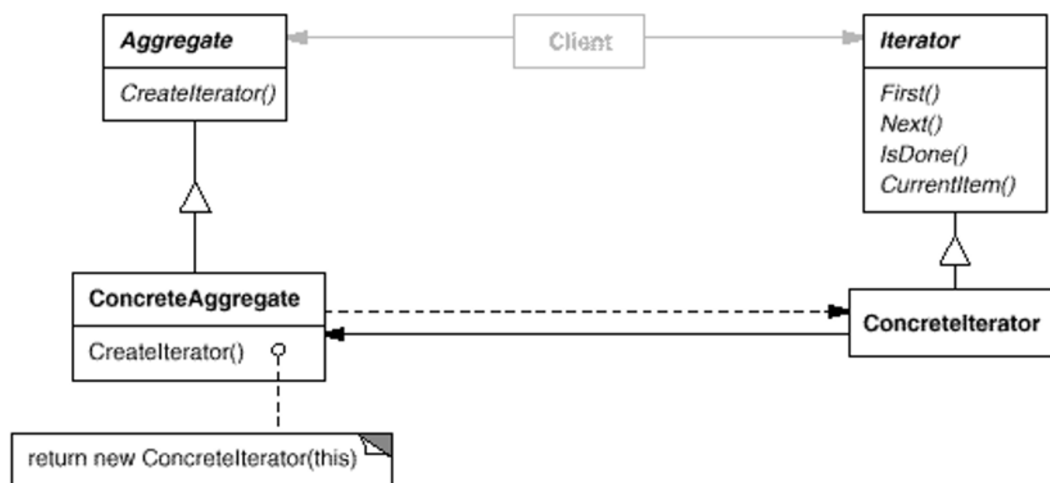
Eseményvezérelt alkalmazások: 12. gyakorlat

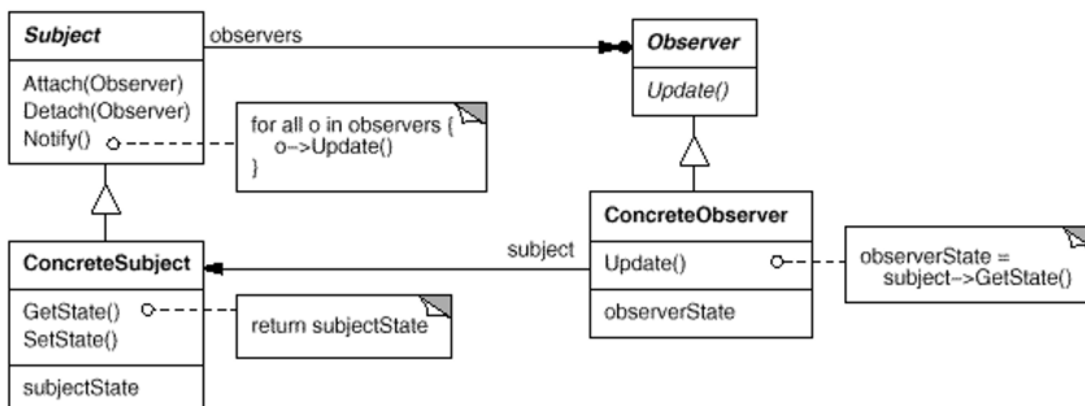
A munkafüzet a reaktív programozásba vezet be egy konzolos alkalmazás segítségével.

A reaktív programozás megértéséhez két tervezési minta (*design pattern*) ismerete szükséges: az Iterátor és a Figyelő (*Observer*) tervezési mintáé. Valójában már mindkét tervmintát jól ismerjük és rendszeresen használtuk:

- az *Iterator* tervezési minta megegyezik az *Objektumelvű programozás* kurzusból megismert felsoroló koncepciójával, és C#-ban az `IEnumerator` interfész implementálásával kerül jellemzően megvalósításra.



- az *Observer* tervezési minta C#-ban nyelvi szinten jelenik meg az események (megfigyelhető objektumok) és az eseménykezelők (megfigyelő objektumok) által.



A két tervminta kombinálásával megkaphatjuk a *megfigyelhető felsorolók* koncepcióját. Ezek olyan felsorolók, amelyek értesítéseket küldenek, amennyiben:

- új elemet állítottak elő (`onNext`);
- sikeresen véget ért a felsorolás (`onCompleted`);
- hibával ért véget a felsorolás (`onError`).

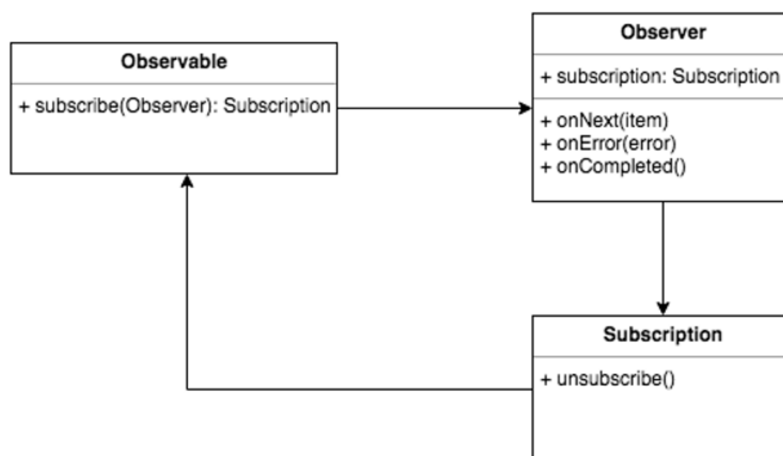


Figure 1: Megfigyelhető felsorolók osztálydiagramja

A felsorolókat adatfolyamoknak (*data streams*) is tekinthetjük. A reaktív programozás ezen megfigyelhető adatfolyamok eseményeinek a funkcionális programozás paradigmájára épülő (és jellemzően aszinkron) feldolgozásáról szól.

Bitcoin árfolyam figyelő alkalmazás
Készítsünk egy olyan egyszerű konzolos alkalmazást, amely a Bitcoin árfolyamát folyamatosan figyeli és a változásokat kiírja. Amennyiben (a program indítása óta) új maximális értéket ért el az árfolyam, külön hívja fel erre a figyelmünket a program.

1 Imperatív megközelítés ^{KM}

Első lépésben készítsük el az alkalmazás "klasszikus", azaz imperatív, nem reaktív változatát. A Bitcoin - EUR árfolyam lekérdezéséhez az ingyenes és publikus *Binance API*-t használjuk, amelyet a munkafüzethez csatolva egy *skeleton* projektként elérhetünk.

1. Adjunk a *BitcoinMonitor* solution-höz egy új konzolos alkalmazás projektet *BitcoinMonitor.Classic* néven.

2. A `Program` osztály `Main` metódusában hozzunk létre egy `System.Timers.Timer` időzítőt, amelyet időzítsünk 5 másodpercenként, valamint a `Tick` eseményére iratkozzunk fel egy `CheckRate` eseménykezelővel. Indítsuk is el az időzítőt.

3. A `CheckRate` eseménykezelőben kérdezzük le az aktuális Bitcoin árfolyamot:

```
decimal? rate = await BitcoinRateFetcher.Fetch();
if (!rate.HasValue)
    return;
```

4. Jelenítsük meg az aktuális árfolyamot egész számként:

```
int rateInt = Convert.ToInt32(rate);
Console.WriteLine($"Bitcoin to EUR: {rateInt} EUR");
```

5. Az árfolyamot csak akkor jelenítsük meg, ha egész értéke megváltozott az előző lekérdezés óta. Ehhez adjunk egy `_lastRate` adattagot a `Program` osztályhoz, hogy eltárolhassuk az előző értéket.
6. A felhasználó figyelmét hívjuk fel, ha az alkalmazás elindulása óta új maximális árfolyammal találkoztunk. Ehhez adjunk egy `_lastMax` adattagot a `Program` osztályhoz.

2 Megfigyelhető felsoroló létrehozása ^{EM}

Készítsük el az alkalmazást a reaktív programozás paradigmáját alkalmazva!

> A reaktív programozáshoz használt egyik népszerű szoftverkönyvtár az eredetileg a Microsoft által kidolgozott, mára nyílt forráskódú *ReactiveX* (röviden: *Rx*) könyvtár. Ez számos programozási nyelvhez elérhető, C#/.NET-hez az Rx.NET (*Reactive Extensions for .NET*) könytárban, amelyet a `System.Reactive` NuGet csomaggal adhatunk a projekteinkhez.

Feladat: Adjunk a *BitcoinMonitor* solution-höz egy új konzolos alkalmazás projektet *BitcoinMonitor.Reactive* néven. A projekthez adjuk hozzá a `System.Reactive` NuGet csomagot. A projektnek legyen függősége a *BitcoinMonitor.Model* projekt.

Rx.NET-ben a megfigyelhető felsorolók közös ős interfésze az `IObservable<T>` típus. Ilyen megfigyelhető felsorolókat számos különböző módon létrehozthatunk az `Observable` osztály gyártó műveleteihez. A legelemibb módja ennek az `Observable.Create<T>()` használata, pl. egy szöveges állomány soronkénti megfigyelhető felsorolására:

```
var myObservable = Observable.Create<string>(observer =>
{
    try
    {
        using var reader = new StreamReader("file.txt");
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            observer.OnNext(line); // Minden sor esetén új elemet jelzünk
        }
        observer.OnCompleted(); // Jelezzük, hogy véget ért a felsorolás
    }
    catch (Exception ex)
    {
        observer.OnError(ex); // Jelezzük, hogy hibával ért véget a felsorolás
    }

    // A végén egy IDisposable objektummal kell visszatérnünk, ami a felsoroló
    // erőforrásait felszabadítja a feliratkozás végén. Most nincs mit felszabadítani,
    // mert a StreamReader egyébként is using blokkban van.
    return () => { };
});
```

Feladat: készítsünk ezek alapján egy olyan `IObservable<decimal?>` felsorolót, amely 5 másodpercenként meghívja a `BitcoinRateFetcher.Fetch()` eljárást. Használjuk az `Observable.Create<decimal?>()` eljárást, benne pedig egy időzítőt (`System.Timers.Timer`). Ügyeljünk arra, hogy az időzítőt felszabadítsuk a visszadott `IDisposable` objektumban.

Megfigyelhető felsorolókat az `Observable` osztály más gyártó műveleteivel is létrehozhatunk, jelen esetben az `Observable.Interval()` lehet segítségünkre, amely a megadott időközzel elkezdi a természetes számokat felsorolni. LINQ használatával a természetes számokat 1-1 Bitcoin árfolyam leképezésre transzformálva az előző feladat megoldását jelentősen egyszerűsíthetjük:

```
var bitcoinObservable = Observable.Interval(TimeSpan.FromSeconds(5))
    .SelectMany(async _ => await BitcoinRateFetcher.Fetch())
```

3 Operátorok ^{EM}

A megfigyelhető felsorolóinkat ún. *operátorok* segítségével szűrhetjük, transzformálhatjuk, kombinálhatjuk. C#/.NET esetén ezt a nyelvbe ágyazott lekérdezések (LINQ) szintaxisával tehetjük meg.

Feladat: a létrehozott `bitcoinObservable` megfigyelhető felsorolóból szűrjük ki a hálózati kapcsolat hibája miatt potenciálisan előforduló `null` értékeket (`Where`). Konvertáljuk és kerekítsük a lebegőpontos `decimal` értékeket egész (`int`) értékekre (`Select`), majd szűrjük ki az egymás utáni ismétlődő elemeket (`DistinctUntilChanged`).

> A `decimal` olyan lebegőpontos típus .NET-ben, amelynek számábrázolása a 10-es számrendszeren alapszik (vö. `float` és `double` 2-es számrendszerbeli ábrázolása). Hatékonysága ezért a `double` és `float` típusokéhoz képest kisebb, azonban a 10-es számrendszerben racionális számok ábrázolása és a velük végzett műveletek nem okoznak a számábrázolásból fakadó hibákat. A `decimal` típus népszerű pénzügyi adatok kezelésekor.

4 Feliratkozás ^{EM}

A megfigyelhető felsorolóknak a munkafüzet bevezetőjében tárgyalt módon 3 eseményére iratkozhatunk fel: `onNext`, `onCompleted` és `onError`.

Feladat: iratkozzunk fel a létrehozott megfigyelhető felsoroló eseményire a `Subscribe()` eljárással. Túlterhelései révén feliratkozhatunk csak az `onNext` vagy akár mindhárom eseményre 1-1 eseménykezelővel.

```
var subscription = bitcoinObservable.Subscribe(
    rate => Console.WriteLine($"Bitcoin to EUR: {rate} EUR"), // onNext
    ex => Console.WriteLine($"Error: {ex.Message}"),          // onError
    () => Console.WriteLine("Monitoring completed.")          // onSuccess
);
```

A visszakapott `subscription` objektum egy egyszerű `IDisposable` objektum, így az egyetlen dolog, amit tehetünk vele, az a `Dispose()` eljárásának meghívása. Ez pontosan az az `IDisposable` objektum, amit az `Observable.Create()` végén visszaadtunk, így ezel végrehajthatjuk a megfigyelhető felsoroló erőforrásainak felszabadítását, ha már nincs rá szükség.

Hívjuk meg a `subscription.Dispose()` metódust a `Main()` eljárás végén.

5 Új maximális értékek figyelése ^{EM}

Feladat: a már elkészített megfigyelhető felsorolónkat újra felhasználva, más operátorok használatával szűrjük ki, ha új maximális Bitcoin árfolyam értéket látunk. Ehhez használjuk a `Scan()` operátort, amely paraméterül egy addig kiszámolt aggregált értéket (jelen esetben a maximumot), és az aktuális felsorolási elemet várja, vissza pedig az új aggregrált értékkel tér (a maximummal). A `Subscribe()` metódus újbóli használatával készítsünk még egy feliratkozást.

Ne felejtsük el ennek a feliratkozásnak is meghívni a `Dispose()` metódusát a `Main()` eljárás végén!

6 Multicasting ^{OP}

Alapértelmezetten egy új feliratkozáskor (`Subscribe`) a megfigyelhető felsoroló inicializálási logikája végrehajtódik, így amennyiben két feliratkozásunk van, valójában két időzítőt indítottunk el. Ezt könnyen megfigyelhetjük például úgy, hogy az `Observable.Create()` végén visszaadott `IDisposable` objektumban kiírjuk a `timer` objektumunk memóriacímét:

```
Console.WriteLine($"Disposing Timer #{timer.GetHashCode()}");
```

Megfigyelhetjük, hogy az üzenet kétszer jelenik meg, különböző memóriacímmel.

Feladat: hatékonyság céljából egyetlen megfigyelhető felsorolóra szeretnénk több feliratkozást kötni, ezt nevezzük *multicasting*-nak. Ehhez két dolgot kell tennünk:

- Megfigyelhető felsorolónk elindítását a `Publish()` eljárással manuálissá alakítjuk. Ezek az ún. *connectable observable* felsorolók nem az első feliratkozáskor kezdik el az elemek felsorolását (ami az alapértelmezett működés), hanem akkor, amikor a `Connect()` eljárást meghívjuk rajtuk.

```
var bitcoinObservable = Observable.Interval(TimeSpan.FromSeconds(5))
    .SelectMany(async _ => await BitcoinRateFetcher.Fetch())
    /* ... */
    .Publish();

var subscription = bitcoinObservable.Subscribe(/* ... */);

bitcoinObservable.Connect(); // felsorolás indítása
```

- Az elindítást és a felszabadítást automatizáljuk a `RefCount()` használatával. Ez egy hivatkozás (referencia) számlálást fog a háttérben végezni, és az első feliratkozáskor indítja a felsorolást, az utolsó feliratkozás felszabadításakor pedig a a felsorolót is felszabadítja.

```
var bitcoinObservable = Observable.Interval(TimeSpan.FromSeconds(5))
    .SelectMany(async _ => await BitcoinRateFetcher.Fetch())
    /* ... */
    .Publish().RefCount();

var subscription = bitcoinObservable.Subscribe(/* ... */);
// az első feliratkozással indul a felsorolás
var maxSubscription = bitcoinObservable.Subscribe(/* ... */);

/* ... */

subscription.Dispose();
maxSubscription.Dispose();
// az utolsó feliratkozással felszabadul a megosztott felsoroló
```

Figyeljük meg, hogy így a *"Disposing Timer ..."* üzenet már csak egyszer jelenik meg.