bridge használjuk, ha

- Egy osztályt több ortogonális (független) dimenzióban kell bővíteni.
- Futás közben implementációt szeretnénk váltani

Bridge implementálása

1. Bridge interfész létrehozása.
2. Bridge osztály létrehozása, ami implementálja a Bridge interfészt.
3. Abstract osztály létrehozása
4. Konkrét osztály létrehozása, ami implementálja az Abstract osztályt

Bridge előnyei és hátrányai

- **Előnyök**
 - o Absztrakció és az implementáció különválasztása
 - o Az implementáció dinamikusan, akár futási időben is megváltoztatható
 - o Az implementációs részletek a klienstől teljesen elrejtethetők
 - o Az implementációs hierarchia külön lefordított komponensbe tehető, így ha ez ritkán változik, nagy projekteknél nagymértékben gyorsítható a fordítás ideje
 - o Ugyanaz az implementációs objektum több helyen is felhasználható
- **Hátrányok**
 - o Bonyolulttá válhat a kód egy idő után

Dependency Injection elve és megvalósítása

- Lazán csatoltság kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.
- **Probléma**
 - o A kód függjön absztrakciótól, ne konkrét implementációtól.
- **Megoldás**
 - o Az interfészt várjuk paraméterként a konstruktorban.
 - o Setter injektálás, amikor az objektumokat setter metódusok segítségével injektáljuk.