DESIGN PATTERNS INTRODUCTIE

WEEK 2

Design Pattern Opdracht: "Decorator"

Deze week gaan jullie een nieuw design pattern maken waarbij gebruik maakt van de design patterns die je in week 1 al gebruikte:

1. Strategy (week 1)
2. Factory method (week 1)
3. Decorator (week 2)

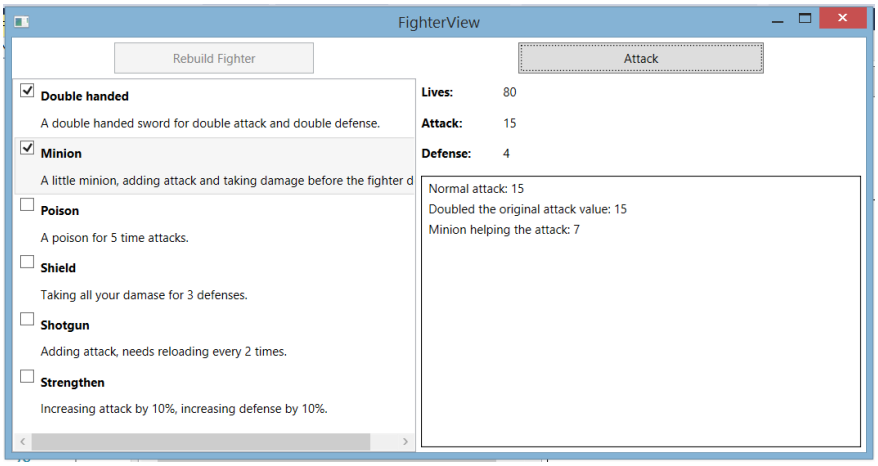
Net zoals een Strategy, is een Decorator gebaseerd op een interface. Sterker nog, ze kunnen gebaseerd zijn op dezelfde interface.

Als je opzoekt wat een decorator is, vind je onder doel: "*Het dynamisch aan een object koppelen van extra verantwoordelijkheden. Decorators leveren een flexibel alternatief voor het maken van subklassen voor het uitbreiden van de functionaliteit.*"[1]

We gaan de decorator gebruiken om dynamisch functionaliteit toe te voegen of weg te laten. In het bijzonder ga je in deze opdracht een fighter maken. Deze fighter kan aanvallen en verdedigen. Afhankelijk van de verschillende opties die deze fighter heeft zal hij beter aanvallen, verdedigen of beide.

1. Clone de repository en run de code

Clone het project https://github.com/Avans/DPINT_Wk2_Decorator.
Hier zie je twee gelijke windows. Links kan je kiezen om je fighter op te waarderen. Rechts kan je een attack uitvoeren. In het window komt te staan hoe deze attack uitgevoerd is, in het andere window zie je hoe de verdediger zijn defend heeft uitgevoerd.
Speel eens met de verschillende instellingen. Let op: Je moet wel elke keer op *Rebuild Fighter* klikken om het toe te passen.



[1] Design Patterns. De Nederlandse versie. Elementen van herbruikbare objectgeoriënteerde software. Gamma, Helm, Johnson en Vlissides

WEEK 2

De gebruiker moet kunnen kiezen uit de volgende aanvullingen op de fighter:

- **Double Handed**
  A double handed sword for double attack and double defense.
- **Minion**
  A little minion, adding attack and taking damage before the fighter does.
- **Poison**
  A poison for 5 time attacks.
- **Shield**
  Taking all your damase for 3 defenses.
- **Shotgun**
  Adding attack, needs reloading every 2 times.
- **Strengthen**
  Increasing attack by 10%, increasing defense by 10%.
  Let op: De strengthen gaan we later in deze opdracht nog invullen!

2. Bekijk de code

We zien in het **FighterViewModel** de volgende regels zitten:

```
var options = OptionList.Where(o => o.Selected).Select(o => o.Name);
_fighter = _fighterFactory.CreateFighter(Lives, AttackValue, DefenseValue, options);
```

We hebben vorige week de Factory Method al gezien. Deze wordt hier ook gebruikt, de geselecteerde opties worden meegegeven om de **FighterFactory** een fighter te laten maken. De code in het viewmodel hoeft dus niet aangepast te worden.

De **FighterFactory** maakt een Fighter door alle opties goed te zetten. Deze gaan we straks wel aanpassen.

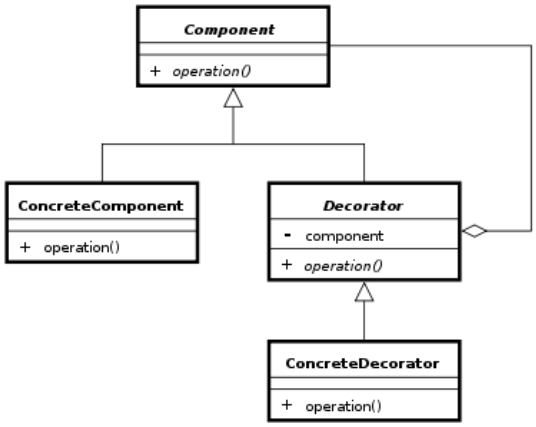
```
Fighter fighter = new Fighter(lives, attack, defense);

foreach (var option in options)
{
    switch (option)
    {
        case DOUBLE_HANDED:
            fighter.DoubleHanded = true;
            break;
        case MINION:
            fighter.MinionLives = fighter.Lives / 2;
            fighter.MinionAttackValue = fighter.AttackValue / 2;
            break;
        case POISON:
            fighter.PoisonStrength = 10;
            break;
        case SHIELD:
            fighter.ShieldDefends = 3;
            break;
        case SHOTGUN:
            fighter.UseShotgun = true;
            break;
    }
}
```

WEEK 2

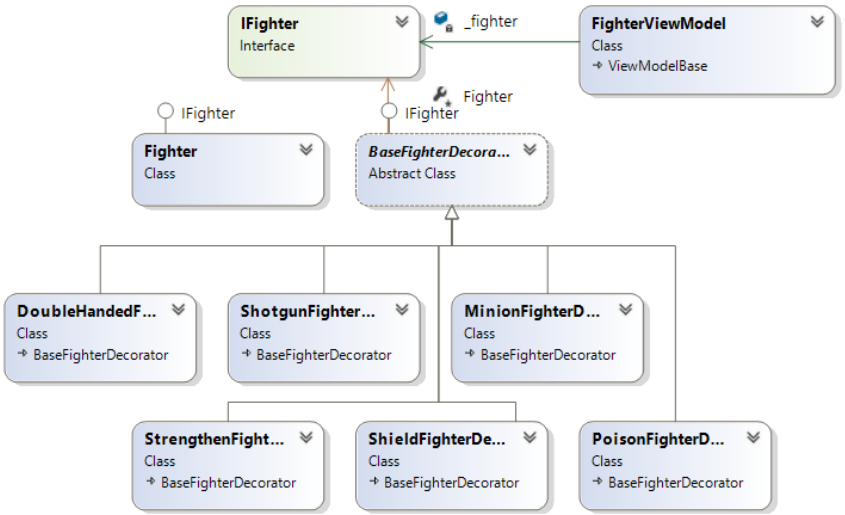
Ten slotte hebben we de **Fighter** klasse. Deze implementeerd de **IFighter** met de properties *Lives, AttackValue en DefenseValue.* Daarnaast heeft hij de methodes *Attack* en *Defend.* Deze methodes staan vol met if-statements en zijn niet heel erg onderhoudbaar. Hier gaan we iets aan doen.

3. Maak kennis met het klassediagram.

Een decorator voldoet aan een interface die gelijk is aan de interface van het basisobject. Daarnaast heeft hij een referentie naar een object van diezelfde interface.
Zie het als een schil om een ui (de groente) heen. De ui is nog steeds een ui zonder die schil. En om die schil kan weer een nieuwe schil zitten, en dan is het nog steeds een ui.



Gelukkig hebben wij die interface al, **IFighter**!
We willen daarom naar de volgende situatie, we hebben de interface **IFighter** (de ui), de **Fighter** (de kern van de ui), o.a. **DoubleHandedFighterDecorator, ShotgunFighterDecorator** (de schillen van de ui) en een basisklasse hiervoor **BaseFighterDecorator**:
Je ziet dat de verschillende decorators dus een referentie hebben naar de volgende schil én zelf een schil zijn!



WEEK 2

4. Maak de klasse BaseFighterDecorator

Maak de klasse **BaseFighterDecorator**. Dit is een schil en moet dus de **IFighter** interface implementeren. Hij wijst ook naar een andere schil dus moet in zijn constructor een **IFighter** binnen krijgen. Hoe zou je de interface implementeren volgens het decorator pattern? Delegeer het naar de volgende schil!
Let er op dat je klasse *abstract* is en dat je de methodes *virtual* maakt!

5. Maak de subklasses voor de verschillende decorators

Maak de klasses **DoubleHandedFighterDecorator**, **ShotgunFighterDecorator**, **MinionFighterDecorator**, **ShieldFighterDecorator** en **PoisonFighterDecorator**.
Dit zijn allemaal subclasses van **BaseFighterDecorator**.

Haal de betreffende code uit de **Fighter** klasse en plaats ze in de juiste klasse door de *Attack*, *Defend* of beide methodes de overriden. Let ook op dat je altijd ook nog de volgende schil aanroept als je hem override!

Zie je hoe schoon je code wordt? Elke klasse heeft nu maar 1 doel!

6. Pas de FighterFactory aan

Pas de **FighterFactory** zo aan dat de Fighter met de geselecteerde opties gemaakt wordt.

Je ziet nu bijvoorbeeld dat de volgende code ontstaat door schillen om elkaar heen de maken:

```
IFighter fighter = new Fighter(lives, attack, defense);

// ...
fighter = new MinionFighterDecorator(fighter);
// ...

return fighter;
```

Zie je dat je nu vanuit je viewmodel nog steeds geen code hebt aangepast? Handig die Factories!

7. Run de code

Als het goed is kan je je applicatie weer runnen, er is echter nog niets veranderd voor de gebruiker.

8. Maak een nieuwe optie: Strengthen

Maak nu een nieuwe optie waarbij de attackwaarde met 10% verhoogd wordt en de defensewaarde met 10% verhoogd wordt.
Als het goed is maak je één nieuwe klasse en pas je maar één bestaande klasse aan: De **FighterFactory**

Dit was het decorator pattern, van ingewikkelde if/else-structuren naar een overzichtelijk en eenvoudig systeem dat enorm uitbreidbaar is.