Decorator Pattern

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

## I4SWD

## 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 forskellige moduler.

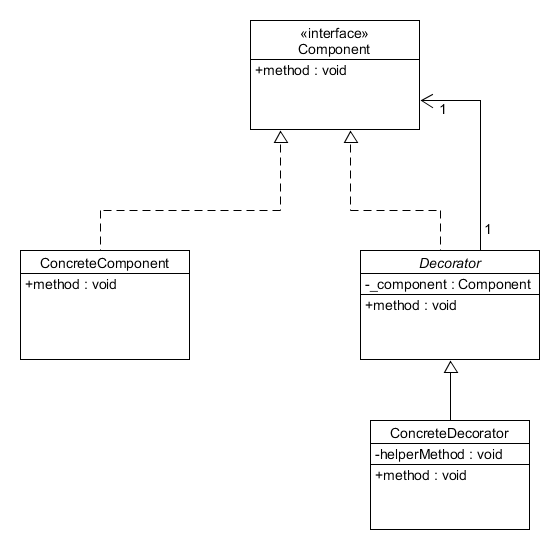
Til at løse dette kan Dectorator pattern benyttes og i GoF beskriver det som:

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

Vi kan se på klassediagrammet for dette pattern.

Der startes med **Component**, der giver et interface til det **ConcreteComponent**, som er den klasse man ønsker at udvide med funktionalitet. Dvs. **ConcreteComponent** er den klasse, hvor funktionalitet kan tilføjes til dynamisk.

**Decorator** er det interface for alle de dynamiske funktioner som kan blive tilføjet **ConcreteComponent**. Decorator har et Component og implementerer dens interface. Decorator fungerer som et interface, men er kodemæssigt ikke et. Det er en abstrakt klasse, hvor dens funktionalitet implementeres af **ConcreteDecorators**. ConcreteDecorators skal implementere metoderne fra Decorator, men kan samtidig have sine egne private metoder.

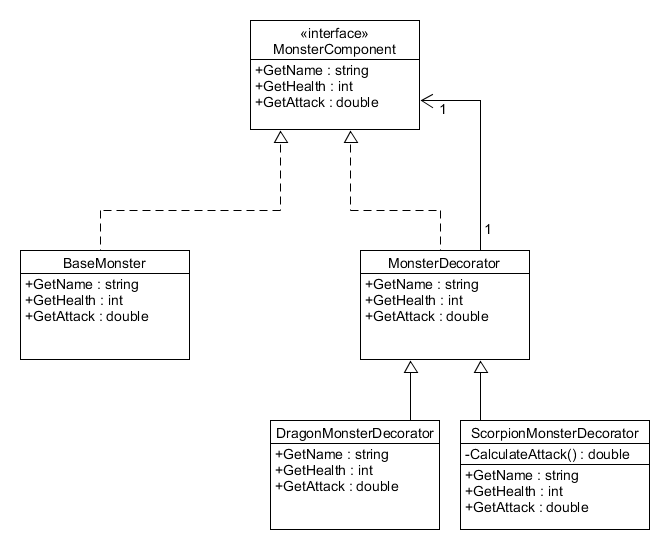


Figur 1. Generisk decorator pattern.

# Eksempel

Decorator Pattern kan bl.a. bruges i spil. Her gør mønstret det muligt nemt at udvide funktionaliteten for et MonsterComponent. Det kunne f.eks. være nyttigt hvis man har et spil med mange monstre, hvor de har forskellige måder at angribe på og deraf får forskellig ’attack points’.

Vi har til formålet opstillet et eksempel beskrevet med et UML diagram, se Figur 2. Decorator Pattern eksempel med monstre.



Figur 2. Decorator Pattern eksempel med monstre.

Der bliver først lavet et *Component* Interface som i dette tilfælde er MonsterComponent. Som sagt skal denne fungere som interface til den klasse, som vi ønsker at udvide. Med nedenstående interface kan vi altså udvide GetName, GetHealth og GetAttack metoderne.



Figur 3. Interface til MonsterComponent.

Vi laver nu vores ConcreteComponent, BaseMonster. Se Figur 4.



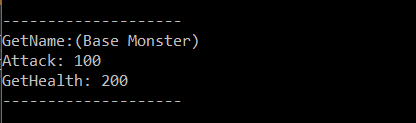
Figur 4. Implementering af Component-interfacet.

Starten af et testprogram bliver lavet, se Figur 5.



Figur . Test program for BaseMonster.

Ovenstående test udskriver nedenstående, se Figur 6.



Figur 6. Udskrift fra testprogram.

Målet er nu at udvide funktionaliteten af de metoder, der kaldes. Dvs. at udvide GetAttack, Gethealth og GetName. Vi udvider nu vores program med en decorator, **MonsterDecorator,** se Figur 7.



Figur 7. Implementering af MonsterDecorator.

Bemærk at alle metoderne er lavet virtuelle. Vi får på denne måde et slags ”interface”, uden at det er et rigtigt interface. Dvs. at vi nu *kan* implementere/overskrive disse virtuelle funktioner i de nedarvede klasser, hvilket netop er formålet, da vi nu får muligheden for at lave forskellige implementeringer af MonsterDecorator.

Vi laver nu en concreteDecorator, **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 end BaseMonster og 100 mere Attack end BaseMonster. Se Figur 8.

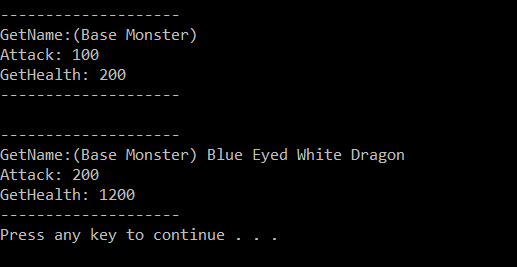


Figur 8. DragonMonsterDecorator.

I et test program kan vi nu oprette en nyt **DragonMonsterDecorator** som får vores **BaseMonster** som parameter. Vi får heraf lavet et BaseMonster med udvidelsen fra DragonMonsterDecorator. Dvs. vores DragonMonster stadig er et BaseMonster, bare med udvidet funktionalitet: mere health, attack osv… I Figur 9 ses en test af vores DragonMonsterDecorator. 

Figur . Test program for DragonMonsterDecorator.

Udskriften fra test-programmet ses herunder, se Figur 10.



Figur 10. Udskrift fra vores test-program.

Dette er princippet i Decorator pattern. 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.



Figur . ScorpionMonsterDecorator constructor.

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:



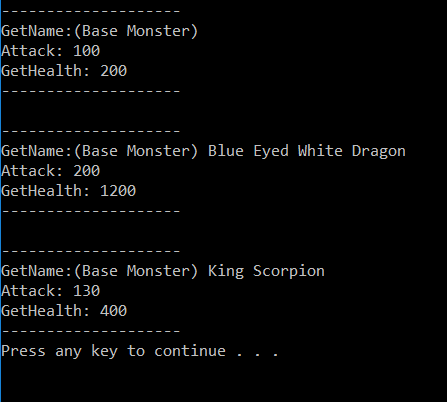
Figur 12. ScorpionMonsterDecorator metoder.

Testprogrammet udvides med et Scorpion monster.



Figur 13. Testprogram til ScorpionMonster udvidelsen.

Udskriften fra testprogrammet ses herunder i Figur 14.



Figur 14. Udskrift fra testprogram til ScorpionMonster udvidelsen.

# Struktur (to-do)

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.