Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 3 – Vidgade vyer och DP Strategy

Designmönster med C++

Syfte:	Lektionen ger fördjupade kunskaper inom det objektorienterade området och introducerar DP Strategy.
Att läsa:	Design Patterns Explained kapitel 8 – 9 sid 115 – 157.

Kapitel 8 'Expanding our horizons'

Objekt

Ett traditionellt synsätt på begreppet objekt är att objekt är data med metoder, 'smarta data'.
Ett mer användbart synsätt är att ett objekt är något som *ansvarar* för att utföra bestämda uppgifter. Genom dessa definieras objektets beteende.
➔ *Fokusering på vad objektet förmodas utföra och vad det ansvarar för.*

Med detta synsätt kan vi bygga mjukvara i två steg:
1. Gör en preliminär design baserad på *uppgifter/ansvar* (responsibilities) utan att ta hänsyn till detaljer.
2. Implementera

Fokus på uppgifter/ansvar underlättar specifikationen av det publika interfacet till en klass.

T.ex. för ett Shape-objekt ska ansvara för att
- veta var det är placerat
- kunna rita ut sig själv
osv

Detta ger oss direkt motsvarande metoder i det publika interfacet:
- getLoacation(...)
- drawShape(...)

Implementation och attribut kan tillföras i senare skede, under modelleringen är det mindre intressant.

Inkapsling

Begreppet *inkapsling* (encapsulation) bör innebära inte bara att 'gömma data' utan även

- att gömma implementationer
- att gömma deriverade klasser
- att gömma... whatever!

Inkapsling i Adapter-exemplet från lektion 11:

- Inkapsling av *data* → attribut i Square, Line osv är gömda för alla andra.
- Inkapsling av *metoder* → t.ex. implementationen av Circle's setLocation är gömd.
- Inkapsling av andra *objekt* → bara Circle känner till XXCircle

- Inkapsling av *typ* → klienter som använder interfacet Shape ser inte Line, Square osv.

Den utvidgade synen på inkapsling matchar synsättet att de olika interfacen blir naturliga skikt för inkapsling. Exempel:

- Nya Shape-klasser kan tillföras utan att detta påverkar klienter till Shape.
- Implementationen av XXCircle kan ändras utan påverkan av klienter.

Många DP använder inkapsling för att arrangera objekten i olika lager med interfacen som gränsskikt
→ ändringar kan ske i implementationen av ett lager utan att övriga lager behöver påverkas
→ svag koppling (loose coupling) mellan de olika lagren

Den klassiska motivet för att använda arv är att det tillåter *återanvändning* av en klass genom *specialisering*: en ny klass deriveras från en superklass och beteendet anpassas genom omdefiniering av metoder. Detta fungerar naturligtvis men konceptet har en del svagheter:

- Kan leda till ’weak cohesion’. Om många olika specialiseringar görs (kanske i flera led) kommer vi längre och längre ifrån det ursprungliga konceptet.

- Minskar möjligheterna till återanvändning. De specialiserade klasserna kan innehålla kod som lika bra skulle kunna användas i kombination med andra klasser än superklassen. Specialiseringarna måste då göras även för dessa klasser. Vi får i princip samma kod duplicerad i en annan klasshierarki. Detta bryter mot en regel som kallas *’Once and only once rule’*, mer om den längre fram.

- Ju flera saker som kan variera samtidigt, desto fler klasser måste deriveras för att täcka in alla kombinationer. Vi får en ’kombinatorisk explosion’.

Kursboken argumenterar för att främst använda arv på ett annat sätt.

Följande citat är från GoF:

’Consider what should be variable in your design. This approach is the opposite of focusing on the cause of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without

redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns.'

Författarna av kurboken sammanfattar detta med orden

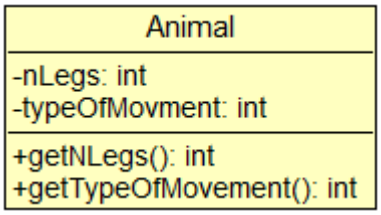
"Find what varies and encapsulate it."

Innebörden i detta är att klasser ska använda *pekare/referenser till abstrakta klasser (interface)*. Dessa pekare/referenser kan referera till objekt av olika deriverade klasser (aggregation) som representerar *variationer i beteende*.

Vilka objekt som faktiskt används är gömt för klienter till klassen: inkapsling.

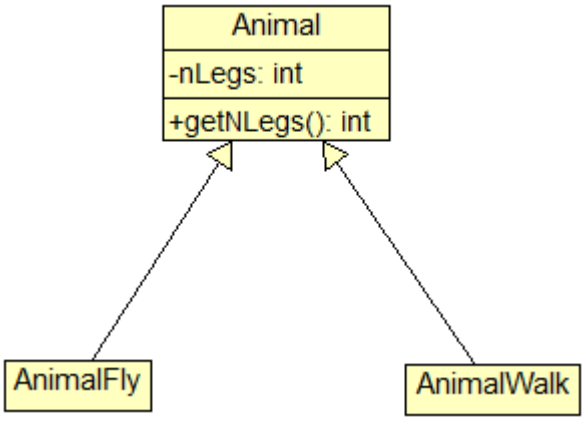
Dessutom kan det refererade objektet bytas ut mot ett annat under runtime utan klientens vetskap. Detta är också ett exempel på 'loose coupling'.

För att illustrera detta diskuterar boken olika sätt att modellera rörelsesätt hos djur.



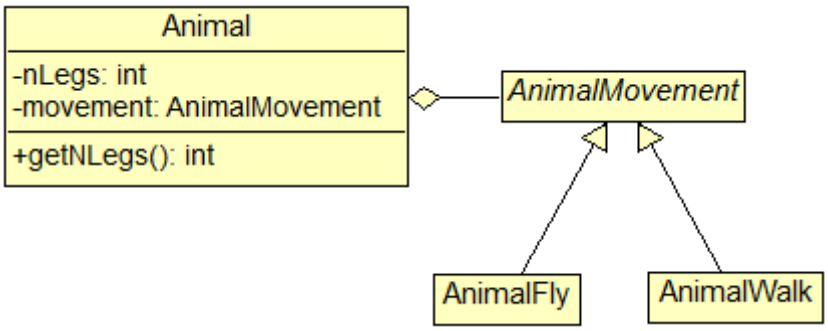
'Movement' motsvaras av attributet typeOfMovement som kodas för olika rörelsesätt.

Eller en deriverad klass för varje typ av movement.



Båda dessa sätt kan duga så länge EN egenskap varierar (och variationen inte ändras dynamiskt). I den första varianten måste dock klassen ändras om nya rörelsesätt ska modelleras. Gemensamt för båda är att de blir ohanterliga om flera egenskaper tillförs och tillåts variera. Vad händer om rörelsesättet ändras dynamiskt? En fågel kan ju gå ibland och flyga ibland...

"Find what varies and encapsulate it" ger den bästa lösningen:



Vi hanterar alltså variationer i beteende genom objekt som refereras via en superklass-typ (eller interface i Java). Objekten kan bytas ut dynamiskt, nya kan läggas till och Animal kan utvidgas med nya beteenden utan case-satser och utan att antalet klasser ökas exponentiellt.

'Commonality and Variability Analysis and Abstract Classes'

'Commonality' står för det stabila och långlivade 'gemensamma' konceptet i en arkitektur medan 'variability' står för anpassning/variation av konceptet. Gemensamt = stabilt!

'Variability' motsvarar de konkreta specifika fallen inom ett problemområde medan 'commonality' definierar de gemensamma koncept som binder dem samman.

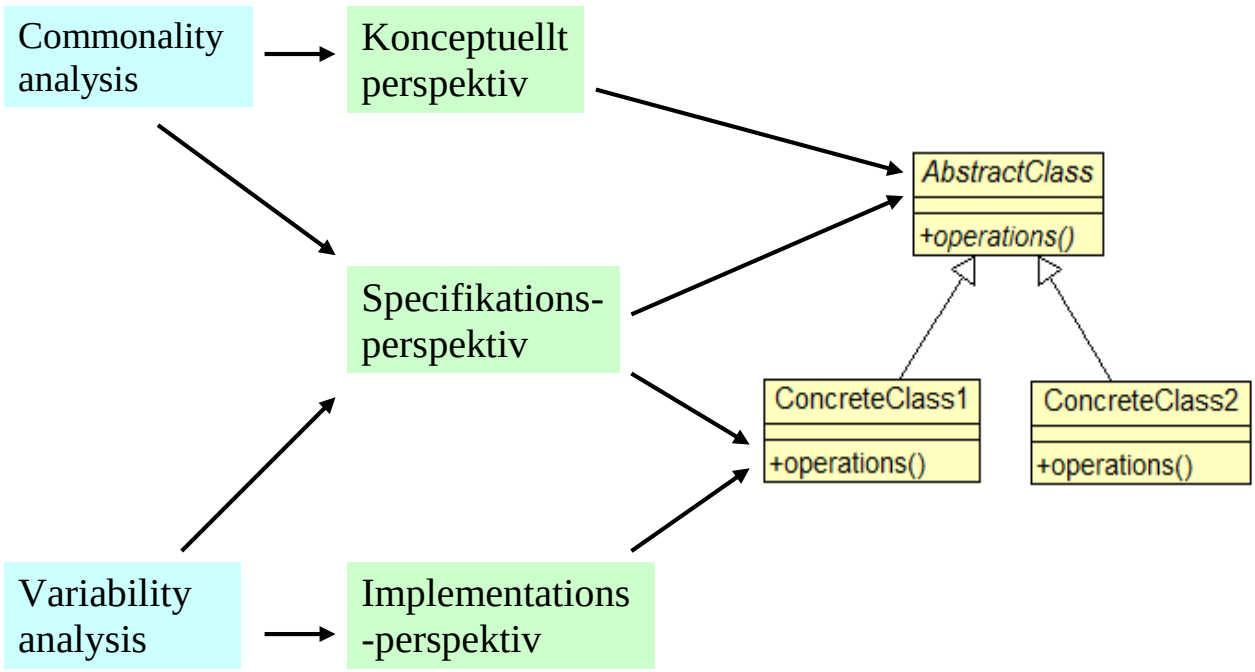
T.ex. en märkpenna, en white-boardpenna och en kulspetspenna har alla gemensamt att de är skrivverktyg. Detta är det gemensamma konceptet ('commonality') som är stabilt. Däremot är de olika egenskaper och beteende vilket avspeglas som 'variability'.

Boken menar att sökandet efter 'commonality' och 'variability' inom en domän ('domain') är ett mycket bättre sätt att hitta klasser och objekt än t.ex. det klassiska sättet att i en beskrivning leta efter substantiv som motsvarar klasser/objekt och därefter söka verb för att hitta metoderna.

Den gemensamma kärnan (’commonality') i ett koncept motsvaras på konceptuell nivå av en abstrakt klass (eller ett interface i Java) medan variationerna (’variability’) motsvarar olika subklasser eller implementationer.

När man definierar...	**Ställ frågan...**
En abstrakt klass eller ett interface	Vilket interface behövs för att hantera uppgifter/ansvar som klassen (och deriverade klasser) har?
Deriverade klasser	Hur kan jag implementera denna klass (denna variation) med det givna interfacet?

Nedanstående figur visar hur analys av ’Commonality/Variability’ är relaterad till de olika perspektiven på mjukvaruutveckling som nämndes i kapitel 1.



Kapitel 9 ’The Strategy Pattern’

Temat för kapitlets fallstudie är *Planera för förändring*. Boken använder termen ’Design with change in mind approach’ och betonar i sammanhanget tre av GoF’s principer:

- ’Program to an interface, not an implementation.’
- ’Favour object (aggregation) over class inheritance.’
- ‘Find what varies in your design and encapsulate it.’

Problem uppstår i fallstudien när klassen SalesOrder ska kunna hantera skatteberäkningar för nya länder. Med detta följer även variationer i t.ex. hur olika valutor och datumformat ska hanteras. Ett antal förslag på lösning diskuteras:

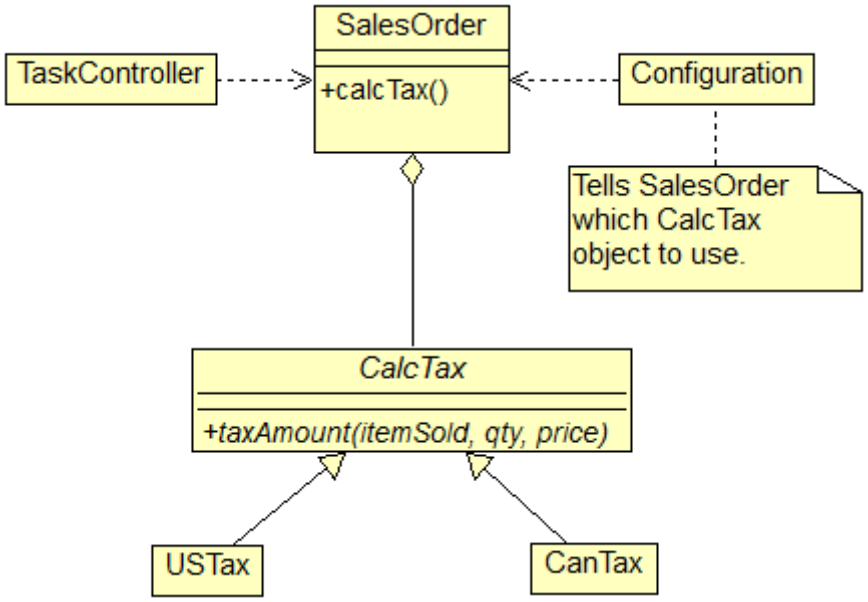
- ’copy and paste’ – kopiera liknande kod och gör de förändringar som krävs.
  Missar helt poängen med återanvändning och medför underhållsproblem med flera versioner av liknande kod.
- ’switches’ – switch-case koden blir snabbt komplex med nya switchar i switchar. Nya alternativ kan ändringar i flera switch-satser osv. Innebär också ’tight coupling’. Inget bra alternativ.
- ’function pointers and delegates’ i C++ resp. C# - kan inte hålla ‘state information’ per anropande objekt, inget alternativ (definitivt inte i Java...).
- ’inheritance’ – om arv används som ett sätt att återanvända kod genom att derivera en ny klass för varje ny variation kommer problem att uppstå när flera saker varierar samtidigt. Detta är ett vanligt sätt att missbruka konceptet arv på.

Bokens lösning:

1. Skatteberäkningen varierar →
   - skapa en abstrakt klass för konceptet CalcTax
   - derivera konkreta klasser för varje variation, t.ex UsTax och CanadianTax

2. Använd aggregation istället för arv →
   - istället för att derivera olika versioner av SalesOrder får SalesOrder innehålla en referens till ett passande CalcTax-object.

SalesOrder kan nu *konfigureras* med ett lämpligt CalcTax-objekt och delegera skatteberäkningen till detta.



Lösningen har ’strong cohesion’, skatteberäkningen är helt inkapslad i ett CalcTax-objekt, SalesOrder vet inga detaljer om detta. Det leder också till en ökad flexibiltet eftersom vi kan derivera nya CalcTax-klasser och använda dem efter behov utan att annan kod än Configuration-klassen behöver ändras.

Fallstudiens resultat är en tillämpning av DP Strategy.

DP Strategy

Syfte: ” *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it* .” (GoF)

Strategy bygger bl.a. på att

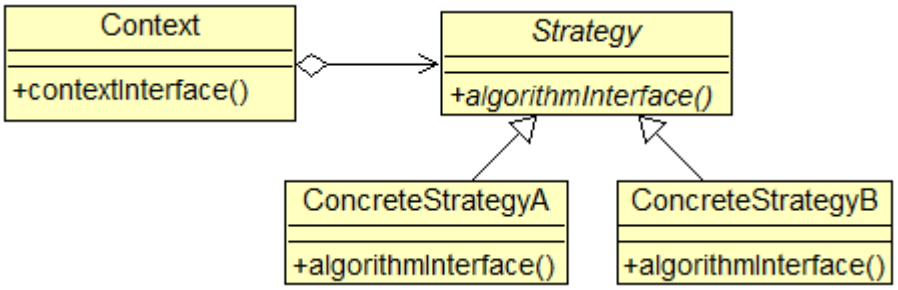
- objekt har ansvar för vissa uppgifter, olika specifika implementationer av dessa uppgifter kan göras genom polymorfism.

- det finns ett behov av att hantera olika implementationer av vad som konceptuellt sett är samma algoritm.

- god designpraxis är att separera olika beteenden inom ett problemområde från varandra. Den klass som är ansvarig för ett visst beteende kan då bytas ut mot en annan utan att påverka övriga beteenden.

Sammanfattning DP Strategy

- Struktur



- Syfte
  - Tillåter dig att använda olika regler eller algoritmer beroende på sammanhanget.

- Problem
  - Valet av algoritm beror på klienten eller på data som ska bearbetas. Om det bara finns en algoritm som alltid ska användas behöver inte Strategy användas.

- Lösning

  - Separera valet av algoritm från implementationerna.
    Strategy erbjuder ett sätt att konfigurera en klass med ett av flera utbytbara beteenden.

- Deltagare
  - Strategy specificerar interfacet till algoritmerna.
  - ConcreteStrategies implementerar de olika algoritmerna.
  - Context använder en ConcreteStrategy genom en referens till Strategy. Context vidarebefordrar en begäran från sin Client till sin Strategy

- Konsekvenser
  - Switch- och villkorssatser kan elimineras
  - Alla algoritmer måsta ha samma interface

- Implementation
  Interfacen för Strategy och Context måste ge en ConcreteStrategy ett sätt att accessa data från Context och tvärtom.
  1. Context kan skicka data som parametrar till Strategy, dvs ta data till Strategy. Detta ger särkoppling mellan Strategy och Context.
  2. Context kan skicka en referens till sig själv (this) som parameter till Strategy och låta Strategy själv hämta de data som behövs.
  3. Klassmallar (generics) kan användas för att konfigurera en klass med en Strategy. Tekniken kan bara användas om
     a) lämplig Strategy kan bestämmas vid compile-time och
     b) den inte behöver ändras under run-time.

- Använd Strategy när
  - många relaterade klasser endast skiljer sig åt i sitt beteende.
  - du behöver olika varianter av en algoritm och dessa varianter är implementerade i en klasshierarki.
  - en algoritm använder data som en klient inte ska känna till. Strategy döljer implementationsdetaljer!
  - en klass kan ha många beteenden och dessa uppträder i multipla flervalssatser. Ta bort grenarna och låt de bli egna Strategy-klasser.