S.O.L.I.D. elvek

Vastag Atila

2024

Objektum orientáltan programozni könnyű. Egy kis osztály itt, egy kis öröklődés ott, interfészek, virtuális metódusok, mifene… Aztán néhány ezer sorral később nyakig ülünk a sz….n, mert egy osztályt ki kellene egészíteni egy apró funkcióval, csakhogy szegényke úgy bele van betonozva a hierarchiába, hogy ember legyen a talpán aki egymáshoz igazítja a részleteket.

Kellene tehát valami, ami kifogja a szelet a káosz vitorlájából és megzabolázza az osztályszerkezetet, hogy úgy viselkedjen ahogyan azt elvárjuk. Szerencsére a világban sok okos ember van, közülük pedig Robert C. Martin a.k.a. Uncle Bob volt az aki felhúzta az acélbetétes bakancsot és a világra szabadította a S.O.L.I.D.–ot, ezt a kedves kis betűszót, amelynek minden karaktere egy-egy alapelvet jelképez (ez egyébként vicces: egy betűszó, ami betűszavakból áll).

A S.O.L.I.D. elvek meglepően egyszerűek, de éppen ebben rejlik a nagyszerűségük: öt apró szabályt kell betartanunk (ez néha nem is olyan könnyű), mégis hamar érezni fogjuk a különbséget.

- **Single Responsibility Principle (Egy felelősség elve)**
- **Open/Closed Principle (Nyílt/zárt elv)**
- **Liskov substitution principle (Liskov helyettesítési elv)**
- **Interface segregation principle (Interface elválasztási elv)**
- **Dependency inversion principle (Függőség megfordítási elv)**

Single Responsibility principle

Egy felelősség elve. Egy osztály csak egy felelősséggel rendelkezzen.
A kérdés már csak az, hogy mit nevezünk felelősségnek? Ezt egy példán keresztül könnyebben meg lehet érteni. Tételezzük fel, hogy van egy adatokat leíró osztályunk és az adatokat be kellene tudnunk olvasni valamilyen formátumban. Adódhat az elképzelés, hogy a fájlbeolvasást metódusként megvalósítjuk az osztályban.
Ez azért problémás, mert így a kód valójában két dolgot csinál: értékeket tárol és beolvas. Ha a későbbiekben ez a funkcionalitás bővül exportálási lehetőséggel, akkor már három dolog van egy helyen implementálva. Ez pedig előbb-utóbb elkerülhetetlenül azt eredményezi, hogy az egyik metódus módosítása ki fog hatni a másikra és szépen csendben akár képes eltörni a funkcionalitást, ami csak futásidőben derül ki.
Arról, hogy egy adott osztály teljesíti-e a Signgle Responsibility elvet úgy tudunk megbizonyosodni, hogy mikor magyarázzuk az osztály/metódus szerepét és kötőszavakat kell használnunk, (és, illetve, továbbá, meg még és szinonimáik), akkor biztos, hogy nem single responsible.

Open/Closed principle

Nyílt/zárt elv. Egy osztály legyen nyílt a kiterjesztésre, de zárt a módosításra.

Tételezzük fel, hogy van egy osztályunk, aminek definiáltuk a publikus részeit. Az osztályunk nyílt a kiterjesztésre, mert örököltethetünk belőle újabb funkciók beletételével, de csak akkor lesz zárt a módosításra, ha az eredeti osztály publikus tagjait később nem változtatjuk.

Nézzünk egy példát:

Open/Closed principle

```
class Pelda
{
    public void CsinalValamit()
    {
        //kód
    }
}

class Gyerek : Pelda
{
    public void CsinalValamit()
    {
        // Open closed sértés és egyben Liskov elvnek sem felel meg.
    }
}
```

A fenti példában a **CsinalValamit()** metódus a **Pelda** osztályban nem virtuális metódusként van szerepeltetve, vagyis nem írhatnánk felül a működését. Azonban a polimorfizmusnak köszönhetően a Gyerek osztályban is definiálhatunk egy **CsinalValamit()** metódust. Ha ilyesmit csinálunk, akkor megsértjük az Open/Closed elvet, mivel az eredeti **Pelda** osztályban a metódus nem volt felülírhatónak jelölve.

Ha szó szerint vesszük az Open/Closed elvet, akkor ha egy alap osztály funkcionalitását bővítjük örököltetés helyett, akkor már sértjük az Open/Closed elvet. De mint az életben, ez a döntés sem csupán fekete és fehér kérdése: minden esetben mérlegeljük a döntésünket és ha lehet, akkor törekedjünk az öröklés észszerű és jó használatára!

Liskov substitution principle

Liskov helyettesítési elv. Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna.

Ez röviden és tömören azt jelenti, hogy ha örököltetünk egy osztályból és felüldefiniáljuk az osztály bizonyos részeit, akkor ne implementáljunk az ősosztálytól radikálisan eltérő logikát a felüldefiniált tagokban.

Ez alatt azt értem, ha az ősosztály rendelkezik egy **ToString()** metódussal, akkor a leszármazott osztályban is csak azt csinálja, amit az ősosztályban elvárunk. Ne valósítson meg belső állapot módosítást és egyéb, a nevéből és az eredeti szándékából levezethető viselkedést.

Továbbá ebbe beleértendő az is, hogy ha egy, az osztályunk által megvalósított interfész egy metódust/tulajdonságot definiál, akkor a megvalósítást tartalmazó osztályban az implementációnak nem szabad NotImplementedException kivételt kiváltania. Ugyanebbe a szabályba értendő az is, ha az ősosztály definiál egy virtuális vagy absztrakt metódust/tulajdonságot, akkor a leszármazott osztályoknak nem szabad az öröklési láncot megszakítaniuk.

Liskov substitution principle

```
abstract class OsOsztaly
{
    public virtual void Valami()
    {

    }
}

class Leszarmazott: OsOsztaly
{
    public new void Valami()
    {
        //öröklés megszakítva! Ilyet ne csináljunk.
    }
}
```

```
interface Ipelda
{
    void PeldaMetodus();
}

class Implementacio: Ipelda
{
    public void PeldaMetodus()
    {
            //szintén Liskov sértés
            throw new NotImplementedException();
    }
}
```

Interface segregation principle

Interfész elválasztási elv. Több specifikus interfész jobb, mint egy általános.

Ha a kódunkat sok kicsi interfésszel valósítjuk meg, akkor elérhetjük azt, hogy a felületet felhasználó osztálynak ténylegesen csak ahhoz lesz hozzáférése, amire szüksége van. Ezzel csökkentjük a hibalehetőségeket.

Ennek az elvnek nem kicsit köze van a Single Responsibility-hez. A single responsibility elsősorban az implementációk (tényleges osztályok) kérdésével foglalkozik, míg az interface segregation az implementációk publikus felületével.

Tételezzük fel, hogy egy moduláris alkalmazást készítünk. Ebben a moduloknak szeretnénk egy közös csatolófelületet biztosítani, amin keresztül mondjuk nyomtathatnak és fájlokat kezelhetnek. Kézenfekvő lenne ilyen módon implementálni az interfészt:

Interface segregation principle

```
interface API
{
    void FajltMegnyit(string fajlnev);
    void Ment(string fajlnev);
    void Nyomtat();
    void NyomtatasiElonezet();
}
```

Ha a single responsibility esetén tanultakat alkalmazzuk, akkor nyilvánvaló, hogy nem egy felelőssége van, mivel kell egy mágikus *ÉS* szót alkalmaznunk: nyomtat *ÉS* fájlokat kezel. Ugyanakkor Interface segregation-t is sértünk, mivel interfészekről beszélünk. Helyesebb megvalósítás:

```
interface FajlApi
{
    void FajltMegnyit(string fajlnev);
    void Ment(string fajlnev);
}
```

```
interface NyomtatApi
{
    void Nyomtat();
    void NyomtatasiElonezet();
}
```

Dependency inversion principle

Függőség megfordítási elv. A kódod függjön absztrakcióktól, ne konkrét implementációktól. Vagyis, ha az osztályunknak szüksége van egy másik osztályra a működéséhez, akkor ne a konkrét osztálytípust várja függőségnek, hanem egy interfészt, amit a függőségosztály megvalósít.

Nézzünk egy példát. Tételezzük fel, hogy van egy osztályunk **Foo**, ami működéséhez egy másik osztályt, a **Bar**-t használja fel.

```
class Foo
{
    private Bar _bar;

    public Foo()
    {
        _bar = new Bar();
    }
}
```

```
class Foo
{
    private Bar _bar;

    public Foo()
    {
        _bar = new Bar();
    }
}
```

A probléma ezzel a kódrészlettel az, hogy **Foo** osztály egyetlen példánya sem létezik **Bar** nélkül. Vagyis, ha a **Bar** osztály módosul, akkor az indirekt módon kihat a **Foo** osztály működésére is, ami azt eredményezheti, hogy mindkét osztály funkcionalitása eltörik. Éppen ezért szerencsésebb lenne a fenti példában, ha a **Foo** osztály nem közvetlenül függne **Bar** osztálytól, hanem mondjuk egy absztrakciójától.

Dependency inversion principle

```
interface IBar
{
    void Publikus();
}

class Foo
{
    private IBar _bar;

    public Foo(IBar bar)
    {
        _bar = bar;
    }
}
```

Ennek a megoldásnak az az előnye, hogy két komponens közötti interakció jól definiált egy interfészen (esetlegesen absztrakt osztályon) keresztül, illetve tesztelés esetén helyettesíthető az **IBar** tetszőleges implementációval, ami nagymértékben megkönnyíti a tesztek írását és a hibák feltárását.