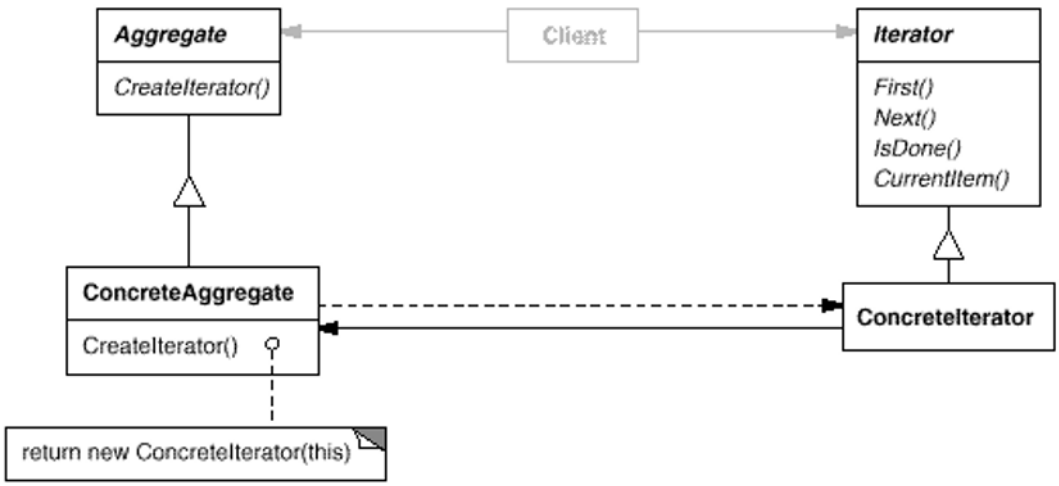
Eseményvezérelt alkalmazások: 12. gyakorlat

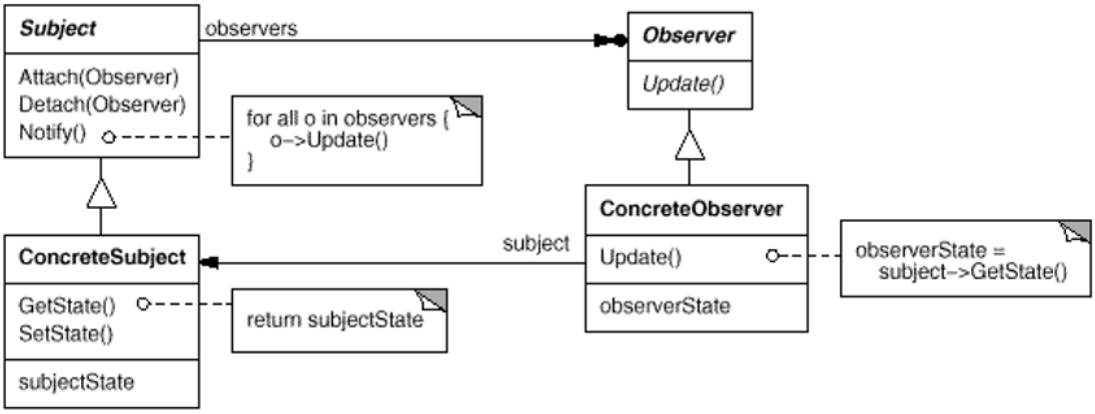
A munkafüzet a reaktív programozásba, és a *Reactive UI keretrendszer* használatába vezet be minket Avalonia UI alkalmazáson példázva.

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`).

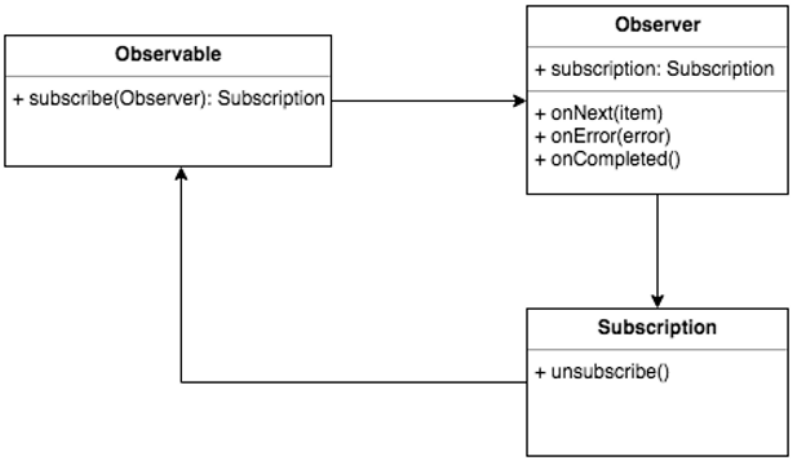


Figure 1: Megfigyelhető felsorolók osztálydiagramja

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

1 Bitcoin árfolyam figyelő alkalmazás ^{KM}

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.

Az alkalmazás "klasszikus", nem reaktív változata elérhető példaként csatolva a munkafüzethez egy *skeleton* projektként. Tanulmányozzuk át alaposan projektet és a program működését! A Bitcoin - EUR árfolyam lekérdezéséhez az ingyenes és publikus *Binance API*-t használjuk, de a feladat szempontjából most a `Program` osztály működésének áttekintése a fontos.

1.1 Megfigyelhető felsoroló létrehozása

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())
```

1.2 Operátorok

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.

1.3 Feliratkozás

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.

1.4 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!

1.5 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.

2 Reaktív programozás Avalonia UI keretrendszerben ^{KM}

Az asztali grafikus alkalmazások felületi eseményei és állapotváltozásai is kezelhetőek reaktív módon, .NET keretrendszerben ezt a ReactiveUI könyvtár teszi lehetővé, amely Windows Forms, Windows Presentation Foundation és Avalonia UI felületű alkalmazásokkal is könnyen integrálható 1-1 NuGet csomagnak a projekthez adásával. A *ReactiveUI* keretrendszer az *Rx.NET* osztálykönyvtárra épül.

MVVM architektúra esetén a nézetmodell megfigyelhető tulajdonságainak (`INotifyPropertyChanged` interfész) változása egy megfigyelhető felsorolóként is értelmezhető. Hasonló szemlélettel a parancsok (`ICommand` interfész) végrehajthatósága (`CanExecute`) is tekinthető egy logikai értékeket felsoroló objektumnak, amely változásairól eseményeket küld (`CanExecuteChanged`), azaz megfigyelhető.

Feladat: Készítsük el a webes képletöltő alkalmazásunkat a reaktív programozás paradigmáját alkalmazva, a *ReactiveUI* keretrendszert használva.

2.1 Projekt létrehozása

Készítsünk Visual Studioban egy új *Avalonia C# Project*-et, a *solution* és a *projekt* neve legyen *ImageDownloader.Avalonia*. Válasszuk ki most is a *Desktop* és *Android* platformokat támogatásra, alkalmazott tervezési mintaként (*design pattern*) azonban most a *ReactiveUI*-t jelöljük ki. Megfigyelhetjük, hogy a platformfüggetlen *ImageDownloader.Avalonia* projekthez a `CommunityToolkit.Mvvm` helyett most az `Avalonia.ReactiveUI` NuGet csomag került hozzáadásra.

A modell nem változik, ezért a *solution*-be átmásolhatjuk a *9. gyakorlaton* elkészített platformfüggetlen *Class Library* projektet *ImageDownloader.Model* néven. Adjuk hozzá az *ImageDownloader.Avalonia* projektünkhöz a létrehozott *ImageDownloader.Model* projektet függőségként (*"Add Project Reference"*).

2.2 Nézetmodell

Figyeljük meg, hogy a generált `ViewModelBase` osztályunk most a *ReactiveUI* `ReactiveObject` típusából származik le. Ez az ősosztály megvalósítja a `INotifyPropertyChanged` interfészt és a reaktív programozáshoz (később használandó) eszközöket biztosít számunkra.

2.2.1 `ImageViewModel`

A *11. gyakorlatról* emeljük át a `ImageViewModel` osztályt, majd igazítsuk a *ReactiveUI* keretrendszerhez. A `RelayCommand` osztály helyett most a `ReactiveCommand<T1, T2>` osztályt használhatjuk a parancsainkhoz, amely `T1` típusú paramétert vár és `T2` típusú eredményt ad eredményül. Ha nem akarunk paramétert vagy eredményt megadni, akkor tekintettel arra, hogy a `void` nem használható generikus típusparaméterként, a `Rx.NET` speciális `Unit` típusát használhatjuk. Ennek megfelelően:

- a `SaveImageCommand` és a `CloseCommand` típusa legyen `ReactiveCommand<Unit, Unit>`;
- definiálni a `ReactiveCommand.Create()` eljárással tudjuk őket, pl.:

```
SaveImageCommand = ReactiveCommand.Create(() =>
{
    SaveImage?.Invoke(this, Image);
});
```

2.2.2 `MainViewModel`

A *11. gyakorlatról* emeljük át a `MainViewModel` osztályt is, majd igazítsuk ezt is a *ReactiveUI* keretrendszerhez:

- A `Progress` és az `IsDownloading` megfigyelhető tulajdonságokhoz használjuk a `ReactiveObject`-ből örökölt `RaiseAndSetIfChanged()` metódust, amellyel egyszerre beállíthatjuk a kapcsolt adattagot és kiválthatjuk a `PropertyChanged` eseményt, pl.:

```
private float _progress;

public float Progress
{
    get => _progress;
    set => this.RaiseAndSetIfChanged(ref _isEnabled, value);
}
```

- Az `IsDownloading` változásának esetén a `DownloadButtonLabel` tulajdonságra is ki kell váltatnunk `PropertyChanged` eseményt. Erre ReactiveUI használata esetén a `ReactiveObject`-ből örökölt `RaisePropertyChanged()` metódust használhatjuk az `IsDownloading` *setter* ágában.
- Az `ImageSelectCommand` típusa legyen `ReactiveCommand<Bitmap, Unit>`, hiszen ez a parancs paramétert vár, de visszatérési értéke nincsen. Definiálásához használjuk a generikus `ReactiveCommand.Create<Bitmap, Unit>()` gyártó műveletet.
- A `DownloadCommand` típusa hasonló módon legyen `ReactiveCommand<string, Unit>`. Definiálásakor a `ReactiveCommand.CreateFromTask<string, Unit>()` eljárást használjuk, így aszinkron tevékenység is végrehajtható a törzsében (a `Download` metódus aszinkron):

```
DownloadCommand = ReactiveCommand.CreateFromTask<string, Unit>(async param =>
{
    await Download(param);
    return Unit.Default;
});
```

2.3 Alkalmazás futtatása

A nézetek változatlan formában felhasználhatóak a *11. gyakorlaton* elkészített megoldásból. Futassuk az alkalmazást!

Vegyük észre, hogy a képek letöltése után a *"Letöltés megszakítása"* gomb nem engedélyezett, így a funkció nem használható. Ennek oka, hogy alapértelemezetten a `ReactiveCommand`-hoz kapcsolt `CanExecute` olyan logikai értékeket sorol fel, hogy a parancs párhuzamosan többször nem hajtható végre. Míg az *MVVM Toolkit* esetében ezt a `RelayCommand` attribútum `AllowConcurrentExecutions=true` argumentumával oldottuk meg, a `ReactiveCommand` esetében más megközelítést alkalmazunk: megadjuk, hogy a `CanExecute` egyetlen `true` értéket soroljon fel, így mindig végrehajtható lesz a parancs:

```
DownloadCommand = ReactiveCommand.CreateFromTask<string, Unit>(async param =>
{
    await Download(param);
    return Unit.Default;
}, Observable.Return(true));
```

3 Képek szűrése a *ReactiveUI* használatával ^{EM}

Egészítsük ki a webes képletöltő alkalmazás funkcionalitást egy szűrési lehetőséggel, amellyel a letöltött képek helyettesítő szövegében (`alt` attribútum) lehet keresni.

3.1 Modell

A modell két osztályát (`WebImage` és `WebPage`) egészítsük ki az új funkcionalitás támogatásával:

- a `WebImage` osztályt egészítsük ki egy publikus `string AltText` tulajdonsággal (legyen publikusan írható is);
- a `WebPage` osztályban egy új `WebImage` elkészítésekor állítsuk be annak helyettesítő szövegét (ha van):

```
var image = await WebImage.DownloadAsync(imageUrl);
if (node.Attributes.Contains("alt"))
{
    image.AltText = node.Attributes["alt"].Value;
}
```

3.2 Nézetmodell

Egészítsük ki a `ImageViewModel` nézetmodellt 2 új tulajdonsággal (`AltText`, `IsEnabled`), a következő módon:

- A `string AltText` a kép helyettesítő szövegét tárolja (nem kell az állapotváltozásait jeleznie, mivel nem fogjuk módosítani). A konstruktor várja paraméterül az `AltText` értékét.
- A `bool IsEnabled` adja meg, hogy a szűrőkifejezés alapján a kép kiválasztott-e. Legyen megfigyelhető, ehhez használjuk a `ReactiveObject`-ből örökölt `RaiseAndSetIfChanged()` metódust a *setter* ágban, a már korábban látottakhoz hasonlóan.

A `MainViewModel` nézetmodellben implementáljuk a letöltött képek közötti szűrést, a reaktív paradigma alkalmazásával.

- Először is adjunk az osztályhoz egy `string SearchContent` megfigyelhető tulajdonságot (és egy kapcsolt `_searchContent` adattagot), amelyet adatkötéssel a felület kereső mezőjének tartalmához tudunk majd kötni.
- Adjunk az osztályhoz egy `IObservable<string> _searchContentObservable` privát adattagot is, amely a `SearchContent`-nak a felhasználói input hatására változó tartalmát fogja megfigyelhető módon felsorolni.
- Az osztály konstruktorában definiáljuk a `_searchContentObservable`-t a következő módon:
  1. A `SearchContent` változásait szeretnénk felsorolni, ehhez használhatjuk a `ReactiveObject` ősosztályhoz tartozó `WhenAnyValue()` eljárást:
     ```
     this.WhenAnyValue<MainViewModel, string>(x => x.SearchContent);
     ```
  2. Csak akkor állítson elő a felsorolónk új elemet, ha 0.5 másodperce nem történt billentyűleütés, azaz állapotváltozás a `SearchContent` értékében. Ehhez használjuk az *Rx.NET* `Throttle(TimeSpan.FromSeconds(0.5))` szűrőjét.
  3. Csak akkor állítson elő a felsorolónk új elemet, ha a keresőmező tartalma a *whitespace*-eket levágva megváltozott. Ehhez használjuk a `Select()` transzformátort és a `DistinctUntilChanged()` szűrőt.
  4. Az aszinkron feldolgozás miatt fontos, hogy a megfigyelők a *UI threaden* kerüljenek majd végrehajtára, hiszen majd a felületen megjelenített képek vezérlőihez hozzá kell férniük. Ezt grafikus keretrendszertől független módon a `ObserveOn(RxApp.MainThreadScheduler)`hozzáadásával érhetjük el.

- Az `Images`-ben a továbbiakban nem csak a képet (`Bitmap`), hanem a képre vonatkozó teljes nézetmodellt (`ImageViewModel`) tárolni fogjuk, így a típusát is ennek megfelelően változtassuk `ObservableCollection<ImageViewModel>`-re.
- Az `OnImageLoaded` eseménykezelőben az új `ImageViewModel` létrehozása után állítsuk be, hogy amennyiben az adott kép helyettesítő szövege tartalmazza a megadott keresőkifejezést (vagy az üres), akkor engedélyezett a kép, egyébként nem.

```
private void OnImageLoaded(object? sender, WebImage webImage)
{
    if (!IsSupportedExtension(webImage.Url.LocalPath))
        return;

    var bitmap = new Bitmap(new MemoryStream(webImage.Data));
    var imageViewModel = new ImageViewModel(bitmap, webImage.AltText);
    _searchContentObservable
        .Select(s => imageViewModel.AltText.Contains(s) || s == string.Empty)
        .Subscribe(s => imageViewModel.IsEnabled = s);

    Images.Add(imageViewModel);
}
```

3.3 Nézet

A `MainView` nézetet egészítsük ki még egy szövegdobozzal, amellyel a kereső kifejezést a felhasználóm megadhatja. Ennek `Text` tulajdonságát kössük a nézetmodell `SearchContent` tulajdonságához.

A képeket megjelenítő `ItemsControl` vezérlő `ItemTemplate`-jében a gomb (`Button`) vezérlő `IsEnabled` tulajdonságát kössük az aktuális `ImageViewModel`-nek az `IsEnabled` tulajdonságához. Így csak azok a képek lesznek kattintásra megnyithatóak, amelyekre illeszkedik a kereső kifejezés.

A kép (`Image`) vezérlő forrását (`Source`) most már nem a teljes nézetmodellhez, hanem annak `Image` tulajdonságához kössük. Ugyanitt a megjelenített képhez vegyünk fel egy `IsHighlighted` stílus osztályt, amelyet szintén az aktuális `ImageViewModel`-nek az `IsEnabled` tulajdonságához kössünk. Ezen stílussal módosítsuk azon képek áttetszőségét 10%-ra, amelyekre nem illeszkedett a keresés:

```
<Image Stretch="Fill" Source="{Binding Image}"
       Classes.IsHighlighted="{Binding IsEnabled}">
  <Image.Styles>
      <Style Selector="Image">
          <Setter Property="Opacity" Value="0.1" />
      </Style>
      <Style Selector="Image.IsHighlighted">
          <Setter Property="Opacity" Value="1" />
      </Style>
  </Image.Styles>
</Image>
```

4 Reaktív forráskód generátorok ^{OP}

Ahogyan az *MVVM Toolkit* esetében is rendelkezésünkre álltak olyan attribútumok, amelyeket adattagok és metódusok fölé illesztve a megfigyelhető tulajdonságok és a parancsok kódgenerálása automatizálható volt; erre a *ReactiveUI* esetén is lehetőségünk van, a `ReactiveUI.SourceGenerators` NuGet csomag projektünkhöz adásával.

Használatával az adattagok a `Reactive` attribútummal ellátva generálhatunk olyan *property*-t, amely változása esetén az adattagot módosítja, és a `PropertyChanged` eseményt is kiváltja magára. Pl.:

```
[Reactive]
private float _progress;

// A Progress property generálva lesz.
```

Metódusokat a `ReactiveCommand` attribútummal ellátva a nevükhöz fűzött `Command` szuffixszel ellátott nevű parancs kerül generálásra, amely végrehajtásakor az annotált metódust fogja végrehajtani. Pl.:

```
[ReactiveCommand]
private void ImageSelect(Bitmap param)
{
    ImageSelected?.Invoke(this, param);
}

// Az ImageSelectedCommand generálva lesz.
```

A `ReactiveCommand` attribútum használatakor a generikus paraméterekre nincs szükség, azt az annotált metódus szignatúrájából dedukálja.

Feladat: használjuk ki a reaktív forráskód generátorok képességeit a programunkban!