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 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.

Til at løse dette kan Dectorator pattern benyttes hvilket bliver beskrevet af GoF som værende:

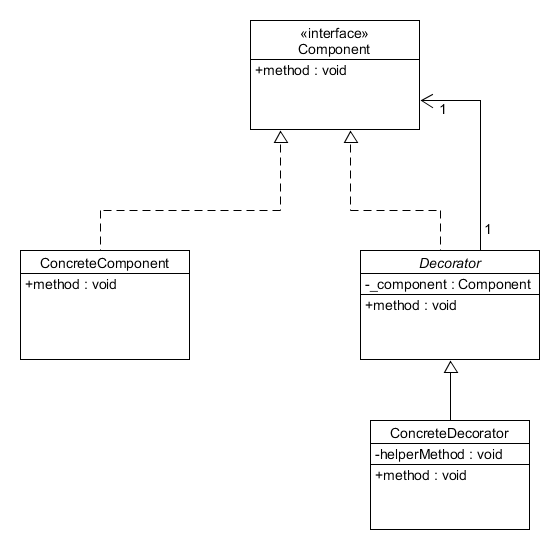
"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."[[1]](#footnote-1)

Dette beskrives nemmet 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 det selvfølgelig et adfærdsmæssigt pattern.

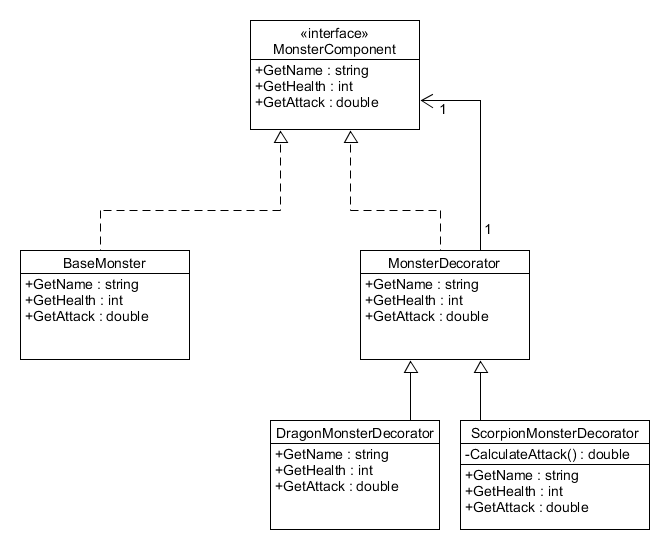


Figur 1. Generisk decorator pattern.

# 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.

Vi har til formålet 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.



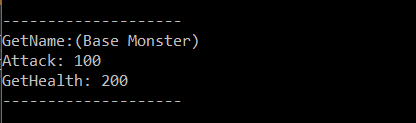
Figur 4. Implementering af Component-interfacet.

På Figur 5 ses et testprogram der gør brug af BaseMonster uden nogen 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. 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. Det eneste funktionerne gør i MonsterDecorator, er at kalde basisimplementeringen fra Basemonster via dens reference.

Der kan nu tilføjes udvidelser dvs. concreteDecorators. Der udvides med de klasser som set på Figur 2, nemlig ”ArmorDecorator”, ”ShieldDecorator” og ”QuickAttackDecorator”. Så til hver af disse klasser overrides de metoder hvor der 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”.



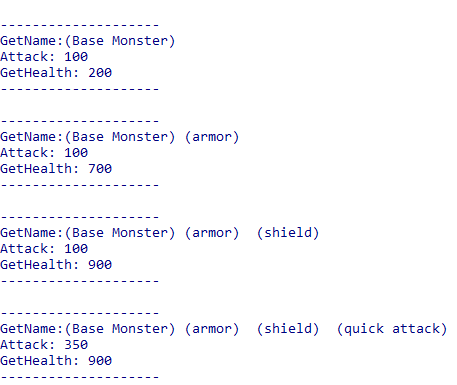
Figur 8. QuickAttackDecorator.

I et test programmet benyttes nu de forskellige decorators til at lave forskellige versioner af monstre. Dette gøres i praktis 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 . 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 forholdvis ligetil at udvide monster objekterne med de opgraderinger de ønskes.

# 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.

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