Decorator Pattern

|  |  |
| --- | --- |
| Navn | Studienummer |
| Jeppe Traberg Sørensen | 201507686 |
| Niklas Sørensen | 201509378 |
| Henrik Søby Jørgensen | 20101979 |

## Software Design

## Beskrivelse

Man kan komme I situationer hvor der skal oprettes et objekt som har noget basalt funktionalitet som skal forøges i en given situation.

Det kan løses ved at tjekke diverse bools og lave subklasser, dette kan dog blive meget stort og give utrolig mange klasser hvis man skal til have kombinationer af funktionalitet.

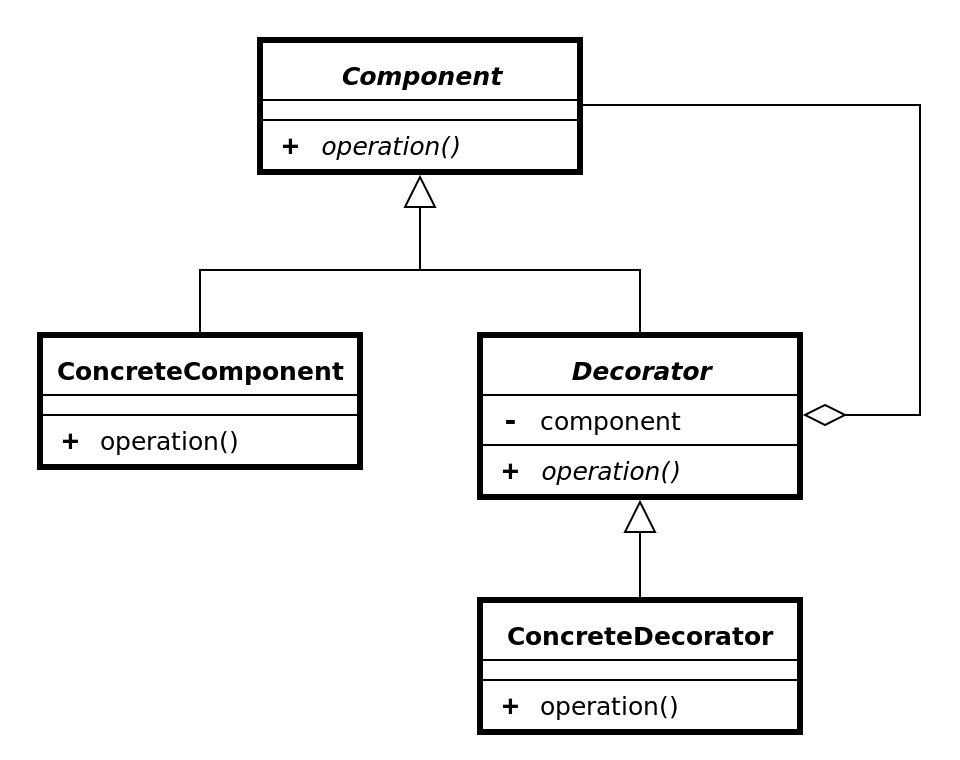
Til at løse dette kan Dectorator patten benyttes og GoF beskriver det som:

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

Vi kan se på klasse diagrammet for dette pattern.

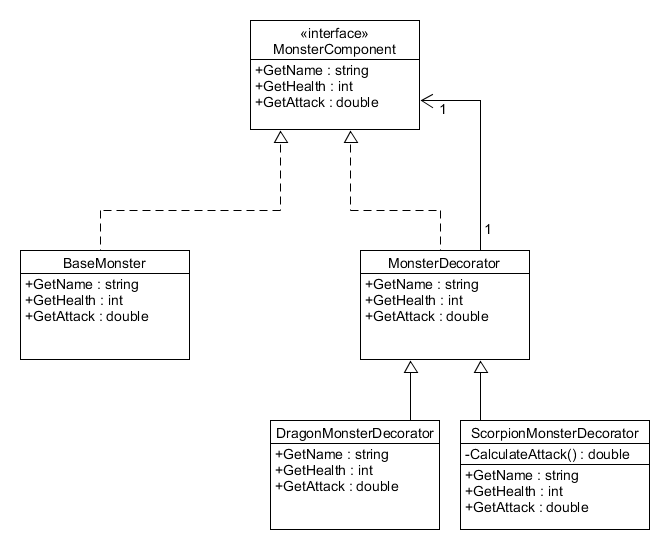
Der startes med den **Component** interface af egentlige base objekt **ConcreteComponent** som skal kunne forøges med funktionalitet

Det **ConcreteComponent** er det objekt hvor funktionalitet kunne blive tilføjet dynamisk. **Decorator** er det interface for alle de dynamiske funktioner som kan blive tilføjet **ConcreteComponent**. Den **ConcreteDecorator** er så der hvor alt funktionaliteten er, hvert funktionalitet skal have sin egen **ConcreteComponent** klasse.



# Eksempel

Vi har til formålet opstillet et eksempel beskrevet med UML diagram. I eksemplet vil vi se på Monstre.



Der bliver først lavet et *component* Interface som i dette tilfælde er **MonsterComponent**:



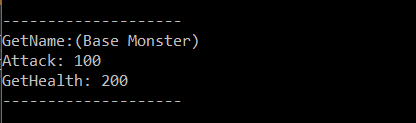
Derefter et BaseMonster som alle andre monstre vil blive dannet udfra. Denne har en base værdi for Navn, Health og Attack:



Starten af et testprogram bliver lavet:



En hurtig test af BaseMonsteret udskriver dette:



Dette monster vil vi så gerne udvikle andre monstre fra. Så vi implementere et interface **MonsterCompnent** og en **MonsterDecorator** med de variabler vi ønsker at ændre. Fx:

I **MonsterDecorator** bliver et sæt variabler oprettet som vi kan bruge til at ændre værdier med.

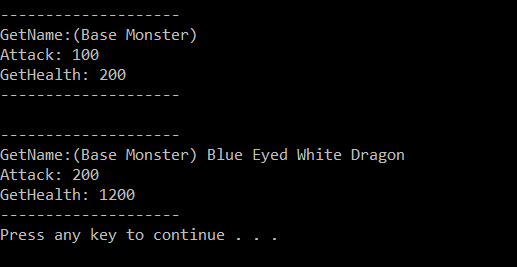


Samt bliver vores funktioner sat til *public virtual* så vi kan override disse senere i **ConcreteDecorators**



Nu vil vi så lave et monster decorator, helt præcist en **DragonMonsterDecorator**. DragonMonster skal naturligvis have et nyt navn samt mere Heath og mere Attack end BaseMonster. Så vi går ind og overrider de 3 metoder: GetName(), GetHealth(), GetAttack() og giver nye værdier med. Vi giver et nyt navn (og BaseMonster navnet for eksemplets skyld), 1000 mere health en BaseMonster og 100 mere Attack end BaseMonster.



I et test program kan vi nu oprette en nyt **DragonMonsterDecorator** som får vores **BaseMonster** som parameter.Og testkørslen bliver således:

Dette er princippet i Decorator patternet. Vi kan bruge det på andre flexible objekter der skal oprettes.

Nu skal et Scorpion monster oprettes, men Scorpionen skal ikke altid have samme Health Attack eller Name. Det er muligt at oprette en **ScorpionMonsterDecorator** som tager imod argumenter i constructoren som kan udnyttes til at gøre hver Scorpion mere unik.



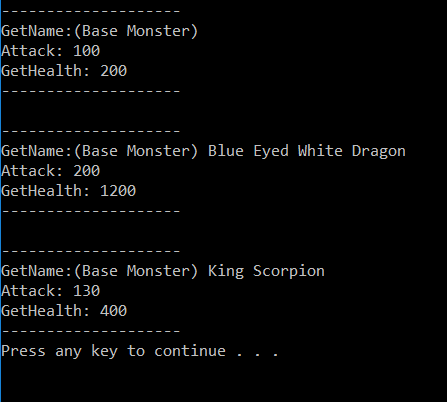
Ligeledes kan hjælpe metoder laves i Decorator klasserne så når der fx skal laves en 1.5x multiplier på Attack kan det let gøres og returneres således:



Testprogrammet udvides med et Scorpion monster.



Og testkørslen bliver således:



# Struktur

Decorator kan ved første øjekast meget blive sammenlignet med Strategy Pattern som vi har gennemgået i undervisningen. Forskellen ligger i at Strategy lader dig ændre implementationen af noget ved runtime.

Hvorimod Decorator lader dig forøge og ’augment’ funktionaliteten uden at ændre på nogen implementering af ’BaseObjektet’

Template

# Konklusion

Arbejdet med decorator patternet har været meget ligetil. Når et eksempel skulle findes var det stortset kun fantasien som sætter grænser. Det kan anvendes i mange praktiske systemer ligefra Bagerier (dekorere kager) til spil sammenhænge hvor mange objekter skal bruges af gangen. Det giver et godt alternativ til nedarvning især hvis noget funktionalitet skal deles.

Decorator er et godt eksempel på Open-Closed Princippet fordi det er meget åben for extension men lukket for modification.