Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 4 – DP Bridge

Designmönster med C++

Syfte:	Lektionen introducerar DP Bridge.
Att läsa:	Design Patterns Explained kapitel 10 sid 159 – 192.

Kapitel 10 ’The Bridge Pattern’

DP Bridge

Syfte:
” *Decouple an abstraction from its implementation so that the two can vary independently*.” (GoF)

Obegripligt?
Med ’implementationer’ menas de objekt som de abstrakta klasserna använder för att implementera sina abstraktioner med och INTE de abstrakta klassernas deriverade klasser!

Detta kan vara svårt att ta till sig. DP Bridge är ett kraftfullt designmönster som kan kräva lite tid att förstå. Det tillämpar principerna

- ’Find what varies and encapsulate it.’
- ‘Favor aggregation over inheritance.’

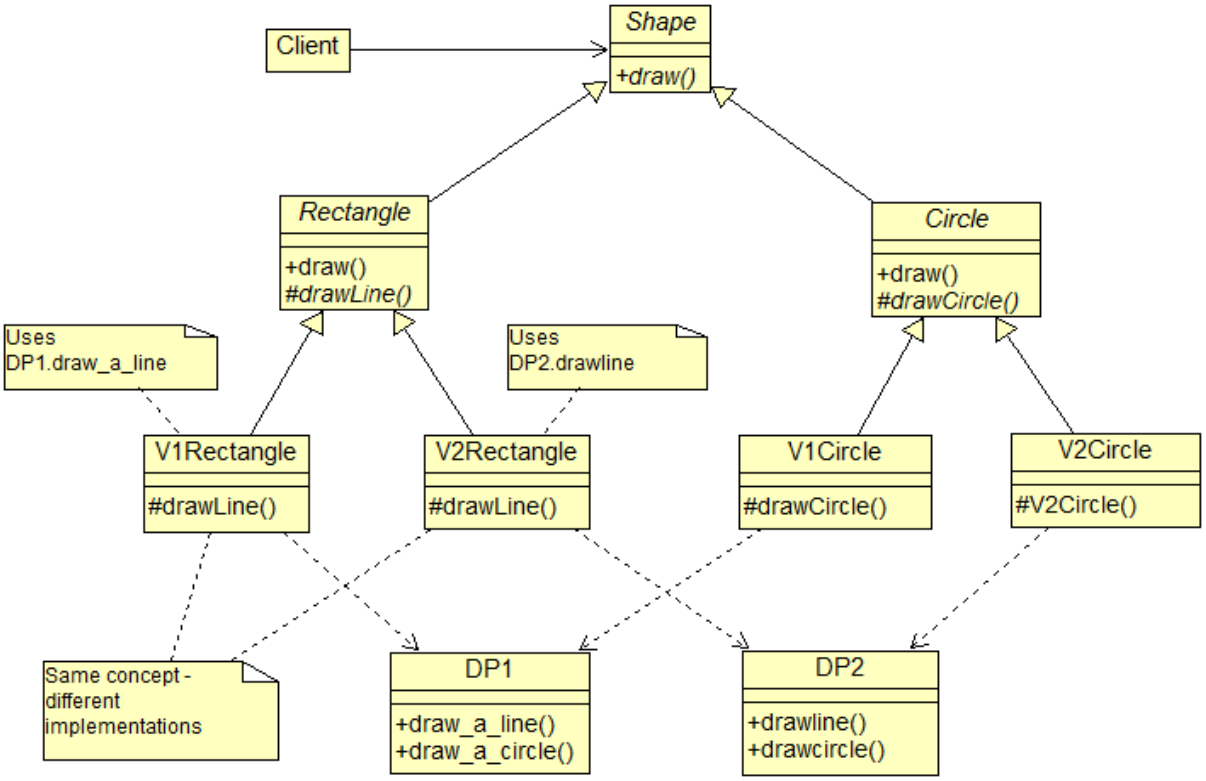
Ett motiverande problem

Vi ska skriva ett program som kan rita rektanglar och cirklar med hjälp av något av två ’drawing packages’, DP1 och DP2 (jag använder hellre ’drawing packages’ istället för bokens ’drawing programs’) .
Information om vilket bibliotek som ska användas kommer att finnas under exekveringen i samband med att objekten ska instansieras.

- DP1 har metoderna draw_a_line() och draw_a_circle()
- DP2 har motsvarande drawline() och drawcircle().

Karaktäristiskt för exemplet är att det innehåller
- Variationer i abstraktioner av konceptet Shape: Rectangle och Circle.
- Variationer i hur dessa koncept implementeras: DP1 och DP2.

En lösning uteslutande baserad på arv kan ha följande utseende:



Lösningen fungerar men lider av ’kombinatorisk explosion’. Om vi blandar in ytterligare ett ’drawing package’ och samtidigt ska hantera en tredje Shape får vi 3 x 3 = 9 konkreta Shape-klasser på rad 3 i diagrammet. Lösningen skalar dåligt beroende på ’thight coupling’. Varje konkret Shape-klass måste veta exakt vilken implementation den ska använda.

Vi får för mycket fokus på *'is-ness*’, dvs. vilken sorts objekt har. Detta leder i detta fall till överanvändning av arv. *Fokusering bör vara på objektens ansvarsområden* (responsibilities), vad de ska kunna utföra.

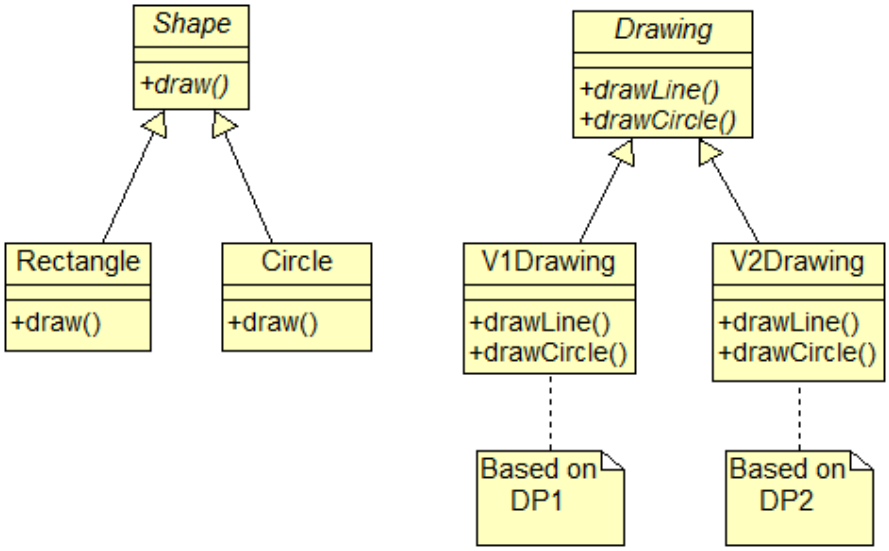
Vi behöver ett sätt att separera variationerna i Shape från variationerna i ’drawing packages’ - vilket är precis vad DP Bridge åstadkommer!

För att hitta fram till vår Bridge återvänder vi till budorden

- ’Find what varies and encapsulate it.’
- ‘Favour aggregation over inheritance.’

I exemplet har vi olika typer av Shape och olika ’drawing packages’. Våra ’common concepts’ kan representeras av de abstrakta klasserna

I nästa steg representerar vi variationerna i koncepten:



Den lösning vi eftersträvar bygger inte på arv utan på att ett av koncepten Shape och Drawing använder (’uses’) det andra.

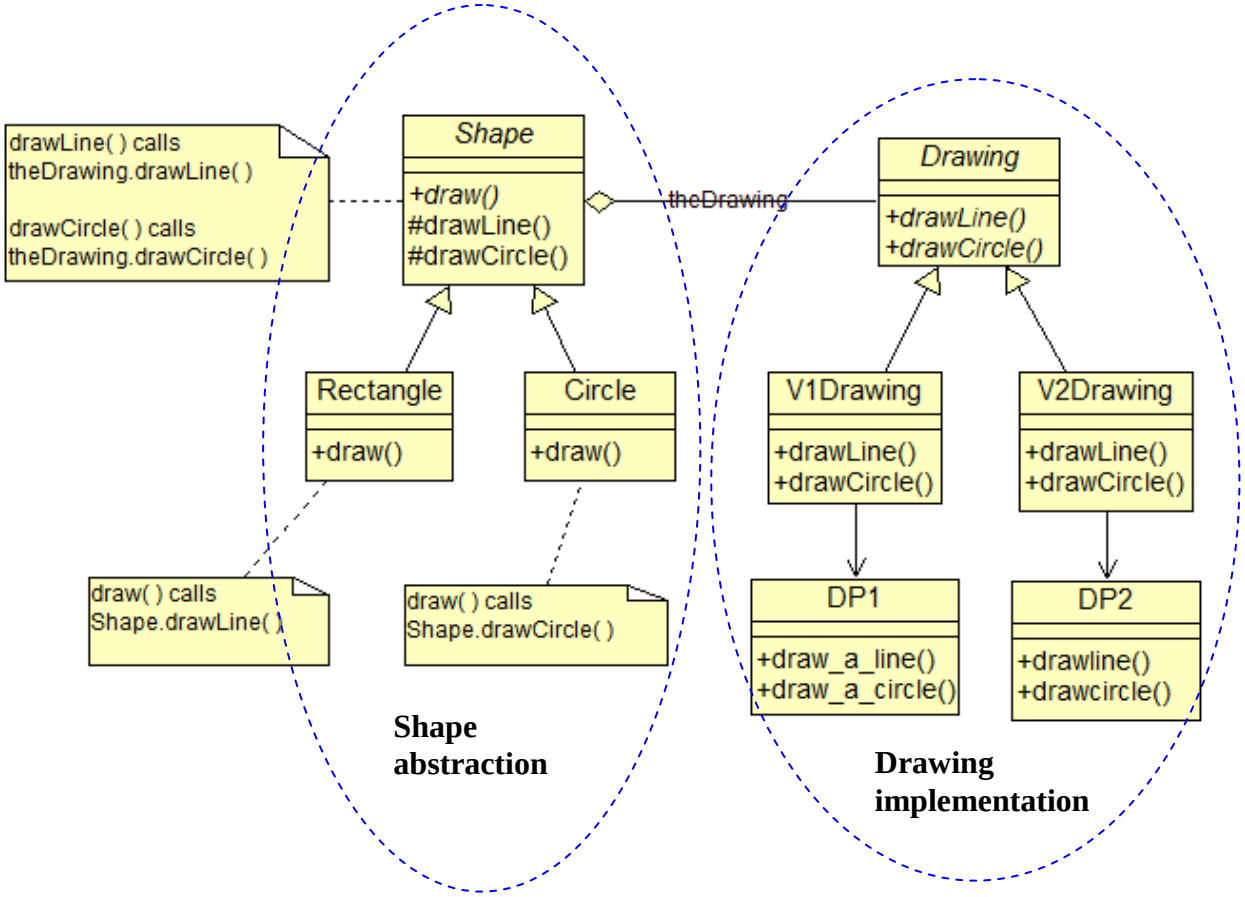
Hur ska då Shape och Drawing relateras till varandra? Klasserna kan relateras till varandra efter kriteriet 'vem-använder-vem' genom att använda komposition.

Alternativ 1: *Drawing använder Shape*
- Detta förutsätter att Drawing känner till hur olika konkreta Shapes ska ritas ut med hjälp av linjer och cirklar. Denna kunskap är dock inkapslad i de olika Shape-klasserna och är därför *inget alternativ*.

Alternativ 2: *Shape använder Drawing*
- En Shape kan utnyttja ett Drawing-objekt för att rita ut sig själv. Shape-objekten behöver inte veta vilket konkret Drawing-objekt de använder eftersom de refererar till interfacet i abstrakta Drawing.
Utmärkt !!

→ Vi har härlett *DP Bridge* !

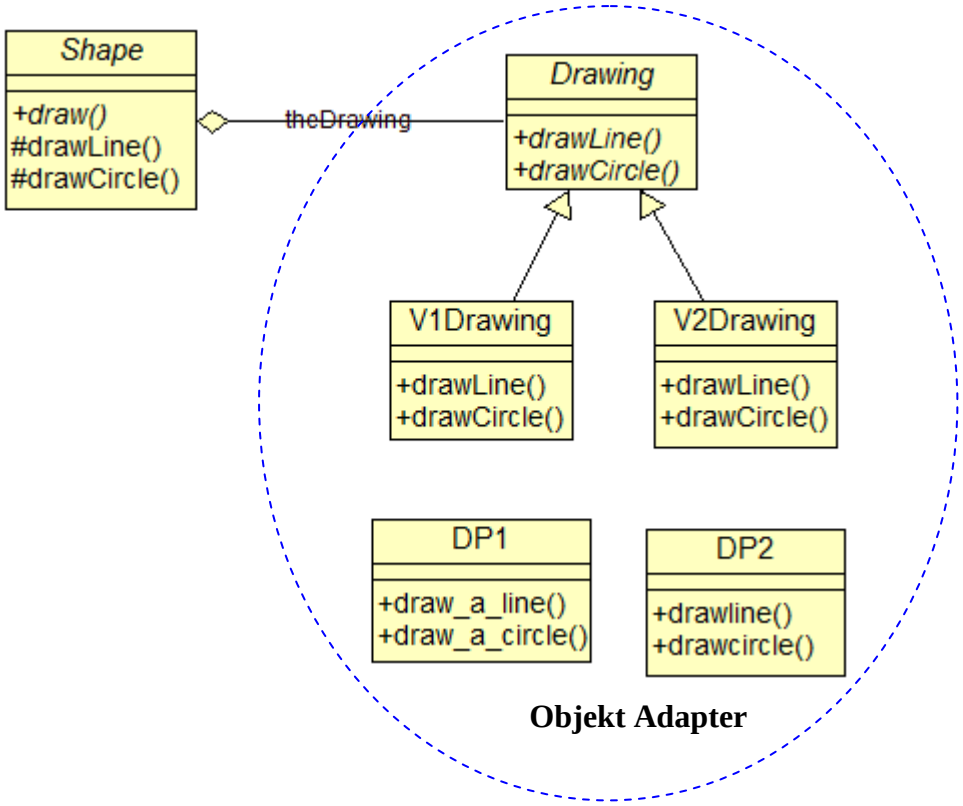
Lösningen där Shape aggregerar en Drawing kan beskrivas i följande diagram:



Lösningen, som är ett exempel på DP Bridge, tillämpar alltså principen ‘Favor aggregation over inheritance.’

Observera att DP Adapter har tillämpats för att ge DP1 och DP2 ett gemensamt interface,

→ vi har fått ett sammansatt designmönster (’Compund Design Pattern’) enligt följande klassdiagram.



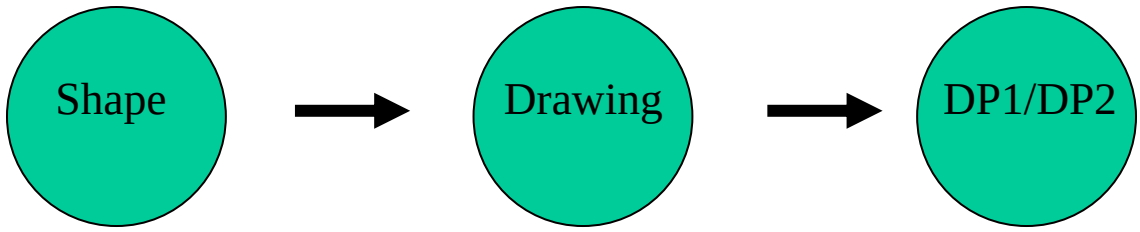
"One rule - one place"

Varför ska t.ex. Rectangle.draw( ) anropa Shape.drawLine( ) istället för att direkt anropa drawLine( )-metoden i det aktuella Drawing-objektet via referensen (theDrawing) i Shape? Det innebär ju en extra nivå av ’indirection’?

Motivet är att erfarenheten visar att duplicering av kod leder till felbenägenhet och problem vid underhåll/utveckling av ett program. Om du har en regel för hur en sak ska utföras ska den implementeras endast på ett ställe.

→ Låt Shape implementera de metoder som ska kommunicera med Drawing. Övriga Shape-typer använder det skyddade (protected) interfacet i Shape istället för att var och en kommunicera direkt med Drawing via referensen i Shape.

Under exekvering av en rit-operation är tre objekt aktiva:



Rectangle, Circle eller annat konkret Shape-objekt.

DP1- eller DP2-Drawing men Shape vet inte vilket eftersom de ser likadana ut.

DP1- eller DP2-objekt. Måste vara korrekt typ. Drawing-objektet vet vilken.

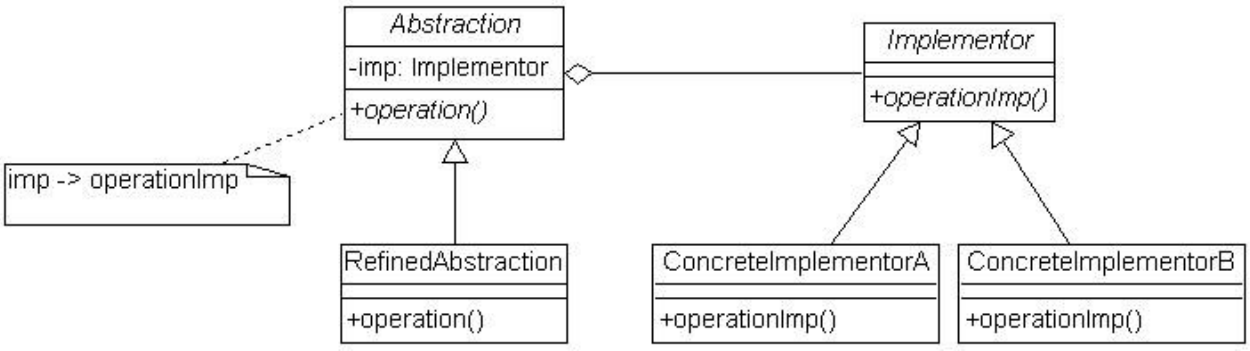
Patterns ger inte alltid helt perfekta lösningar men representerar kollektiva erfarenheter från många designers. Lösningarna blir oftast bättre än om en designer själv försöker 'uppfinna hjulet' igen. För Bridge-exemplet gäller att

- En ny konkret Shape kan kräva ändringar dels i Shape och dels av interfacet för Drawing samt eventuellt ändringar av dess implementationer. T.ex. Ellips kräver utökade ritfunktioner i implementationerna av Drawing.

- *Fördelen är att ändringarna är lokaliserade till vissa klasser.*

Sammanfattning DP Bridge (’Handle/Body’)

- Struktur



- Syfte
  - Att koppla isär en abstraktion från dess implementationer så att de kan variera oberoende av varandra.

- Problem
  - Deriverade klasser från en abstrakt klass ska använda flera olika implementationer utan att förorsaka en exponentiell tillväxt av antalet klasser.

- Lösning
  - Definiera ett interface som alla implementationer ska använda och låt de deriverade klasserna använda det.

- Deltagare
  - Abstraction definierar interfacet för klasserna som ska implementeras.
  - Implementor definierar interfacet för de specifika implementationsklasserna.
  - Klasser deriverade från Abstraction använder klasser deriverade från Implementor utan att veta vilken ConcreteImplementor som används.

- Konsekvenser
  - Minskar kopplingen mellan implementationerna och klasserna som använder dem. Klientobjekten har ingen kunskap om detaljer i implementationerna.

- Implementation
  - Kapsla in implementationerna i en hierarki med en abstrakt basklass.
  - Låt basklassen för abstraktionerna innehålla en referens till implementationernas basklass. I Java kan ett interface användas istället för en abstrakt klass för implementationen.

- Använd Bridge när
  - du vill undvika en permanent bindning mellan en abstraktion och dess implementation, t.ex. när valet av implementation ska göras under run-time. Både abstraktionerna och deras implementationer ska vara öppna för sub-classing. Bridge tillåter dig att kombinera de olika abstraktionerna med olika implementationer och att utvidga (extend) dem oberoende av varandra.
  - ändringar i implementationen inte ska påverka klienter så att de måste kompileras om.
  - när du vill dela en implementation mellan flera objekt (t.ex. genom referensräkning) och detta faktum ska döljas för klienter.