Szoftvertechnológia és-technikák
8. Gyakorlat

Observer és singleton tervezési minták

A gyakorlat menete

A gyakorlat során két programozási feladatot kell megoldani, melyek az Observer és a Singleton tervezési minták használatát mutatják be. Az első feladatban vezetett módon több lépésben, míg a második feladatnál önállóan jutunk el a végső megoldáshoz. A gyakorlat során egy C# konzol alkalmazásban dolgozunk. Célszerű Visual Studioban egy .NET Core 3.1 (vagy annál újabb), C# nyelvű Console alkalmazást létrehozni.

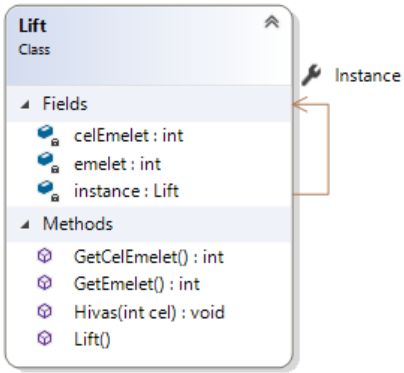
Gyakorlatvezetőknek: A példák úgy vannak felépítve, hogy csak a legszükségesebb mennyiségű gépelést igényeljék. Ha az idő engedi, a megértés segítésére lehetőség szerint minden kódot kézzel gépeljünk be, ne másoljunk ki az útmutatóból kódrészleteket.

Közös (vezetett) feladat

A feladat során egy emeletes ház liftvezérlését fogjuk elkészíteni 6 lépésben. A követelmények lépésről lépésre változni fognak, melyek során szükség lesz korábbi megoldásaink felülbírálására is.

1. lépés: Singleton

Követelmény: *Készítsen egy Lift osztályt, mely egy emeletes ház felvonóját reprezentálja! Az osztály rendelkezik egy Hivas(int) függvénnyel, melyet meghívva a lift az aktuális emeletet másodpercenként 1-gyel növelve/csökkentve addig módosítja, amíg el nem éri a célemeletet. Az aktuális emelet lekérdezését a GetEmelet() a célemelet lekérdezését a GetCelEmelet() függvények valósítják meg. Az osztályból egyetlen egy példányt lehessen létrehozni, melyet a programunk bármely pontjából elérhetünk!*



Megjegyzés: Figyeljük meg, hogy az osztályaink és függvényeink neveiben vegyesen használunk magyar szavakat és angol terminológiát. Bár a legtöbbször célravezető következetesen angol elnevezéseket használni a kódunkban, egyes szakterületeken néha segítheti a megértést, ha nem angolosítjuk az egyébként magyarul használt kifejezéseket. A mai példák ezt a vegyes használatot illusztrálják.

Az utolsó követelmény megvalósítására a Singleton minta szolgál. Hozzunk létre egy új osztályt Lift néven és első lépésként valósítsuk meg a mintát. Emlékeztetőül a 7. előadásból: „**A Singleton minta biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez az egy példányhoz globális hozzáférést biztosít**”.

Lift.cs

```
class Lift
{
    // Az egyetlen példányt statikus tagváltozóban tároljuk, alapból inicializáljuk
    private static Lift instance = new Lift();
    // A példányt csak olvasható módon érjük el
    public static Lift Instance { get { return instance; } }
    // A privát konstruktor biztosítja, hogy kívülről ne lehessen példányokat létrehozni
    private Lift(){}
}
```

Elemezzük a minta működését…

- Az egyetlen példányt maga az osztály tárolja egy statikus, védett (private) változóban
- *instance* mező értékét közvetlenül inicializáljuk. Mivel statikus mezőről van szó, ez a logika osztály első elérésekor fog lefutni
- Az egyetlen példányt a többi osztály egy statikus, Instance nevű tulajdonságon keresztül éri el (ez lehetne egy statikus GetInstance függvény is, a mostani megoldásunk közelebb van a .NET konvenciókhoz)
- Az osztály konstruktora védett (private), így más osztályok nem tudják a new operátorral példányosítani. Ez garantálja, hogy más osztály nem tudja megkerülni az *Instance* használatát, így további példányokat nem tud létrehozni.

…majd egészítsük ki az osztályt a tényleges funkciókkal:

Lift.cs

```
private int emelet = 1;
private int celEmelet = 1;

public int GetEmelet()
{
    return emelet;
}
public int GetCelEmelet()
{
    return celEmelet;
}
public void Hivas(int cel)
{
    this.celEmelet = cel;
    // Irány meghatározása
    int lepes = emelet > cel ? -1 : 1;

    while (emelet != cel)
    {
        emelet += lepes;
        // 1 másodpercig blokkoljuk a végrehajtást
        Thread.Sleep(1000);
    }
}
```

Végül egészítsük ki a Main függvényünket az osztályunk használatával:

Program.cs

```
public static void Main(string[] args)
{
    Console.WriteLine(Lift.Instance.GetEmelet());
    Lift.Instance.Hivas(5);
    Console.WriteLine(Lift.Instance.GetEmelet());
}
```

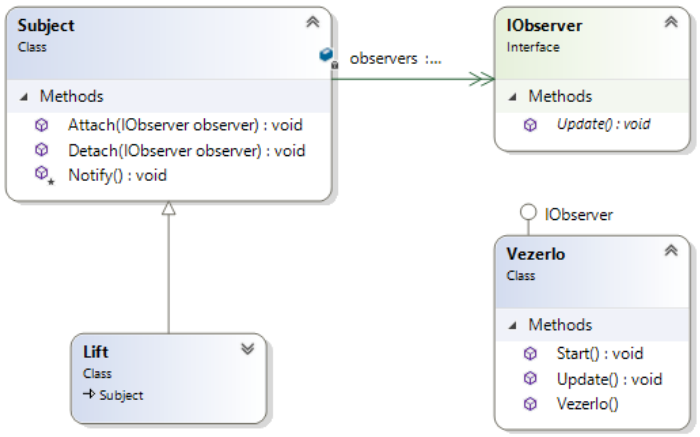
Futtassuk le a programot! A program kiírja, hogy az első emeleten vagyunk, majd 5 másodperccel később láthatjuk, hogy felértünk az 5. emeletre.

2. lépés: Observer

Követelmény: *Liftünket egy központi irányítóközpontból fel és le küldhetjük. Készítsünk el ehhez egy Vezerlo osztályt, mely a felhasználótól újból és újból bekér egy számot 1 és 5 között, majd azt a liftnek megadja célemeletként! A vezérlő folyamatosan jelenítse meg az aktuális emeletet és a célemeletet a következő formátumban: „Vezérlés: aktuális_emelet->célemelet”, ahol az aktuális_emelet és a célemelet behelyettesítendő az aktuális értékekkel. A vezérlés a lift mozgásáról az Observer minta alapján értesüljön.*

Az Observer minta „**lehetővé teszi, hogy egy objektum (subject/alany) értesítést küldjön más objektumoknak (observer/megfigyelő) az állapotának változásairól. Mindezt anélkül, hogy az alany függene a megfigyelők konkrét típusától**.”. A mintáról részletesebb információkat a 7. előadás fóliái tartalmaznak

Első körben itt is a tervezési minta alaposztályait hozzuk létre. Hozzuk létre az alábbi osztályokat:



IObserver.cs

```
public interface IObserver
{
    void Update();
}
```

Figyeljük meg, hogy az Observer esetében nincs előre definiált működés (mint ami a Subject-nél lesz), csupán tudnia kell reagálnia a Subject értesítéseire. Ezt a követelményt ősosztály helyett akár egy interfésszel is kielégíthetjük. Az interfész használatának az előnye, hogy később olyan osztályokat is

Observerekké tehetünk, melyeknek már van ősosztálya. Ez utóbbi gyakori lehet különböző GUI keretrendszerek használatakor.

Subject.cs

```
public class Subject
{
    private List<IObserver> observers = new List<IObserver>();
    public void Attach(IObserver observer)
    {
        observers.Add(observer);
    }
    public void Detach(IObserver observer)
    {
        observers.Remove(observer);
    }

    protected void Notify()
    {
        foreach (var observer in observers)
        {
            observer.Update();
        }
    }

}
```

Egészítsük ki a Lift osztályunkat úgy, hogy a Subject-ből származzon, illetve változásairól kiértesítse a feliratkozott Observereket:

Lift.cs

```
class Lift : Subject
{
    public void Hivas(int cel)
    {
        this.celEmelet = cel;
        int lepes = emelet > cel ? -1 : 1;

        while (emelet != cel)
        {
            emelet += lepes;
            Notify();
            Thread.Sleep(1000);
        }

    }
}
```

Hozzuk létre a Vezerlo osztályt és implementáljuk benne az IObserver interfészt.

Vezerlo.cs

```
class Vezerlo : IObserver
{
    public Vezerlo()
    {
        Update();
    }
    public void Update()
    {
        int liftEmelet = Lift.Instance.GetEmelet();
        int celEmelet = Lift.Instance.GetCelEmelet();
        Console.SetCursorPosition(0, 11);
        Console.Write($"Vezérlés: {liftEmelet}->{celEmelet}");
    }

    public void Start()
    {
        // A kurzort a képernyőn a 10. sor elejére mozgatjuk
        Console.SetCursorPosition(0, 10);
        Console.WriteLine("Hová szeretnél menni?");
        // Beolvassuk a következő billenytyű lenyomást
        var key = Console.ReadKey(true);
        // Ha nem 1,2,3,4,5 karakter valamelyikét kapjuk, megszakítjuk a futást
        while ("12345".Contains(key.KeyChar))
        {
            Lift.Instance.Hivas(int.Parse(key.KeyChar.ToString()));
            key = Console.ReadKey(true);
        }
        Console.WriteLine("Viszlát!");
    }
}
```

Végül módosítsuk a Main függvényt:

Program.cs

```
public static void Main(string[] args)
{
    var vezerlo = new Vezerlo();
    Lift.Instance.Attach(vezerlo);
    vezerlo.Start();
}
```

Futtassuk le a programot! Az 1-5 számok megnyomásával irányíthatjuk a liftet. A debugger használatával kövessük végig az értesítési láncot!

3. lépés: Liftajtó

*Követelmény: Épületünk minden emeletén van egy liftajtó. A liftajtó kijelzője mindig a lift aktuális emeletét mutatja, kivéve, amikor a lift az adott liftajtó szintjére érkezik, ilyenkor ugyanis egy * jelenik meg a kijelzőn. Valósítsuk meg a Liftajto osztályt és illesszük be programunkba!*

A liftajtó megvalósítása **nem fogja igényelni meglévő osztályaink módosítását, hiszen a liftünk fel van készülve a változásértesítésre**. Az új osztályunknak így elég az IObserver interfészt megvalósítania és kezelnie az aktuális szint kijelzését. Készítsük el a Liftajto osztályt!

Liftajto.cs

```
public class Liftajto: IObserver
{
    private int emelet;

    public Liftajto(int emelet)
    {
        this.emelet = emelet;
        Update();
    }
    public void Update()
    {
        int liftEmelet = Lift.Instance.GetEmelet();
        // A liftajtót az emeletének megfelelő sorba rajzolunk ki
        Console.SetCursorPosition(0, emelet);
        if (emelet == liftEmelet)
            Console.Write($"{emelet}: [*]");
        else
            Console.Write($"{emelet}: [{liftEmelet}]");
    }
}
```

A Main függvényünk most már kicsit hosszabb lesz, hiszen itt kell elvégeznünk a környezet inicializálását.

Program.cs

```
public static void Main(string[] args)
{
    var a1 = new Liftajto(1);
    var a2 = new Liftajto(2);
    var a3 = new Liftajto(3);
    var a4 = new Liftajto(4);
    var a5 = new Liftajto(5);
    var vezerlo = new Vezerlo();

    Lift.Instance.Attach(a1);
    Lift.Instance.Attach(a2);
    Lift.Instance.Attach(a3);
    Lift.Instance.Attach(a4);
    Lift.Instance.Attach(a5);
    Lift.Instance.Attach(vezerlo);

    vezerlo.Start();
}
```

Futtassuk le a programot! Az 1-5 számok megnyomásával irányíthatjuk a liftet.

4. lépés: GetData() helyett tulajdonságok

Korábbi példáinkban betű szerint ragaszkodtunk az Observer mintához, ezért hoztuk létre a Lift osztályban a Get* függvényeket. A gyakorlatban az Observer minta megvalósítását igazítani szoktuk az adott platform specialitásaihoz. .NET-ben a privát mezők elérésére nem getter metódusokat, hanem tulajdonságokat (property) szoktunk használni. Ennek megfelelően töröljük a GetEmelet() és GetCelEmelet() függvényeket és helyettük vezessünk be két tulajdonságot a Lift osztályban.

Lift.cs

```
public int Emelet { get; set; } = 1;
public int CelEmelet { get; set; } = 1;

private int emelet = 1;
private int celEmelet = 1;

public int GetEmelet()
{
    return emelet;
}
public int GetCelEmelet()
{
    return celEmelet;
}
```

Módosítsuk a Lift osztály és a többi osztályunk kódját, hogy az új tulajdonságokat használják! Ilyenkor a lokális változók használatára sem feltétlenül van szükség. A Vezerlo osztály Update metódusa pl. így változik:

Vezerlo.cs

```
public void Update()
{
    Console.SetCursorPosition(0, 11);
    Console.Write($"Vezerles: {Lift.Instance.Emelet}->{Lift.Instance.CelEmelet}");
}
```

Mivel a régi függvényeket töröltük, fordítási hibát fogunk kapni addig, amíg az összes osztályunkon keresztül nem vezettük a változtatást.

Futtassuk le a programot! A programunk funkcionalitása változatlan, hiszen a változtatásaink nem módosították a működést, még az adatáramlás irányát sem. A platformkonvenciók követése által ugyanakkor más .NET programozók számára egyszerűbb, olvashatóbb lett a kódunk (ennek jó indikátora az is, hogy csökkent a kódsorok száma, kompaktabb lett a kódunk).

5. lépés: Több lépcsőház

Követelmény: Épületünk két független lépcsőházat (és lépcsőházanként 5 liftajtót) tartalmaz, lépcsőházanként egy-egy lifttel. A két lift külön vezérelhető. Módosítsa a kódját a megváltozott követelmények szerint!

A fenti változtatás legfőbb következménye, hogy fel kell számolnunk a Singleton mintánkat, hiszen mostantól a Liftből is több példányra lesz szükségünk. Ezzel együtt a Lift osztályba bevezetünk egy tulajdonságot a lépcsőház tárolására.

Lift.cs

```
class Lift : Subject
{
    public int Emelet { get; set; } = 1;
    public int CelEmelet { get; set; } = 1;
    public int Lepcsohaz { get; set; } = 1;

    private static Lift instance = new Lift();
    public static Lift Instance { get { return instance; } }

    public Lift()
    {
    }
    …
}
```

A `Liftajto` és `Vezerlo` osztályok mostantól nem Singletonként érik ez a `Lift`-et, hiszen több `Lift` objektum is létezik a rendszerben. Helyette mindkét osztály a kontruktorában a konstruktorában kapja meg azt a `Lift` objektumot, mellyel együttműködik, és ezt el is tárolja egy tagváltozóban. Ezen felül egészítsük ki `Liftajto` és `Vezerlo` osztályainkat úgy, hogy a lépcsőháztól függően más-más pozícióba írják ki az üzeneteiket.

Liftajto.cs

```
class Liftajto : IObserver
{
    private int emelet;
    private Lift lift;

    public Liftajto(int emelet, Lift lift)
    {
        this.emelet = emelet;
        this.lift = lift;
        this.lift.Attach(this);
        Update();
    }
    public void Update()
    {
        int liftEmelet = Lift.Instance.Emelet;
        Console.SetCursorPosition(lift.Lepcsohaz*20, emelet);
        if (emelet == lift.Emelet)
            Console.Write($"{emelet}: [*]");
        else
            Console.Write($"{emelet}: [{lift.Emelet}]");
    }
}
```

Vezerlo.cs

```
class Vezerlo : IObserver
{
    private Lift lift;

    public Vezerlo(Lift lift)
    {
        this.lift = lift;
        lift.Attach(this);
        Update();
    }
```

```
    public void Update()
    {
        Console.SetCursorPosition(lift.Lepcsohaz * 20, 11);
        Console.Write($"Vezerles: {lift.Emelet}->{lift.CelEmelet}");
    }
   …
}
```

A `Vezerlo` osztály `Start` függvényét nem fogjuk már haszni, ki is kommentezhetjük: a liftek hívását az egyszerűség érdekében a `Lift` osztály `Hivas()` műveletével kódból fogjuk a többliftes környezetben megtenni.

A környezet incializálása már kifejezetten összetett lesz.

Program.cs

```
public static void Main(string[] args)
{
    // Liftek
    var lift1 = new Lift() { Lepcsohaz = 1 };
    var lift2 = new Lift() { Lepcsohaz = 2 };

    // 1. lépcsőház
    var a1 = new Liftajto(1, lift1);
    var a2 = new Liftajto(2, lift1);
    var a3 = new Liftajto(3, lift1);
    var a4 = new Liftajto(4, lift1);
    var a5 = new Liftajto(5, lift1);
    var vezerlo1 = new Vezerlo(lift1);

    // 2. lépcsőház
    var b1 = new Liftajto(1, lift2);
    var b2 = new Liftajto(2, lift2);
    var b3 = new Liftajto(3, lift2);
    var b4 = new Liftajto(4, lift2);
    var b5 = new Liftajto(5, lift2);
    var vezerlo2 = new Vezerlo(lift2);

    // A 2. lépcsőház 3. emeletének liftajtó kijelzője „elromlott”
    lift2.Detach(b3);

    // Az input kezelés már bonyolultabb lenne, ezért innen
    // adunk pár utasítást a lifteknek, majd kilépünk
    lift1.Hivas(5);
    lift2.Hivas(5);
    lift1.Hivas(1);
    lift2.Hivas(1);
}
```

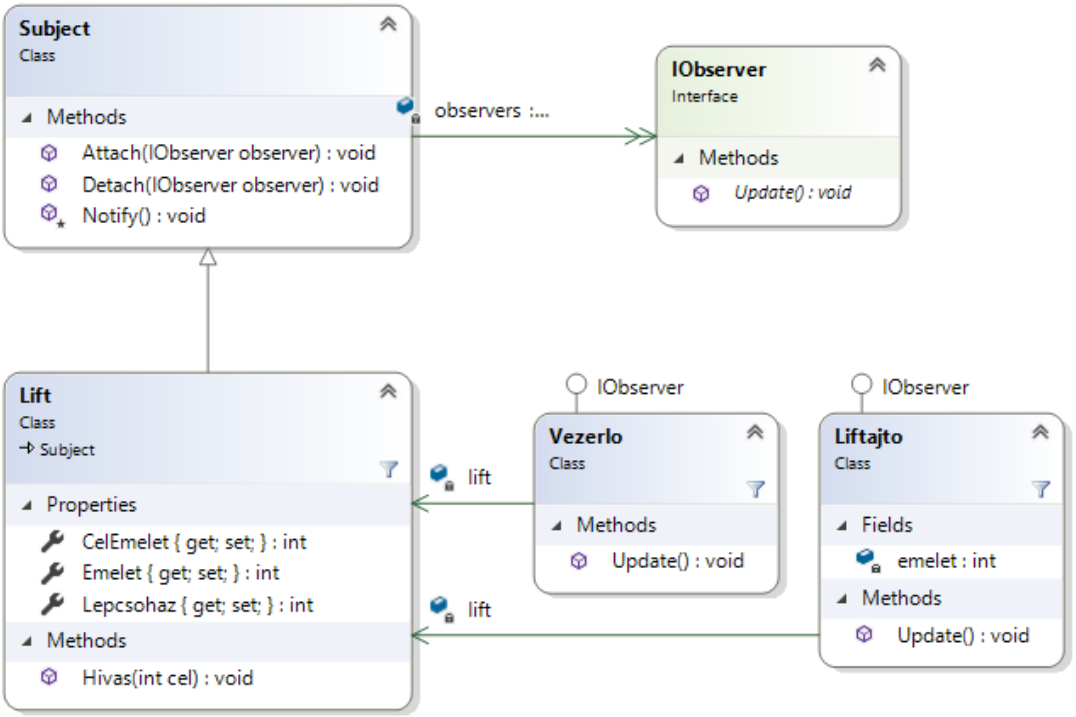
A fenti megoldás kapcsán két dolgot itt mindenképp meg kell jegyeznünk. Először is a tény, hogy az egyszerű, csupán saját feladataikról tudó osztályainkat egyetlen helyen fel tudjuk így konfigurálni és összedrótozni azt mutatja, hogy jól választottuk szét a felelősségi köröket. Ha pl. egy 7 emeletes, 3 lépcsőházas házat kéne modelleznünk, elég lenne ezen az inicializáló kódon módosítani, magukat az osztályokat nem érintené a változás.

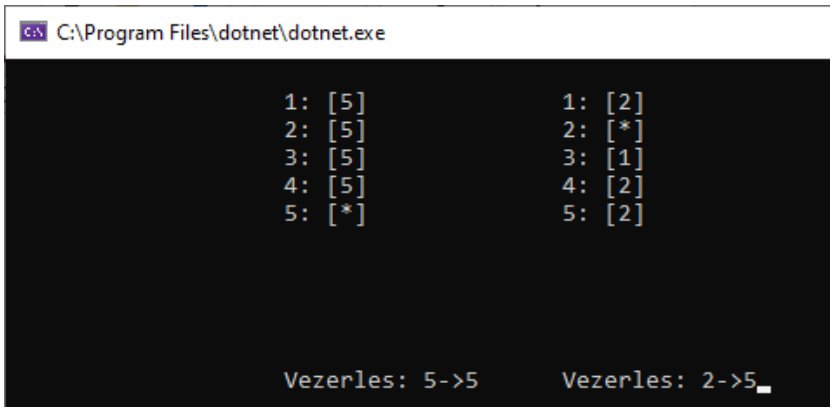
Szintén fontos leszögezni, hogy bár a labor szempontjából egy ilyen megoldás rendben van, ezt a típusú inicializáló kódot egy éles rendszerben (majd a házi is annak számít 😊) mindenképpen ki kéne szervezni egy osztályba, melynek **ez** a felelőssége (egységbe zárás elve). Az osztály neve pl.

LiftSystemModel, vagy pl. Building lehetne, de ez természetesen már a konkrét alkalmazástól is függene. Ez az osztály

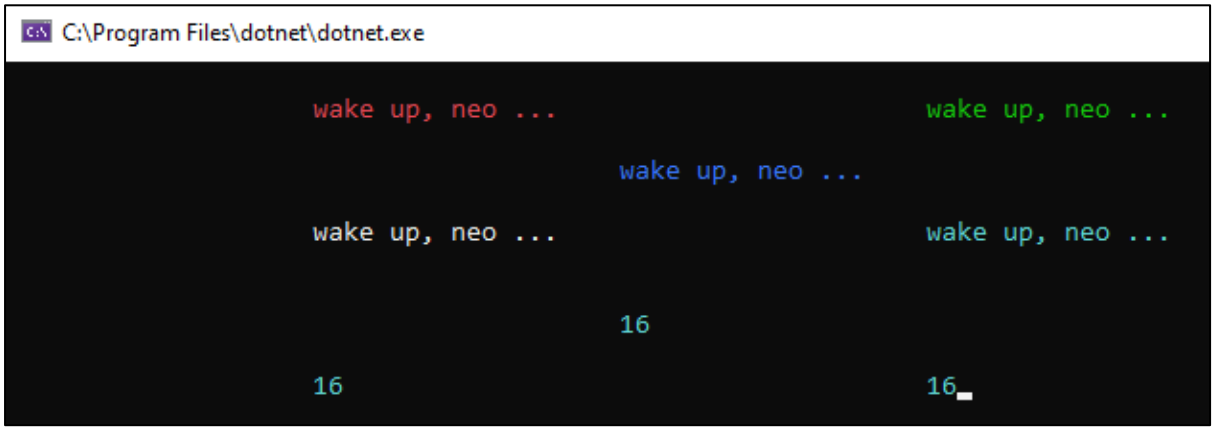
- Tagváltozókban tárolná a modell objektumokat (Lift, Vezerlo, Liftajto stb.)
- Tartalmazna egy Run() műveletet, ebben történne a liftek hívása, illetve a vezérlők inicializálása.

Futtassuk le a programot! A lenti diagrammot is felhasználva, foglaljuk össze, mit értünk el a feladat során!





Önálló feladat



Készítsük el önállóan az alábbi „Fényújság” konzolos alkalmazást. A fényújság lelke egy mindenhonnan elérhető egy példányban létező `Vezerlo` osztály, mely egy megjelenítendő szöveget tartalmaz. Ehhez kétféle kijelző is kapcsolódhat:

- `SzovegKijelzo`: megjeleníti a `Vezerlo` által tartalmazott szöveget, egy előre megadott színnel
- `HosszKijelzo`: megjeleníti a `Vezerlo` által tartalmazott szöveg hosszát

Mindkét kijelzőtípus konstruktorparaméterben megadott konzol pozícióban (sorban és oszlopban) és konstruktorparaméterben megadott színnel végzi a tartalma megjelenítését. A kijelzők tartalma mindig frissüljön, amikor a vezérlő szövege módosul. A változásértesítést az Observer tervezési minta alkalmazásával valósítsuk meg!

A `Vezerlo` osztály tartalmaz egy `Kiir(string)` függvényt. Miután valaki ezt a függvényt meghívja, a vezérlő másodpercenként frissíteni fogja a megjelenítendő szöveget a paraméterként kapottra, úgy, hogy először csak az első karaktert jeleníti meg, majd az első kettőt, első hármat … és így tovább addig, amíg a teljes szöveg nem látszik.

Amikor az alaposztályok elkészültek, állítsa össze a következő konfigurációt:

- 5 szöveges kijelző jelenjen meg az 1., 3. és 5. sorokban
- 3 hossz kijelző jelenjen meg a 8. és 10. sorokban

Az alkalmazás Main függvényében `Vezerlo` meghajtására a következő parancsokat használja

Program.cs

```
Vezerlo.Instance.Kiir("wake up, neo ...");
// k1 az egyik szövegkijelző
Vezerlo.Instance.Detach(k1);
Vezerlo.Instance.Kiir("The Matrix has you");
// k2 az egyik hosszkijelző
Vezerlo.Instance.Detach(k2);
Vezerlo.Instance.Kiir("Follow the white rabbit");
```

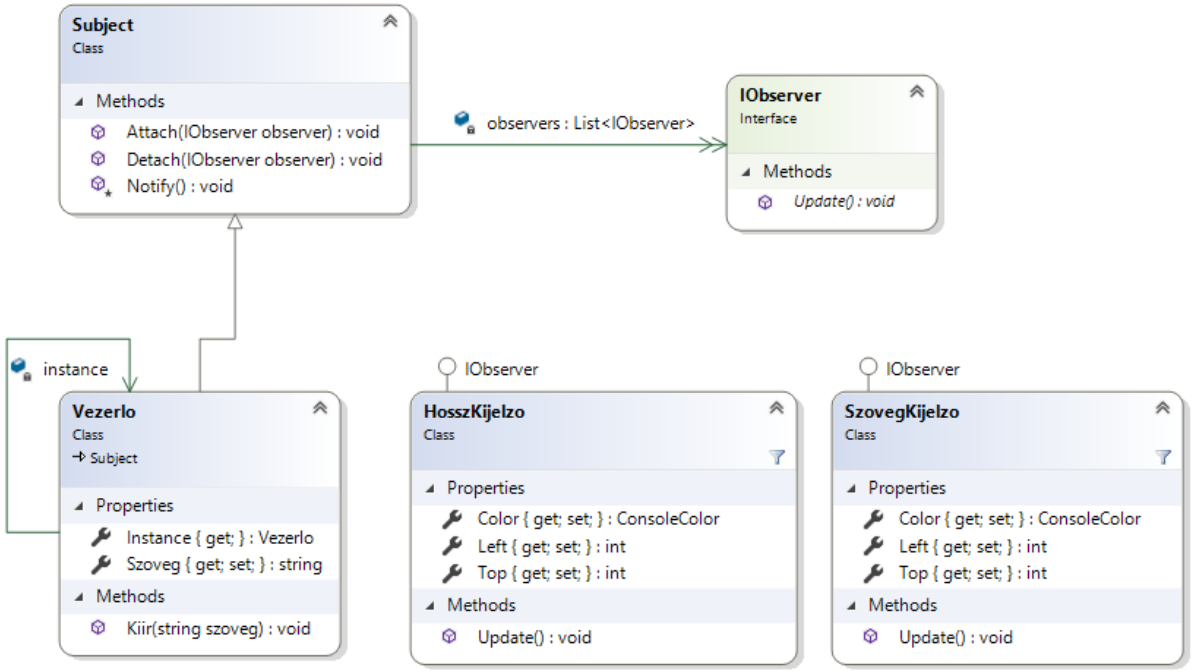
Az alkalmazásnak az alábbi felvételen látható kimenetet kell mutatnia: https://www.aut.bme.hu/Upload/Course/SzTT/publikus_anyagok/F%c3%a9ny%c3%bajs%c3%a1g.mp4

Segítségképpen megadjuk a Kiir függvény kódját:

Vezerlo.cs

```
public void Kiir(string szoveg)
{
    Szoveg = "                         ";
    Notify();
    for (int i = 0; i <= szoveg.Length; i++)
    {
        Szoveg = szoveg.Substring(0, i);
        Notify();
        Thread.Sleep(100);
    }
    Thread.Sleep(2000);
}
```

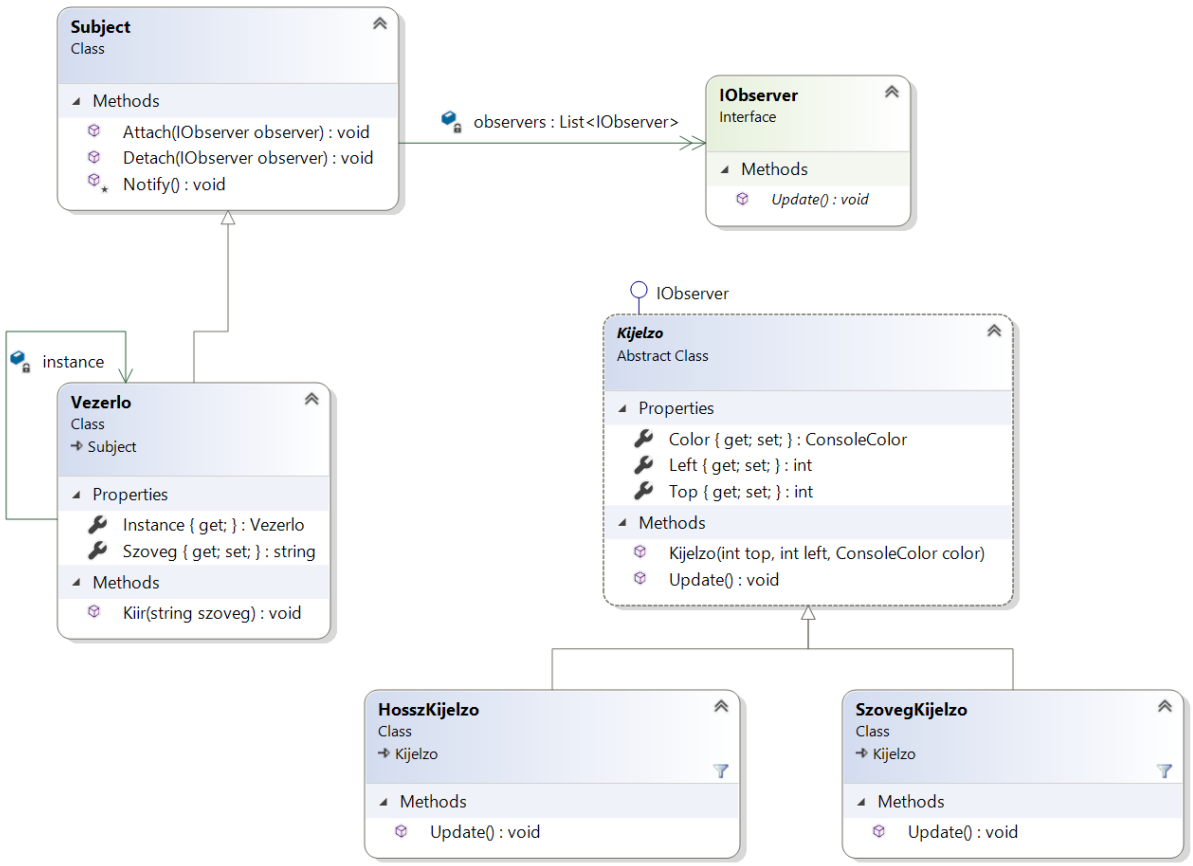
Az alapmegoldás osztálydiagramja a következő:



Ez a megoldás a `HosszKijelzo` és `SzovegKijelzo` vonatkozásában jelentős kódduplikációt tartalmaz. Így két utat járhatunk be:

- Első körös megoldásként elkészíthetjük ezt a megoldást, és ezt követően refaktoráljuk a megoldásunkat úgy, hogy bevezetünk egy `Kijelzo` nevű absztrakt ősosztályt.
- A rutinosabbak egy lépésben is elkészíthetik a végső megoldást.

Segítségképpen a kódduplikációt nem tartalmazó megoldás osztálydiagramja:



További gondolatok:

- Már a `Kijelzo` ősben célszerű implementálni az `IObserver` interfészt.
- Az ábrán nem látszik, de a `HosszKijelzo` és `SzovegKijelzo` osztályokban is szükség van háromparaméteres konstruktorra (melyben az ős konstruktora a `base` kulcsszóval hívható).
- C# nyelven egy osztályban azon függvényeket, melyeket a leszármazottakban felül akarunk definiálni, a virtuálisnak kell definiálni, és a leszármazottban meg kell adni az `override` kulcsszót.
- Amikor egy leszármazott osztályban felüldefiniálunk egy ősbeli virtuális függvényt, lehetőség van az ősbeli változat hívására a `base` kulcsszó segítségével (pl. `base.Update()`).

Több vezérlő támogatása

Ha marad idő, alakítsuk át a megoldást úgy, hogy több vezérlő is létezhessen az alkalmazásban, vagyis az observerek ne singletonként érjék el a subjectet. Ehhez a `Kijelzo` ősben kell egy `Vezerlo` hivatkozást felvenni és az ehhez kapcsolódó változtatásokat a szükséges osztályokon átvezetni.

Extra feladatok

Ezekhez a feladatokhoz csak a fentiek megoldása után, vagy otthoni gyakorlás során nyúljuk, ha még maradt idő. Ezen feladatok a gyakorlat teljesítésébe nem számítanak bele. A feladatok itt már szándékosan csak a célt mondják meg, a megoldás kitalálása a feladat része.

Feladat 1

Valósítsa meg a két feladat valamelyikét úgy, hogy UWP keretrendszert használ! Az egyes kijelzők lehetnek a TextBlock-ból származtatott osztályok, melyek implementálják az IObservable interfészt.