Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 11 – DP Composite

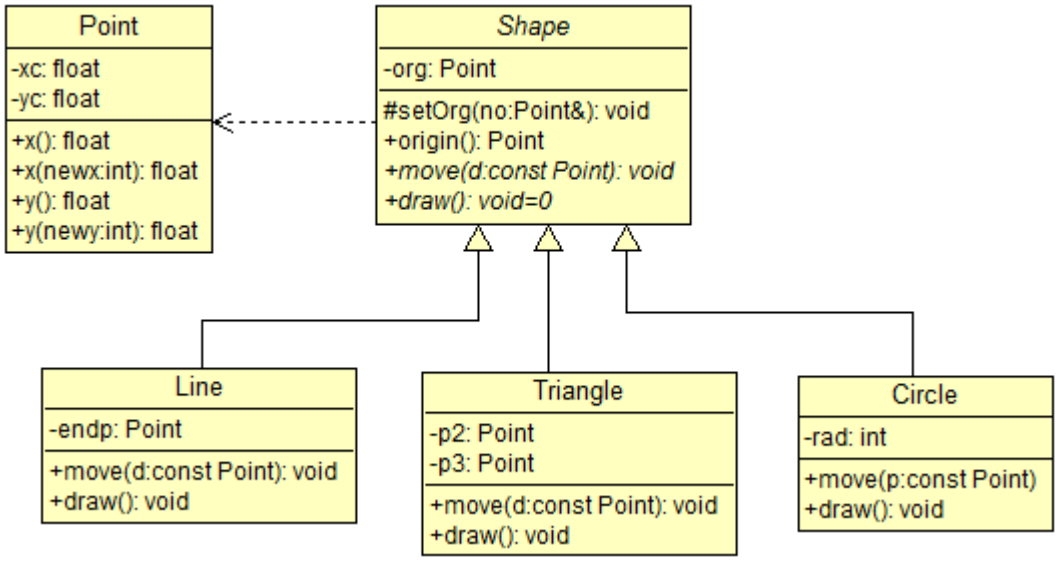
Designmönster med C++

Introduktion av designmönstret Composite för hantering av hierarkiska datastrukturer.

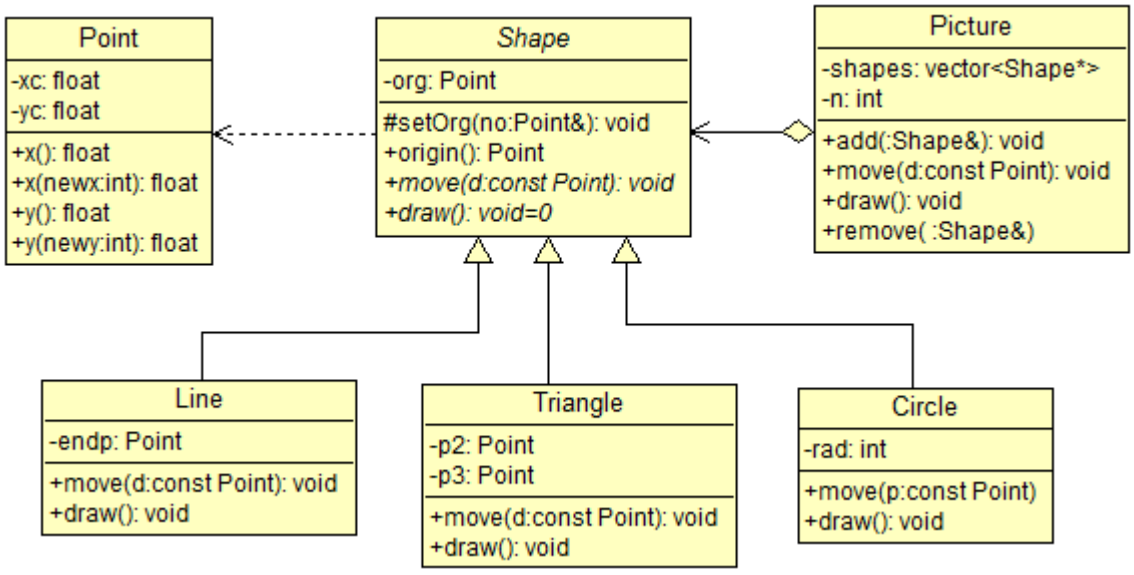
Akronymen SOLID

DP Composite

För att motivera mönstret ska vi titta på ett exempel. Antag att vi har ett att antal klasser för geometriska abstraktioner: Line, Circle, Triangle etc. För att kunna hantera dessa polyorfiskt så inför vi en gemensam abstrakt basklass, Shape. Alla Shape-klasserna är beroende av klassen Point, i beskrivningen av en Shape ingår minst en punkt (org) och den placerar vi basklassen. Vi får följande klassdiagram:



Alla figurer måste implementerar de abstrakta operationerna `draw()` och `move()`. Nu kan vi hantera hantera konkreta Shape-objekt via Shape-pekare/referenser. Objekten ska själv veta hur de ritas ut. Att flytta ett objekt är att flytta alla dess punkter på samma sätt.
Nu kan vi gå vidare och skapa en bild genom att kombinera ett antal figurer. Vi tillför klassen Picture som aggregerar ett godtyckligt antal Shape-objekt:



Att 'rita ut' en Picture innebär att alla ingående Shape-objekt får rita ut sig själva. Alla har sin egen implementation av `draw()` som binds dynamiskt:

```
void Picture::draw() const {
      for (int i = 0; i<shapes.size(); i++)
            shapes[i]->draw(); // let the objects draw themselves
}
```

Att flytta en Picture innebär att alla ingående objekt får flytta sig själva. Alla har sin egen implementation av move som binds dynamiskt:

```
void Picture::move(const Point d) {
      for (int i = 0; i<n; i++)
            shapes[i]->move(d); // let the objects move themselves
}
```

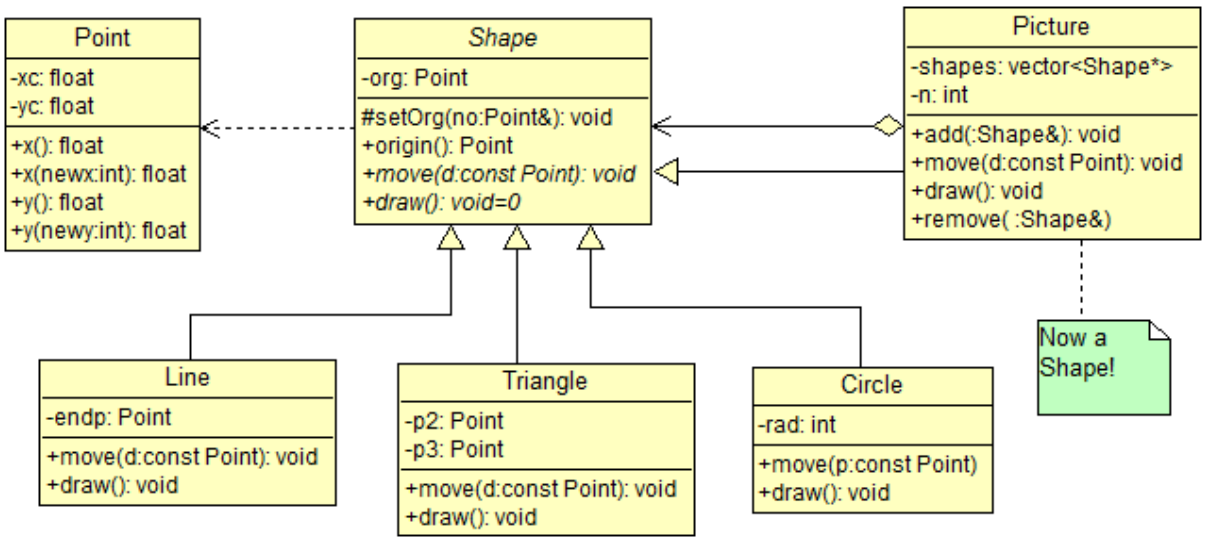
Vi kan nu skriva kod som

```
Line li(Point(10, 20), Point(100, 200));  // create a Line
Circle c(Point(100, 200), 100);          // create a Circle
li.draw();              // draw the line
c.draw();     // draw the circle

Picture pic;   // create a Picture
pic.add(li);  // add the line to the picture
pic.add(c);   // ..and the circle
cout << "==>Picture 1:" << endl;
pic.draw();      // draw the picture

pic.move(Point(-10, -5));  // move the picture
cout << "==>Picture 1 moved (-10,-5):" << endl;
pic.draw();      // draw the moved picture
```

Så långt är allt gott och väl.
Men om vi tänker oss att vi nu har flera olika Picture-objekt och kommer på idén att vi vill *lägga in en bild i annan bild* så fungerar inte lösningen längre.
För att kunna bygga upp en *hieraki* där en bild består av olika Shape-objekt och en annan bild som i sin tur består av en eller flera bilder och Shape-objekt osv. måste vi hitta ett sätta att behandla en Picture som om den vore något som kan läggas till en annan Picture. Lösningen är att göra även klassen Picture till en Shape!
Vi deriverar alltså Picture från Shape och får följande struktur:



Nu kan vi skriva kod som

```
Line li1(Point(10, 20), Point(100, 200));  // create a Line
Circle c1(Point(100, 200), 100);          // create a Circle

Picture pic1;   // create a Picture
pic1.add(li1);  // add the line to the picture
pic1.add(c1);   // ..and the circle
pic1.draw(); cout << endl;      // draw the picture

Triangle t1(Point(-10, 0), Point(20, 30), Point(40, 60));
Circle c2(Point(50, 75), 80);

Picture pic2;  // create another picture
pic2.add(t1); // add t1 to the picture
pic2.add(c2);  // ...and c2
pic2.draw();   cout << endl;   // draw the pictire

Picture bigPicture; // define a third picture
bigPicture.add(pic1); // add pic1 to the picture
bigPicture.add(pic2);  // ...and pic2
bigPicture.draw();   cout << endl;   // draw everything
bigPicture.move(Point(5, -5)); // move everything uniformly
```

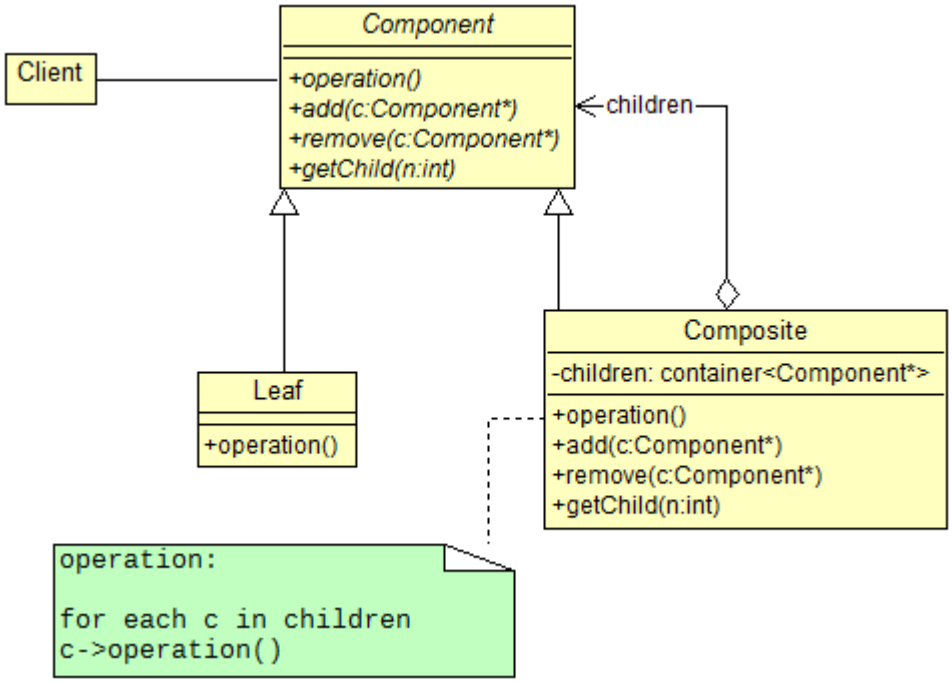
(Fullständig kod finns i Geometry.cpp)

Vi har genom detta exempel härlett designmönstret **Composite** (komposition) som tillhör de strukturella mönstren.

- Syfte

” *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*.” (GoF)

- Struktur



- Deltagare
  - Component definierar interfacet för delatagarna i kompositionen.
  - Leaf representerar ett enskilt löv-objekt i kompositionen. Ett löv har inga barn.
  - Composite definierar beteendet för komponenter som har barn, lagrar barnkomponenter och implementerar barn-relaterade operationer i Component-interfacet.

- Konsekvenser
  - Enskilda objekt kan kan komponeras till mer komplexa objekt vilka I sin tur kan komponeras, och så vidare på ett rekursivt sätt. Närhelst klientkoden förväntar sig ett enskilt objekt kan den också hantera en komposition.
  - Klientkoden kan behandla kompositioner och enskilda objekt på ett likartat sätt. Klienter behöver i de flesta fall veta om de hanterar ett enskilt objekt eller en komposition. Detta förenklar klient-koden.
  - Gör det enkelt att lägga till nya komponenter utan att ändra klientkoden.
  - För att kunna hantera operationer som bara är relevanta för enskilda löv-objekt måste interfacet Component göras bredd och innehålla dessa operationer. Detta leder till att operationer ibland måste ges tomma implementationer. T.ex. är operationer för att hantera barnobjekt inte relevanta för löv-objekten.

- Använd Composite när
  - Du vill representera en datastruktur som bildar en hierarki. I de flesta fall ska användaren inte behöva bry sig om huruvida det aktuella objektet är en hierarki eller ett enskilt objekt.

SOLID

Kursen har introducerat ett antal av de vanligaste designmönstren för objektorienterad programmering. Med denna bakgrund har du fått en terminologi som kommer att underlätta din kommunikation med andra utvecklare och ni kommer lättare att kunna utbyta tankar och idéer om design.

Utöver själva mönstren har ett antal grundläggande principer för objektorienterad programmering presenterats. Dessa är giltiga i all objektorienterad design och ska alltid finnas i bakhuvudet när man designar en lösning. En akkronym som är lätt att komma ihåg och som syftar på dessa regler är SOLID där varje bokstav står för en regel eller en princip:

S: Single responsibility principle (SRP)
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

O: Open/closed principle (OCP)
Classes should be open for extension, but closed for modification.

L: Liskov's substitution principle (LIP)
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

I: Interface segregation principle (ISP)
Many client-specific interfaces are better than one general-purpose interface.

D: Dependency inversion principle (DIP)
One should depend upon abstractions, not implementations.

Tack!
Örjan