Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 7 – Mot ett nytt paradigm för design och DP Decorator

Designmönster med C++

Syfte:	Lektionen diskuterar principer och strategier för OOP och designmönster samt introducerar DP Decorator.
Att läsa:	Design Patterns Explained kap 14 sid 253 – 267 och kap 17 sid 297 – 311.
	Kapitel 15 och 16 läses kursivt.

Kapitel 14 ’The Principles and Strategies of Design Patterns’

Kapitlet presenterar ett antal principer som enligt författaren bör ligga till grund för arbetet med objektorienterad design och designmönster. Först bara en liten repetition av hur begreppen ’abstrakt klass’ och ’interface’ implementeras i C++ (och Java för jämförelsens skull). En gemensam egenskap är att de inte kan instansieras i sig utan måste helt eller delvis implementeras av andra klasser, i C++ alltid i samband med arv.

	C++	Java
Interface	`class` med bara ’pure virtual’ medlemsfunktioner =0 som alla saknar implementation. Endast static datamedlemmar.	Nyckelordet `interface`. Endast metoder utan implementation. Inga instansvariabler, däremot får klassvariabler finnas.
Abstrakt klass	`class` med minst en ’pure virtual’ medlemsfunktion =0. Implementationer får finnas liksom alla typer av datamedlemmar.	`abstract class`, metoder kan vara implementerade. Metoder som saknar implementation markeras som `abstract`. Instans- och klassvariabler.

Open-Closed principle (OCP)

Grundtanken med OCP är att så lite kod som möjligt ska *ändras*. Förändringar medför alltid risk för oväntade (och oönskade) sidoeffekter. Utvidgning av koden för att möta nya krav ska göras genom att skriva *ny* kod, inte ändra befintlig. Koden bör vara

- *stängd för förändring av de delar som gäller de grundläggande koncepten som representeras av interface och abstrakta klasser*.

- *öppen för utvidgning genom att nya implementationer av interface och abstrakta klasser kan läggas till*.

Till lektionen finns bifogad en artikel, ”The Open-Closed Principle” (ocp.pdf) som diskuterar detta mer ingående.

Principle of Designing from Context

- Koppling mellan ”using objects” och ”used objects” bör göras på konceptuell nivå (interface/abstrakta klasser) och inte på implementationsnivå i enlighet mellan GoF:s princip att designa mot

interface.

- Det är behoven för ”using objects” som ska avgöra hur interfacet för ”used objects” ska utformas (trots att ”using objects” är beroende av ”used objects”).

- Abstraktioner ska inte vara beroende av detaljer. Detaljer ska vara beroende av abstraktioner.
- Högnivåmoduler ska inte vara beroende av lågnivåmoduler. Båda typerna ska vara beroende av abstraktioner

} ”Dependency inversion principle”

En röd tråd i kapitlet är att författaren betonar hur viktigt det är att vara medveten om det sammanhang (context) där de klasser jag designar ska användas. ”Hur ska de användas?” är en central fråga.

Principle of Encapsulating Variation
Denna princip tillämpas i alla de DP som hittills presenterats.

Bridge – de olika implementationerna kapslas in bakom ett gemensamt interface

Abstract Factory – kapslar in variationer av hur olika familjer av objekt ska instansieras bakom ett gemensamt interface.

Adapter – anpassar en klass till ett önskat interface, det adapterade objektet som implementerar interfacet kapslas in som privat instansvariabel i adaptern.

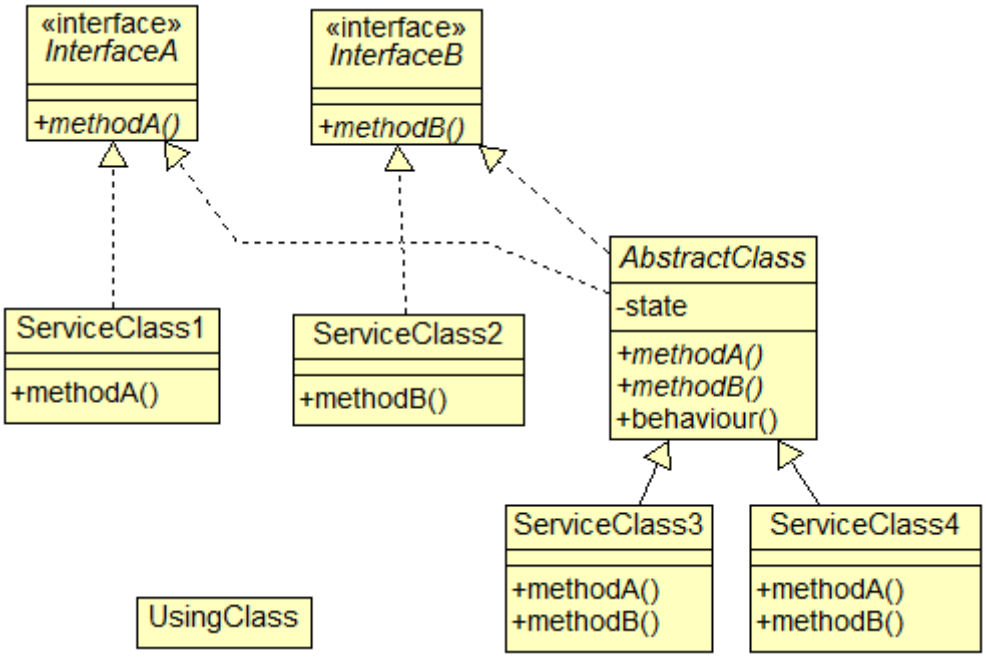
Facade – om flera olika subsystem ges fasader med samma interface kan variationer i subsystemen kapslas in bakom detta gemensamma interface.

Abstract Classes vs. Interface
Hur väljer man mellan en abstrakt klass och ett interface?
Om alla deriverade klasser har gemensamma attribut eller gemensamt uppträndande kan detta placeras som operationer och attribut i en abstrakt klass för att undvika redundans. Operationer kan där implementeras med ett ’default behaviour’ för deriverade konkreta klasser.
Abstrakta klasser möjliggör alltså ’state’ och ’behaviour’ (till skillnad från interface som endast deklarerar operationer och därmed är ’state-less’).

Om man i designen utgår från ett ”using object” som kräver tjänster från många olika abstraktioner kan detta leda till ett interface för varje abstraktion. I stället för ett stort ”used object” får vi flera interface som är tunnare men bätttre sammanhållna (strong cohesion).

Interface och abstrakta klasser kan samarbeta för att ge en flexibel lösning.

Någon ”best practise” finns inte, olika varianter måste utvärderas i sitt sammanhang. Följande klassdiagram kan illusterera bokens resonemang på s. 264.



ServiceClass1 och ServiceClass2 implementerar InterfaceA respektive InterfaceB.
ServiceClass3 och ServiceClass4 implementerar InterfaceA och InterfaceB indirekt genom att implementera de abstrakta metoderna i sin basklass. Dessutom har de både ett beteende (operationen behaviour) som de delar och behov av att representera internt tillstånd (attributet state). För att unvika reundans låter vi AbstractClass definiera bevaviour/state. Här finns även möjligheten att definiera ”default behaviour” genom att låta AbstractClass implementera något eller båda av interfacen InterfaceA och InterfaceB.

UsingClass kan alltså få tillgång till implementationer av InterfaceA och InterfaceB genom instanser av en eller flera av serviceklasserna.

Interface vs type
Till diskussionen om klasser och interface hör också begreppet typ.

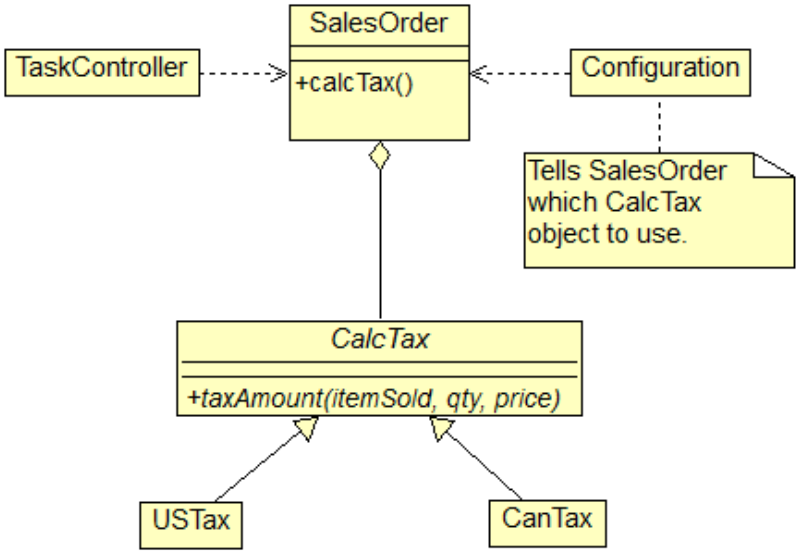
Vad menar vi när vi säger att ett objekt är av en viss typ? GoF diskuterar detta i det inledande kapitlet av ”Design Patterns”. Enligt GoF är ett objekts *typ* relaterat till dess *interface*, dvs. den uppsättning av meddelanden som det kan reagera på. *Ett objekt kan alltså ha flera typer och objekt av olika klasser kan ha samma typ.*

Principle of Healthy Skepticism
Avsnittet tar upp begränsningar och fallgropar i användandet av DP.

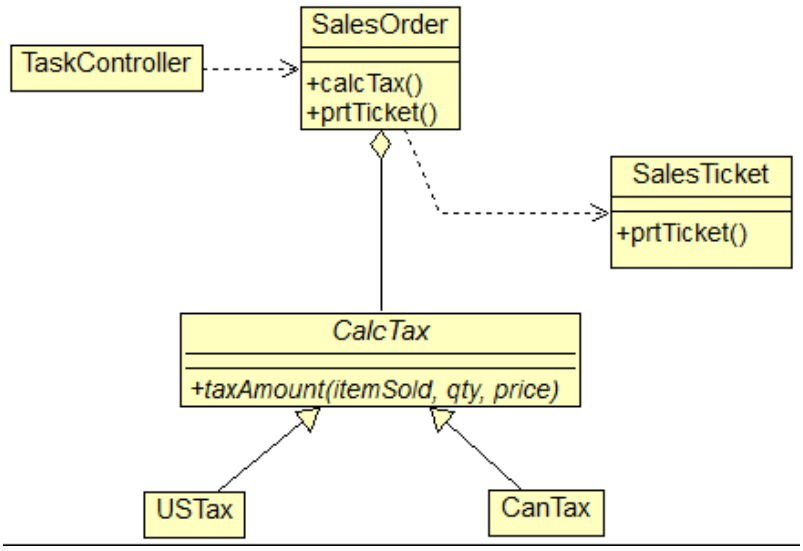
Kap 15 och 16 är inriktade mer mot analysprocessen och kan läsas kursivt eftersom de faller lite utanför ramen för kursen. Detta hindrar inte att det kan vara intressant läsning!
Delar av kap 15 ”Commonality/Variability” har behandlats tidigare.

Kapitel 17 ’The Decorator Pattern’

Vi återvänder till kapitel 9 och exemplet med SalesOrder och olika strategier för skatteberäkning.

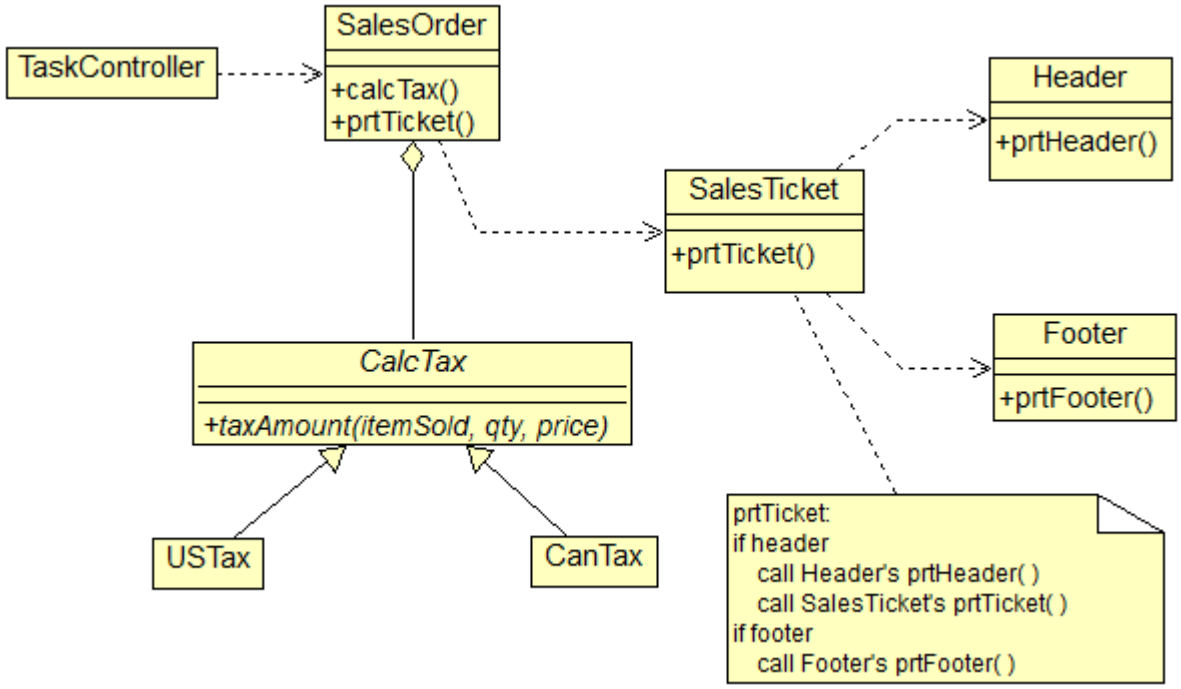


Om vi lägger till ett nytt krav, utskrift av kvitto, kan detta ske med klassen SalesTicket.



Nästa krav som dyker upp är att det på kvittot ska skrivas ut en ’header’ eller ’footer’ eller kanske både och.

En *enkel* design för att göra detta är att lägga till kontrollsatser i SalesTicket.



Om vi ska skriva ut olika varianter på header och footer kan man tänka sig ett Strategy DP för header och ett annat för footer.

Men…
Hur ska vi hantera krav på olika typer av header/footer och dessutom varierande antal beroende på kvittotyp?

Problemen tornar upp sig när antalet kombinationer växer...

Lösningen kan vara ett *Decorator* DP!

DP Decorator

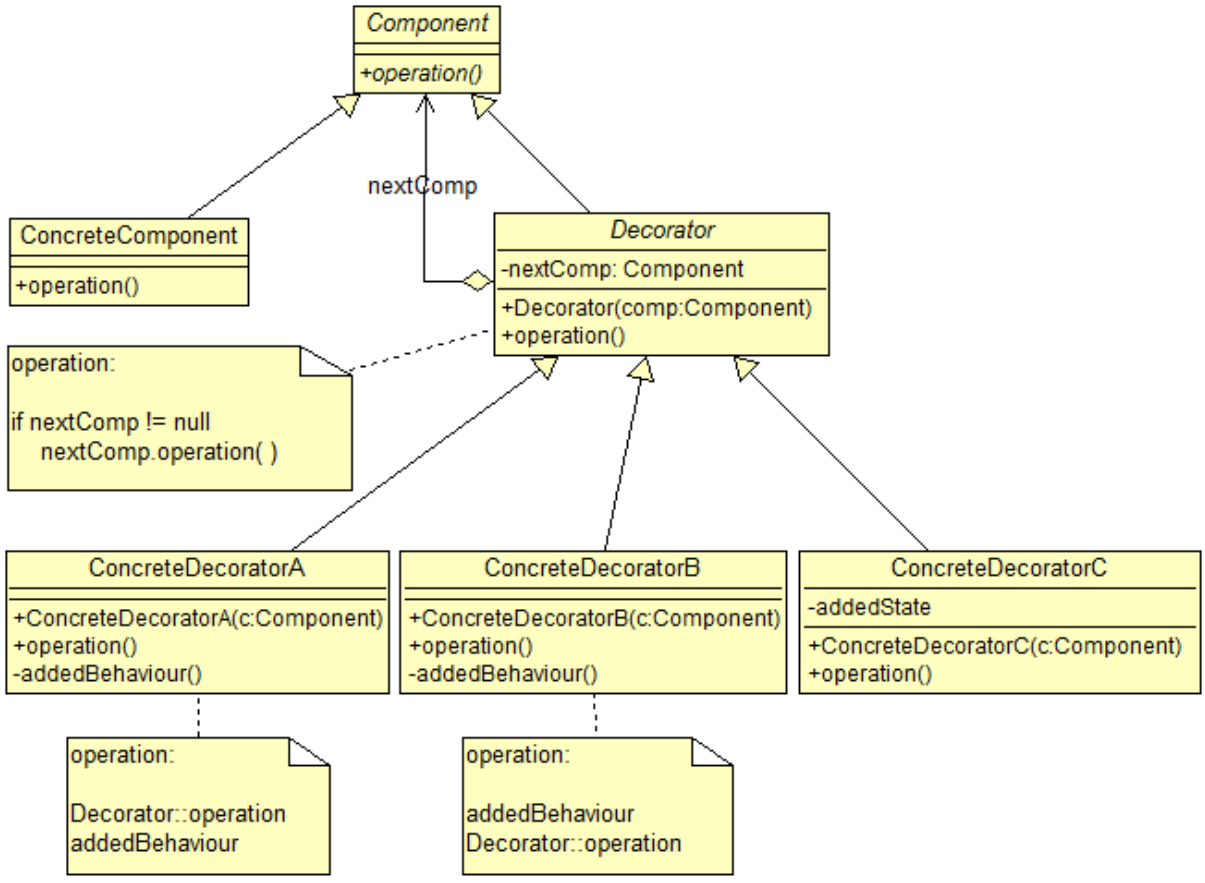
Syfte:
” *Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality*.” (GoF)

Ett viktigt ord här är *dynamically*, innebörden är att Decorator kan tillämpas även under runtime.
Decorator arbetar genom att skapa en länkad kedja av objekt vars operationer ska tillämpas i en viss ordning. Kedjan börjar med en *decorator* och slutar alltid med det ursprungliga objektet som ska ”dekoreras”.
Varje objekt i kedjan känner till nästa objekt.

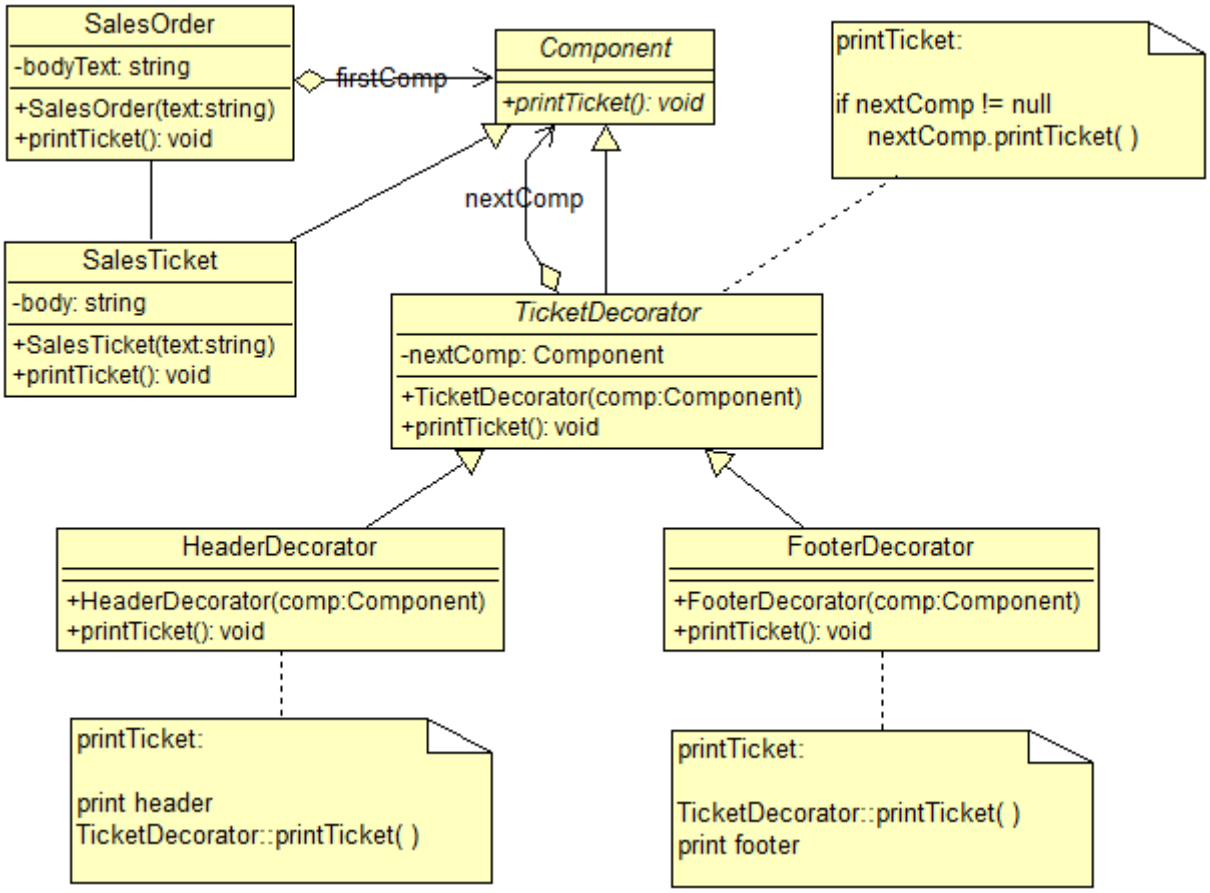
- För att kunna använda en polymorfisk metod, *operation( )*, måste alla decorators och det ursprungliga objektet ha en gemensam basklass - *Component*.

- Klienten anropar operation( ) för det första objektet i kedjan

Struktur:



Följande klassdiagram visar Decorator instansierat för exemplet SalesOrderTest.



- SalesTicket::printTicket( ) skriver ut kvittots kropp

- TicketDecorator::printTicket( )
  - anropar printTicket( ) för nästa objekt i kedjan om kedjan inte är slut

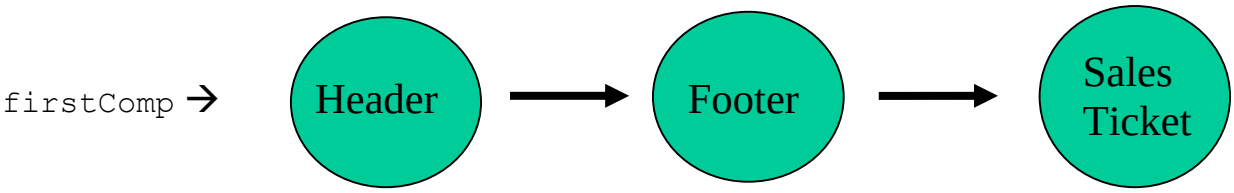
- HeaderDecorator::printTicket( )
  - skriver ut header-text
  - anropar TicketDecorator::printTicket( )

- FooterDecorater::printTicket( )anropar
  - TicketDecorator::printTicket( )
  - skriver ut footer-text

För enkelhetens skull låter vi konstruktorn i SalesOrder bygga upp kedjan med Components:

```
SalesOrder(string text)
     :bodyText(text)
{
     // Build chain of decorators
     // firstComp points to the start of chain
     firstComp = new SalesTicket(bodyText);
     firstComp = new Footer(firstComp);
     firstComp = new Header(firstComp);
```

Detta ger oss följande länkade kedja:



Anropskedja vid anrop till `SalesOrder::printTicket( )`:

```
SalesOrder::printTicket
    Header::printTicket
        Output : header text
        Decorator::printTicket (from Header)
            Footer::printTicket
               Decorator::printTicket (from Footer)
                   SalesTicket::printTicket
                       Output : body text
                   ←
               Output: footer text
            ←
        ←
    ←
←
```

Observera att i exemplet är det klienten som bygger upp kedjan med decorators. Normalt görs detta av något annat objekt, t.ex. någon form av factory-objekt eller konfigurationsobjekt. SalesOrder behöver inte känna till hur kvittot i detalj ska vara utformat. Det är en viktig egenskap hos DP Decorator: dekorationerna kan göras oberoende av, och dolt för, det objekt som dekoreras.

Decorator DP ger möjlighet att dela upp problemet i två delar:

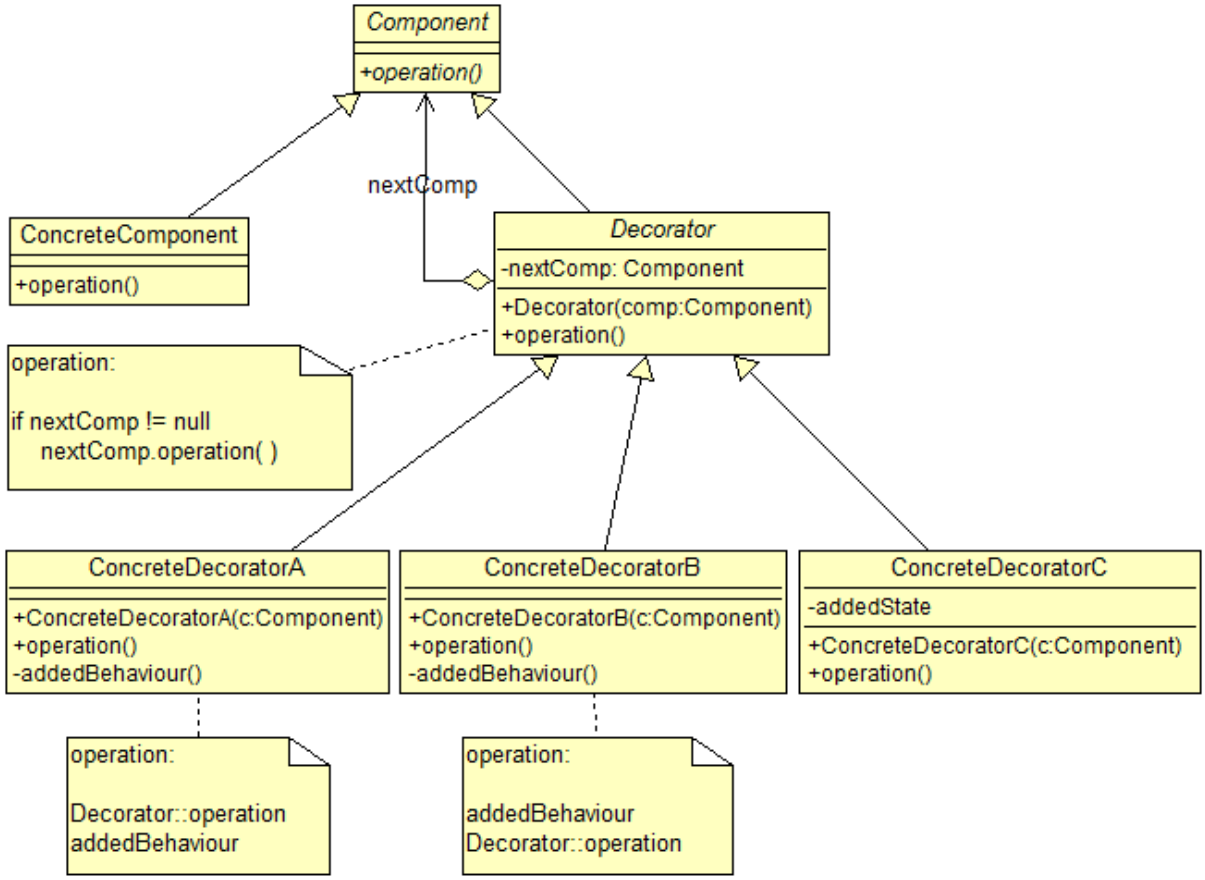
1. Implementation av de objekt som tillför ny funktionalitet (Decoraters)

2. Hur objekten ska organiseras i varje speciellt fall, dvs. hur kedjan ska byggas.

➔ Vi får en ökad ’cohesion’ eftersom Decoraters endast befattar sig med sin funktionalitet och inte hur de länkas ihop med andra objekt

Sammanfattning DP Decorator

- Struktur



- Syfte
  - Att dynamiskt lägga till extra ansvar/uppgifter till ett objekt. Decorator erbjuder ett flexibelt alternativ till sub-classing för utökad funktionalitet.

- Problem
  - Ett objekt utför de grundläggande uppgifterna du efterfrågar men du vill dessutom lägga till annan funktionalitet före och efter basfunktionerna

- Lösning
  - Erbjuder utökning av funktionalitet utan att använda sub-classing.

- Deltagare
  - ConcreteComponent får utökad funktionalitet genom Decorators.
  - Ibland är det klasser deriverade från ConcreteComponent som ger basfunktionaliteten och då är ConcreteComponent en abstrakt klass.
  - Component definierar interfacet för övriga klasser

- Konsekvenser
  - De utökade funktionerna ges genom små objekt som dynamiskt kan läggas till före eller efter funktionaliteten hos ConcreteComponent.

- Implementation
  - I Decorator-klasserna placeras de nya funktionerna före eller efter anrop till efterföljande objekt i kedjan för att uppnå korrekt ordning.
  - Skapa en abstrakt klass (Component) som en basklass för den ursprungliga klassen (ConcreteComponent) och de nya Decorator-klasserna.
  - Den hoplänkade kedjan av objekt ska avslutas med en ConcreteComponent.

- Använd Decorator när du vill
  - du vill lägga till uppgifter till ett objekt dynamiskt och transparent, dvs utan att andra objekt berörs.
  - utvidgning genom sub-classing är opraktiskt, t.ex. när antalet nya klasser för att täcka alla möjliga kombinationer blir för stort.