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 udvides i en given situation. Det kan løses ved at tjekke diverse bools internt i klassen, eller ved at benytte sig af nedarvning, men dette kan dog blive meget stort og give utrolig mange klasser hvis man skal til have kombinationer af forskellige moduler/udvidelser. Hvis vi f.eks. har tre forskellige nedarvninger, der skal kunne kombineres ender vi ud med 1\*2\*3=6 forskellige klasser.

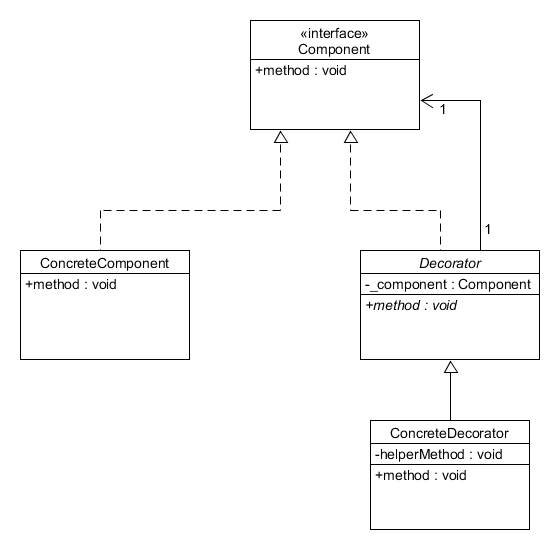
Vi kan undgå at lave så mange overflødige klasser ved at benytte os af Decorator Pattern. Mønstret er i GoF beskrevet som:

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."[[1]](#footnote-1)

Mønstret er nemmest at overskue ved at kigge på klassediagrammet på Figur 1 der viser mønstret generisk. 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 en reference til Component og implementerer samtidig dets interface. Decorator har referencen til **ConcreteComponent** så den kender til basisimplementeringen af dens metoder. Decorator fungerer som et interface, men er ikke et interface i C#’s forstand. Det er derimod en abstrakt klasse, hvor dens funktionalitet udvides af **ConcreteDecorators**. ConcreteDecorators skal implementere metoderne fra Decorator (inkluderet basisimplementeringen fra **ConcreteComponent**, således der *udvides* i stedet for erstattes) men kan samtidig have sine egne private metoder.

Da Decorator pattern udvider funktionalitet er betegner man det som et adfærdsmæssigt pattern.

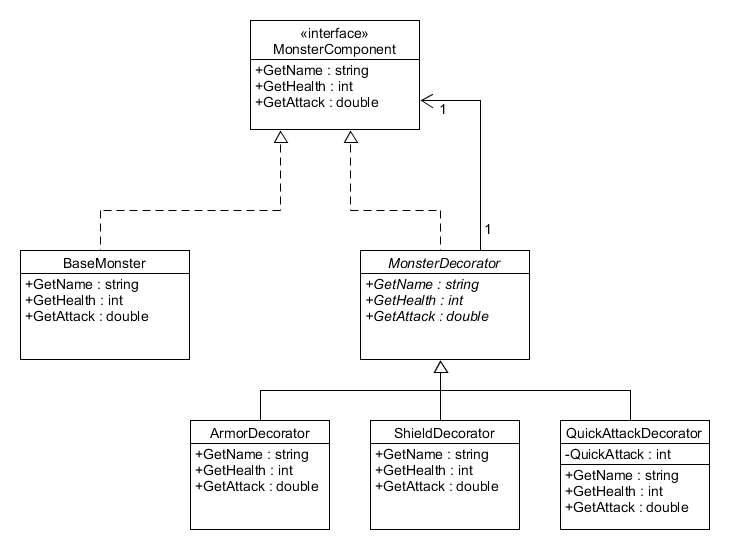


Figur 1. Generisk decorator pattern.

# Eksempel

Decorator Pattern kan bl.a. bruges i sammenhæng med et 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 en ”monster unit” (f.eks. i forbindelse med RTS spil), som kan udvides/opgraderes som spillet skrider frem.

For at illustrere et sådan scenarie er opstillet et eksempel beskrevet med et UML diagram, se Figur 2.



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. Det er denne klasse, der kan udvides med ny funktionalitet senere hen.



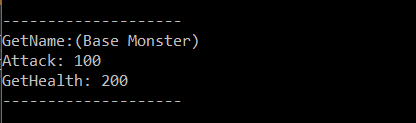
Figur 4. Implementering af Component-interfacet.

På Figur 5 ses et testprogram der gør brug af BaseMonster uden udvidelser.



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.

Der bemærkes at alle metoderne er lavet virtuelle og klassen er lavet abstract. 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. Bemærk dog, at vi ikke behøver at implementere *alle* funktionerne i MonsterDecorator, men kun de vi ønsker at udvide. Det eneste funktionerne gør i MonsterDecorator, er at kalde basisimplementeringen fra Basemonster via dens reference.

Der kan nu tilføjes udvidelser dvs. concreteDecorators. Der udvides nu med klasserne på Figur 2, nemlig ”ArmorDecorator”, ”ShieldDecorator” og ”QuickAttackDecorator”. Så til hver af disse klasser overrides de metoder hvor det giver mening. F.eks. giver Armor og Shield mere health, og derfor overrides GetHealth() metoden. QuickAttack giver derimod mere attack, hvilket er hvorfor GetAttack() overrides. Se Figur 8 for et eksempel på dette, vist med implementeringen af QuckAttackDecorator. Læg især mærke til at der stadig benyttes basisimplementeringen (baseMonster) set i f.eks. GetName() med ”base.GetName”.



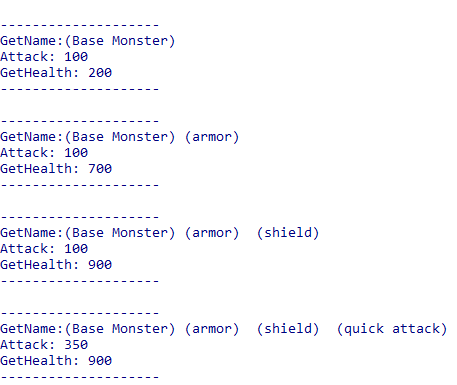
Figur 8. QuickAttackDecorator.

I et test programmet benyttes nu de forskellige decorators til at lave forskellige versioner af monstre. Dette gøres i praksis ved først at oprette et baseMonster objekt, hvilket så kan bruges som parameter i de forskellige decorators constructors. Dette kan ses i praksis på Figur 9.



Figur 9. Test program med benyttelse af decorators

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



Figur 10. Udskrift fra vores test-program.

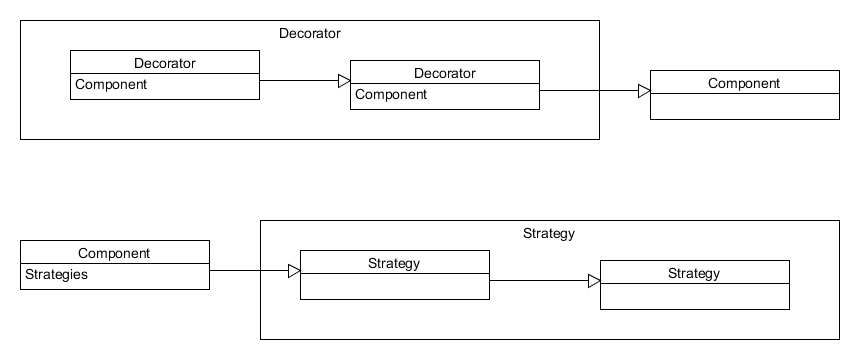
Altså er det ligetil at udvide monster objekterne med de opgraderinger de ønskes. Vi har stadig kun tre forskellige decorators, hvis vi havde brugt nedarvning havde vi som bekendt skulle bruge seks.

# Struktur

Decorator Pattern er meget sammenligneligt med Strategy Pattern, som er gennemgået i undervisningen. Strategy pattern går ind og ændrer implementeringen af en komponent baseret på hvad opgave der skal løses. Derimod giver decorator mulighed for at udvide et komponents funktionalitet, uden at ændre det. Med andre ord er den store forskel, at strategy *ændrer* og decorator *udvider*. Decorator Pattern er derfor helt oplagt til at overholde Open-Closed princippet i SOLID: *”Open for extension, but closed for modification.”*.

Decorator ændrer kun en komponent udefra, det vil sige at komponenten ikke kender til dens mulige decorators. Når strategier er brugt kender komponenten de mulige strategier og har en logik til at holde styr på disse. På Figur 11 kan dette samspil ses. Først illustreret vises hvordan en komponent kan få tilknyttet et ukendt antal decorators. Derefter vises hvordan at ved brug af strategy så skal komponenten kende til de mulige strategier den kan vælge til et ønsket formål.

Det er oplagt at bruge Decorator når komponentklassen skal holdes simpel. Vi undgår at komponentklassen indeholder funktionalitet, der ikke bliver brugt. Skulle komponenten blive større og have vigtig adfærd som fx en søge algoritme vil Strategy pattern være mere oplagt at bruge.



Figur 11 Visuel Repræsentation af Decorator- og Strategy pattern

# Konklusion

Decorator Pattern har vist sig at være brugbart i mange sammenhænge. Mønstret er mest brugbart i de situationer, hvor man vil udvide funktionaliteten af allerede implementerede funktioner. Det er f.eks. oplagt i spilsammenhænge, hvor man vil udvide en spillers Attack-metoder i takt med at han f.eks. stiger i level, eller lærer nogle nye angreb.

Det smarte ved Decorator Pattern er også, at det er ligetil, at kombinere forskellige decorators. Hvis vi f.eks. har 20 forskellige armor-decorators i et spil, så kan vi kombinere disse uden at lave yderligere klasser i modsætning til hvis man havde brugt nedarvning.

Decorator er et godt eksempel på Open-Closed Princippet, da det er åben for udvidelse, men lukket for ændring.

1. https://www.codeproject.com/Articles/479635/UnderstandingplusandplusImplementingplusDecoratorp [↑](#footnote-ref-1)