DESIGN PATTERNS INTRODUCTIE

WEEK 3

Design Pattern Opdracht: “Observer”

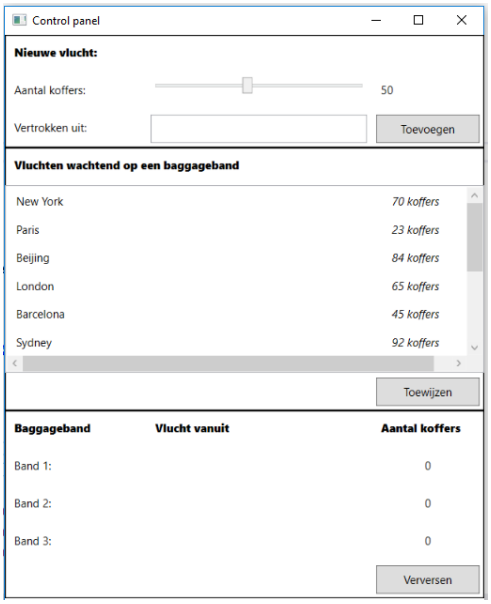
Deze week ga je aan de slag met het Observer patroon. Het observerpatroon is nodig om de koppeling tussen klasses kleiner te maken en om het mogelijk te maken meerdere objecten naar één object te laten kijken.

Een voorbeeld hiervan is een baggagebandsysteem. Op een vliegveld komen meerdere vluchten in korte tijd binnen, al deze vluchten hebben baggage bij zich die opgehaald moet worden bij een baggageband. Wanneer een baggageband leeg is, kan een volgende vlucht daar aan toegewezen worden zodat de passagiers hun baggage weer op kunnen halen.

1. Clone de repository en run de code

Clone het project https://github.com/Avans/DPINT_Wk3_Observer.
Hier zie je een window die alles controleert. Dit scherm bestaat uit drie delen:

- In het bovenste deel kan je nieuwe vluchten toevoegen aan de wachtrij.
- In het middelste deel kan je de wachtrij van vluchten zien. Wanneer er minstens één baggageband vrij is kan je op toewijzen klikken zodat de eerste vluchten naar de lege baggagebanden worden toegeleid.
- In het onderste zie je de baggagebanden, welke vluchten hieraan toegewezen zijn en hoeveel koffers er nog op de band liggen. Elke band heeft zijn eigen verwerkingssnelheid, klik op verversen om de nieuwe status op te halen.



DESIGN PATTERNS INTRODUCTIE

WEEK 3

2. Bekijk de code

We hebben drie modelklasses:

- **Vlucht** heeft een property om aan te geven waar hij vandaan komt en hoeveel koffers hij bij zich heeft.
- **Baggageband** werkt met een timer. Uit de huidige toegewezen vlucht worden een x-aantal koffers per minuut van de band gehaald.
- **Aankomsthal** weet welke vluchten wachten, welke banden (drie in ons geval) er zijn en kan de langst wachtende vluchten toewijzen aan een baggageband.

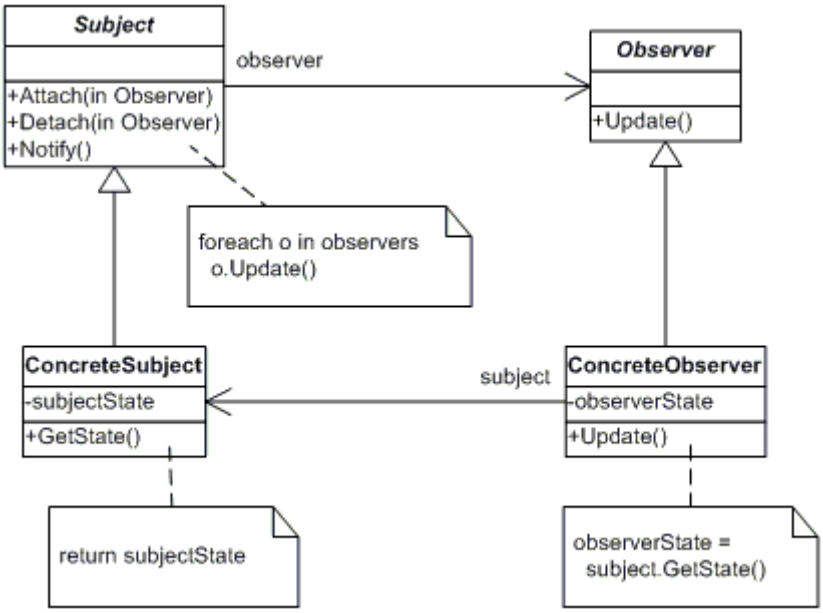
Daarnaast hebben we een viewmodels voor de banden en voor de vluchten, deze wordt in de onderste twee schermdelen gebruikt om vluchtinformatie te tonen.
In het **MainViewModel** worden de knoppen afgehandeld en worden de *VluchtInformatieViewModels* toegewezen aan de betreffende lijsten. De schermen worden hier up to date gehouden door op de juiste momenten de methodes *VerversBaggagebanden* en *VerversWachtendeVluchten* aan te roepen. Stel we willen nu een nieuw scherm maken die hier ook van afhankelijk is, dan moeten we erg veel code aanpassen!
Bekijk maar eens hoe veel update-methodes er in de applicatie zitten.

3. Leer het Observable patroon kennen

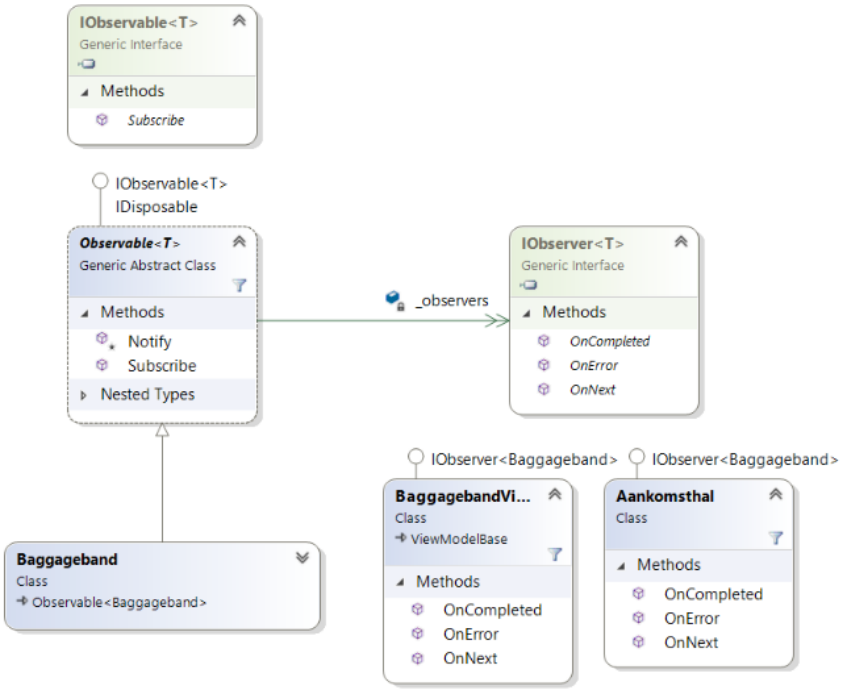
Het observablepatroon wordt tegenwoordig veel toegepast en zit ook al vaak in de volwassen talen verwerkt. Bindings binnen WPF worden namelijk ook al zo gedaan (Hint: *RaisePropertyChanged* is hier een onderdeel van).
Wij gaan gebruik maken van de bestaande **IObservable<T>**, **IObserver<T>** en **ObservableCollection<T>**. We gaan hiermee de verschillende klasses elkaar alleen middels een generieke interface laten kennen, hierdoor kunnen we gemakkelijk onderdelen vervangen.

Het idee is dat een Subject weet dat er andere objecten hem bekijken, observeren. Hij laat het weten als er iets veranderd is. De andere objecten zullen vervolgens zichzelf updaten zodra ze hier een seintje van krijgen.

WEEK 3



In ons voorbeeld is dit de implementatie, zoals je ziet gaan we 2 observers bouwen om de baggageband in de gaten te houden:



4. Maak een Observable<T> en Unsubscriber<T> klasse

We willen graag de **Baggageband** (en later de **Vlucht**) observeerbaar maken, maar omdat we code willen kunnen hergebruiken zorgen we ervoor dat we een klasse **Observable<T>** hebben die abstract is. Hierin kunnen we alle code stoppen die we nodig hebben om observeerbaar te zijn.
De **Baggageband** zal dan overerven van deze klasse en automatisch deze functionaliteit er bij krijgen.

Maak de abstracte klasse **Observable<T>**, deze moet de interface **IObservable<T>** en de interface IDisposable implementeren, vul onderstaande klasse daarom aan:

DESIGN PATTERNS INTRODUCTIE

WEEK 3

Let op: Binnen C# hebben we andere methodes dan het standaard Observer pattern aangeeft. De observable roept all zijn observers aan met de methode OnNext als hij een nieuwe waarde heeft. OnCompleted wordt aangeroepen wanneer het object 'beëindigd' is en hij dus nooit meer een update zal sturen.
Wij maken de klasse Observable<T> zodat we herbruikbare code niet opnieuw hoeven te schrijven in de (mogelijk) verschillende subklasses zoals BaggageDestination.

```
public abstract class Observable<T> : IObservable<T>, IDisposable
{
    // Aan hen moeten we een seintje geven als we veranderd zijn.
    private List<IObserver<T>> _observers;

    public Observable()
    {
        _observers = new List<IObserver<T>>();
    }

    /// <summary>
    /// Deze private class gebruiken we om terug te geven bij de Subscribe methode.
    /// </summary>
    private struct Unsubscriber : IDisposable
    {
        private Action _unsubscribe;
        public Unsubscriber(Action unsubscribe) { _unsubscribe = unsubscribe; }
        public void Dispose() { _unsubscribe(); }
    }

    public IDisposable Subscribe(IObserver<T> observer)
    {
        // TODO: We moeten bijhouden wie ons in de gaten houdt.
        // TODO: Stop de observer dus in de lijst met observers. We weten dan
        // welke objecten we allemaal een seintje moeten geven.

        // Daarna geven we een object terug.
        // Als dat object gedisposed wordt geven wij
        // de bovenstaande observer geen seintjes meer.
        return new Unsubscriber(() => _observers.Remove(observer));
    }

    /// <summary>
    /// Deze methode kunnen we aanroepen vanuit onze subklasses.
    /// Hier geven we dan een seintje aan al onze observers dat we veranderd zijn.
    /// </summary>
    /// <param name="subject">Dat is de "this" van onze subklasses</param>
    protected void Notify(T subject)
    {
        // TODO: Hier moeten we iedere observer die ons in de gaten houdt een seintje
geven dat we een nieuwe waarde hebben. We roepen dus hun OnNext methode aan.
        throw new NotImplementedException();
    }

    public void Dispose()
    {
        // Deze implementeren we later
        throw new NotImplementedException();
    }
}
```

WEEK 3

We willen nu ook dat de klasse **Baggageband** observeerbaar wordt. Zorg daarom dat hij overererft van de klasse Observable<T> op de volgende manier:

```
public class Baggageband : Observable<Baggageband>
```

Zorg er nu voor dat elke keer als er iets verandert in de **Baggageband** dat alle observers een seintje krijgen.

5. Zorg ervoor dat BaggagebandViewModel en de Aankomsthal de IObserver<T> implementeert

Implementeer de volgende methodes (volgens de interface):

```
/// <summary>
/// Deze gaan we niet gebruiken
/// </summary>
void OnCompleted();
/// <summary>
/// Deze gaan we niet gebruiken
/// </summary>
/// <param name="error"></param>
void OnError(Exception error);
/// <summary>
/// Als er een update is wordt deze aangeroepen, je krijgt hier heel het object
binnen. Dus elke keer als er een waarde binnen het object dat wij in de gaten houden
verandert zal deze methode aangeroepen worden. We kunnen dan onze view aansturen dat
de nieuwe waarde op het scherm moet komen.
/// </summary>
/// <param name="value"></param>
void OnNext(T value);
```

Hoe zou je nu in de constructor omgaan in je **BaggagebandViewModel**? Je subscribet dan op de **Baggageband**. Roep ook meteen je eigen methode *OnNext* aan, zo heb je meteen de juiste waardes.

In de klasse **Aankomsthal** kennen we de methode *WachtendeVluchtenNaarBand,* deze werd aangeroepen vanuit een knop en de methode *AssignVluchten* vanuit de **MainViewModel.** Dit willen we natuurlijk ook automatisch laten gaan, en dat kan nu!
We krijgen namelijk een seintje van de banden wanneer er koffers af gaan en wanneer de laatste koffer er af gaat kunnen we meteen een nieuwe vlucht toewijzen aan deze band.
Schrijf daarom de *WachtendeVluchtenNaarBand* methode zo om dat hij in de *OnNext* methode past.
Let ook op dat de *NieuweInkomendeVlucht* nu rekening moet houden met lege banden.

De baggagebanden werken nu automatisch! Je hebt de knop verversen dus niet meer nodig! Ook de methode *VerversBaggagebanden en AssignVluchten* in het **MainViewModel** kunnen dus leeggemaakt worden.

6. Gebruik de ObservableCollection<T>

In C# kennen we de klasse **ObservableCollection<T>**, deze laat het weten als een collectie is aangepast. Dit is handig voor de **WachtendeVluchten** in onze **Aankomsthal**, maak van deze list dan ook een **ObservableCollection<T>** overerft.

WEEK 3

Abonneer in de klasse **MainViewModel** op het CollectionChanged event zodat je meteen alle veranderingen doorkrijgt. Daarnaast kenden we natuurlijk de methode *VerversWachtendeVluchten*. Daar zit een code snippet in om te luisteren naar de **ObservableCollection** van de **Aankomsthal.** Gebruik deze code snippet.
Ten slotte hoeven we niet meer zelf de WachtendeVluchten te vullen in de methode *AddNieuweVlucht*, haal daarom deze regel code ook weg.

Merk op dat de methode *VerversWachtendeVluchten* nu dus automatisch werkt en we de hele knop niet meer nodig hebben!

7. Maak de vlucht zelf ook nog observable

We willen van de vlucht dat we kunnen zien hoe lang hij al wacht, dit kunnen we natuurlijk ook mooi met een observable oplossen.

Onderstaande kan je doen als bonus om de observable beter te leren begrijpen.

Gebruik het volgende als basis:

```
// Vlucht.cs
// TODO: Zorg voor een private timer _waitingTimer
// TODO: Zorg voor een public property TimeWaiting (TimeSpan)
// Constructorcode:
     _waitingTimer = new Timer();
     _waitingTimer.Interval = 1000;
     _waitingTimer.Tick +=
                 (sender, args) => TimeWaiting = TimeWaiting.Add(new TimeSpan(0,0,1);
     _waitingTimer.Start();
// Nieuwe methode erbij:
public void StopWaiting()
{
     _waitingTimer.Stop();
     _waitingTimer.Dispose();
}

// Aankomsthal.cs (OnNext methode)
volgendeVlucht.StopWaiting();
```

Zorg er voor dat de vlucht observable is en dat je de TimeWaiting mooi kan weergeven in de wachtrij (of loggen in console log). Let op, dan moet je dus ook het VluchtViewModel en de MainWindow.xaml nog aanpassen.

Nu ben je klaar! Een hoop dingen gaan nu automatisch en we hoeven niet zelf de hele tijd meer op verversen te klikken, dat is een groot voordeel. We hebben nu altijd realtime data én kan het proces in de applicatie ook realtime op basis van gegevens elders in de applicatie doorgaan.