Mittuniversitetet
MID SWEDEN UNIVERSITY

Lektion 2 -
Design patterns: Facade och Adapter

Designmönster med C++

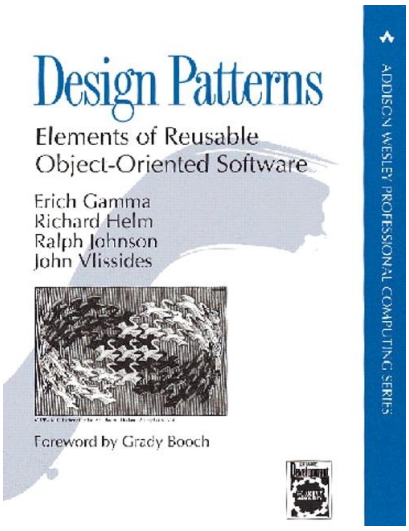
Syfte:	Introduktion av designmönster. DP (Design Pattern) Facade och Adapter
Att läsa:	Design Patterns Explained kapitel 5 - 7 sid 75 – 112.

Kapitel 5 'An introduction to Design Patterns'

En del idéer bakom begreppet designmönster kommer ursprungligen från byggnadsarkitektur. En klassisk bok är 'The timeless way of building' från 1979 av Christopher Alexander. Han argumenterar för att vad som är vackert och vad som har bra design inte är en fråga om personlig smak utan att det kan objektivt kan mätas. Alla goda konstruktioner har mycket gemensamt med varandra även om de är olika. Dessa likheter kallar han för patterns. Hans definition av patterns är

'*Varje mönster beskriver ett problem som återkommer gång efter gång i vår omgivning och beskriver därefter kärnan av lösningen på problemet på ett sådant sätt att man kan använda lösningen miljontals gånger utan att man gör det exakt lika två gånger.*'

Alexanders bok uppmärksammades av mjukvaruutvecklare på 90-talet. De ansåg att om idéerna var sanna för arkitektur och byggnader så borde de kunna tillämpas även inom mjukvarudesign. Mycket av det som vi idag kallar designmönster användes då redan av utvecklare på olika håll som erfarenhetsmässigt hade utvecklat smarta designlösningar som kunde återanvändas för olika problem. Dessa 'idiom' blev informellt spridda till en allt större skara men saknade både allmänt vedertagna namn och strukturerade beskrivningar. De mest allmänt användbara av dessa sammanställdes och dokumenterades 1995 av '*The Gang of Four*' (GoF): Erich Gamma, Richard Helm, Ralph Johnson och John Vlissides i boken '*Design Patterns: Elements of Reusable Object-Oriented Software*' som sedan dess har varit den mesta använda referensboken i ämnet. I och med denna bok myntades begreppet *Design Pattern (DP)*. De olika designmönstren fick i samband med detta de namn som de idag är kända under. En del DP har dock alternativa namn sedan gammalt.



I kursbokens beskrivning av varje DP ingår
-Namn som identifierar detta DP
-Syfte
-Problem som detta DP försöker lösa
-Lösningen som erbjuds
-Deltagare som samverkar i lösningen
-Konsekvenser/effekter vid tillämpning

-Förslag till implementation och exempelkod
-En generisk strukturbeskrivning (UML)
I GoF's text ingår även information om 'Known uses' för varje DP.

Motiv för att studera och använda designmönster:

- De erbjuder återanvändning av välbeprövad design, ett utmärkt tillfälle att lära av andra.
- De ger en etablerad vokabulär som underlättar kommunikation med andra utvecklare,om du säger "Singleton" så vet de vad du menar!
- Du och dina kolleger får enklare ett perspektiv på problemen på högre nivå. Det blir lättare att se skogen och inte bli skymd av alla träd!
- Du får en ökad insikt i de koncept som objektorienterad programmering bygger på och din kod blir lättare att underhålla och att utvidga.

Designmönster utnyttjar ständigt de grundbultar inom objektorienterad programmering som GoF framhåller:

- Programmera mot ett interface och inte en specifik implementation. Tillåt varierande implementationer under ett gemensamt interface.

- Välj helst aggregation/komposition före arv. Vid arv bestäms många egenskaper vid kompileringstillfället medan aggregation/komposition tillåter en referens att bindas till olika objekt under runtime vilket innebär större flexibilitet.

- Avgör vad som är gemensamt och vad som kommer att variera hos ett interface och dess implementationer. Gemensamt = = Stabilt! Hitta det som varierar och kapsla in det. Detta diskuteras mer ingående i lektion 4.

Utöver detta kan följande princip nämnas:

- Beroenden ska i allmänhet gå i riktning mot stabilitet,
  en viss komponent ska inte vara beroende av en mindre stabil komponent.

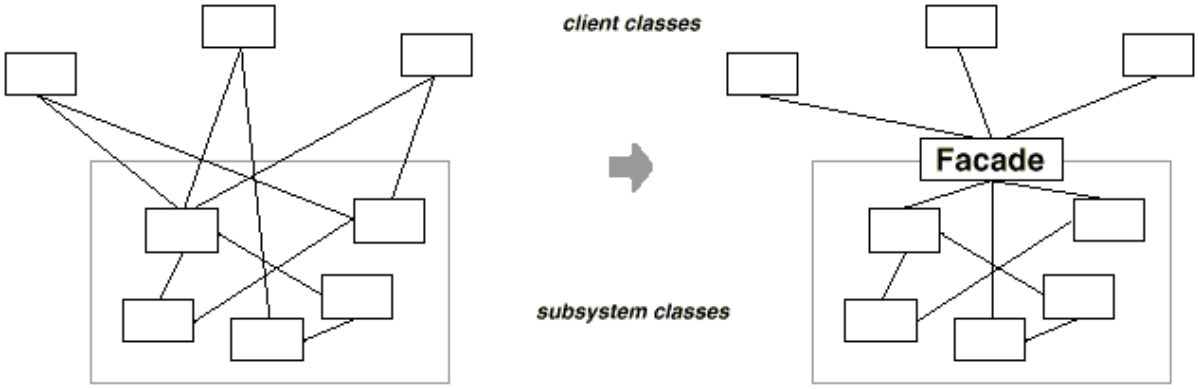
Viktiga konsekvenser av dessa principer är bl.a

- Abstraktion är bra!
- Polymorfism är bra!
- Publika dataobjekt och globala data är dåligt!

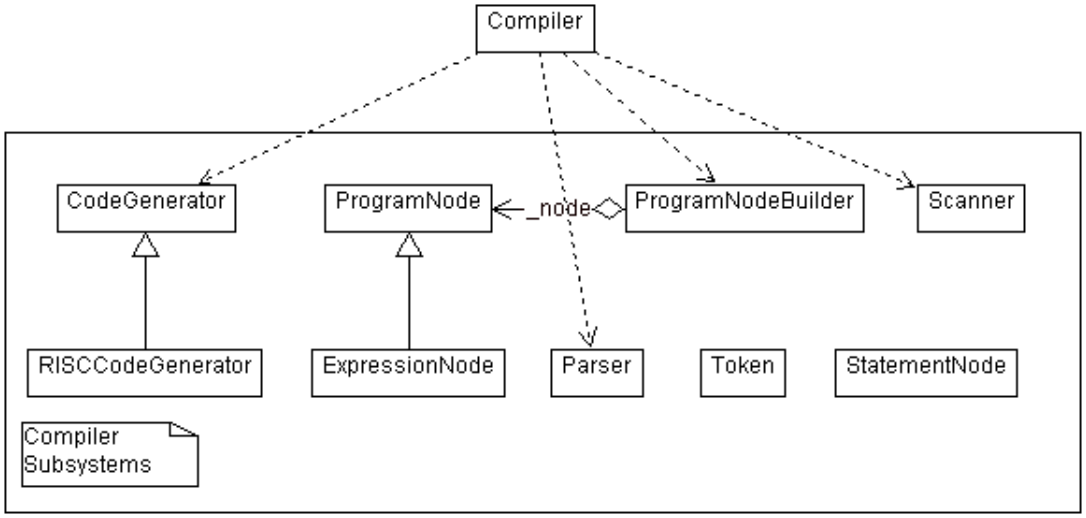
Kapitel 5 'The Façade pattern'

Syfte:
" *Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.* " (GoF)

Med Facade skapar vi alltså ett interface med endast de nödvändiga metoderna för att utnyttja delar av andra komplexa system. Kommunikationen med de underliggande systemen kapslas in i implementationen av Facade.

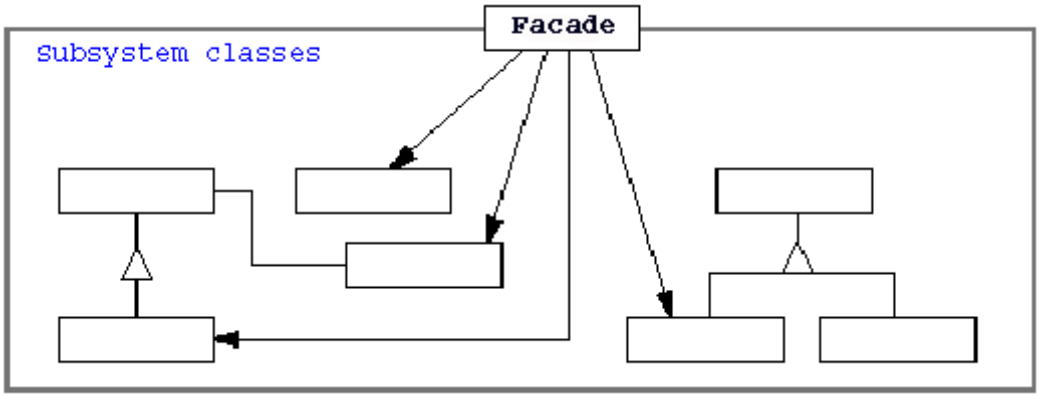


Ex. En kompilator
Klassen Compiler kan erbjuda ett förenklat publikt interface för att kompilera en fil utan att klienten behöver känna till det underliggande systemet.



Sammanfattning DP Facade

- Struktur



- Syfte
  - Du vill förenkla användningen av ett existerande system genom att definiera ett eget anpassat interface.

- Problem
  - Du vill använda bara en delmängd av ett komplext system eller du vill använda systemet på ett speciellt sätt

- Lösning
  - Facade erbjuder ett nytt interface för klienter till systemet.

- Konsekvenser
  - Facade förenklar användningen av ett system. Eftersom Facade ger ett anpassat och förenklat interface kommer kanske viss funktionalitet att saknas.

- Implementation
  - Definiera ett nytt interface med önskade egenskaper. Implementationen av den nya klassen använder det befintliga systemet
- Använd Facade när
  - du vill använda ett enkelt interface till ett komplext subsystem
  - det finns många beroenden mellan klienter och implementationerna som de använder. Genom att introducera en Facade kan subsystem kopplas loss från klienter och andra subsystem
  - du vill införa olika skikt för subsystemen. En Facade kan definiera accesspunkter till varje skikt. Om subsystemen är beroende av varandra kan de kommunicera genom sina respektive Facader.

Kapitel 6 'The Adapter pattern'

Syfte:
" *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces*." (GoF)

Innebörden är alltså att man skapar ett nytt interface till en klass som gör rätt saker men har "fel" interface.
Exempel:
Vi har en klasshierarki för figurer: Point, Line och Square deriverade från abstrakta basklassen Shape.

```
class Shape {
public:
      void setLocation() { .. }
      void getLocation() { .. }
      virtual void display() = 0;
      virtual void undisplay = 0();
      virtual void fill() = 0;
      void setColor() { .. }
}

class Line : public Shape {
public:
      virtual void display() { .. }
      virtual void fill() { .. }
      virtual void undisplay() { .. }
}

class Square : public Shape {
public:
      virtual void display() { .. }
      virtual void fill() { .. }
      virtual void undisplay() { .. }
};
```

Vi vill utvidga systemet med en ny Shape: Circle. Nu finns redan en cirkel-klass utvecklad och testad, XXCircle. XXCircle har all önskad funktionalitet men ett *annat interface* än Shape:

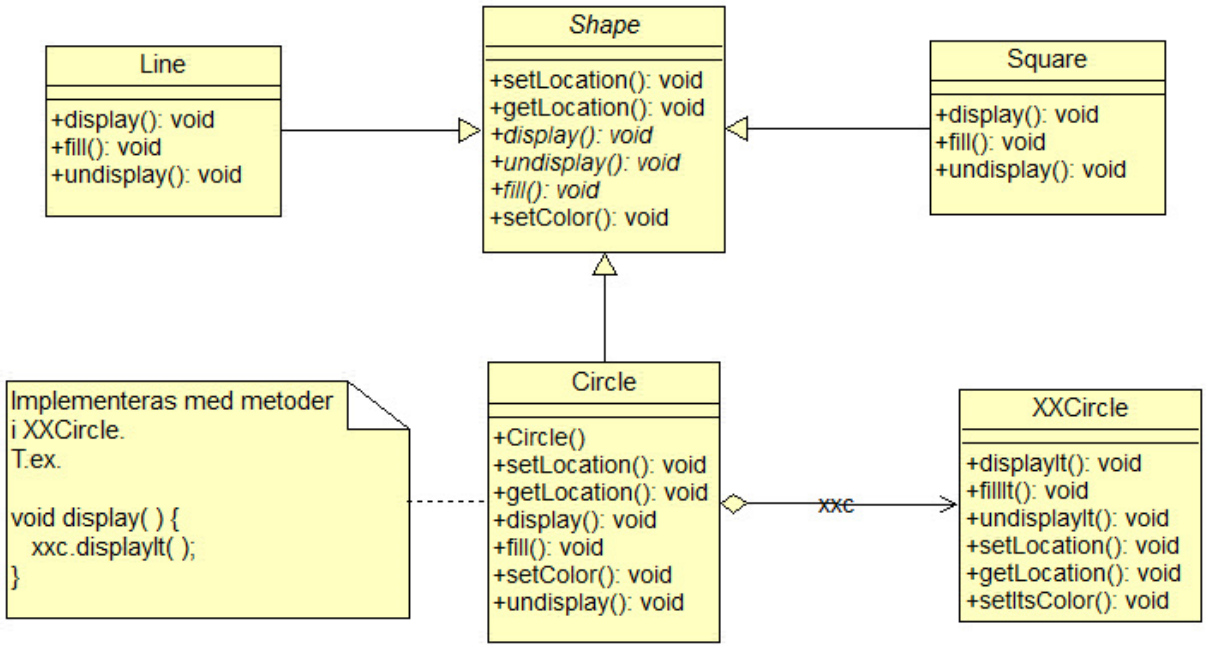
```
class XXCircle {
public:
      void displayIt() { .. }
      void fillIt() { .. }
      void undisplayIt() { .. }
      void setLocation() { .. }
      void getLocation() { .. }
      void setItsColor() { .. }
};
```

Ett annat problem är att XXCircle inte är deriverad från Shape - vi kan alltså inte direkt utnyttja polymorfism.

Frågan är: Hur kan vi utnyttja XXCircle för att implementera Circle som en klass av Shape-typ och ändå bevara polymorfism?

Lösning 1: Object Adapter

Låt XXCircle vara ett attribut i Circle och utnyttja XXCircle's interface *internt i Circle* för att implementera det publika interfacet i Circle.




```
// The Object Adapter
class Circle : public Shape {
public:
      Circle() { xxc = new XXCircle(); }
      void setLocation() { xxc.setLocation(); }
      void getLocation() { xxc.getLocation(); }
      virtual void display() { xxc.displayIt(); }
      virtual void fill() { xxc.fillIt(); }
      virtual void undisplay() { xxc.undisplayIt(); }
      void setColor() { xxc.setItsColor(); }

      private XXCircle xxc; // The Adaptee
};
```

- Alla operationer i det publika interfacet till Circle kan implementeras genom operationer i XXCircle.

- Polymorfismen behålls
  ➔ Circle kan användas som ett Shape-objekt men implementationen av XXCircle gör det egentliga jobbet.
- Inkapsling är inte bara att gömma data, Circle kapslar in XXCircle både vad gäller interface och implementation.

Lösning 2 – Class Adapter

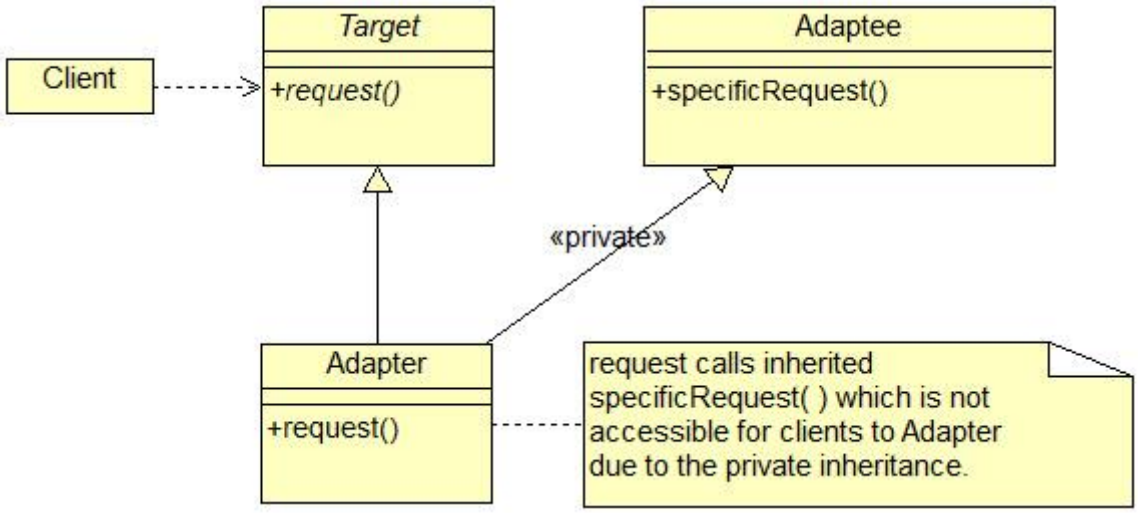
Denna lösning bygger på multipelt arv och att man kan reglera accessnivåer i arvet och passar därför bra i C++ (ej genomförbar i Java).

Man deriverar Circle med multipelt arv:

- Genom publikt arv från Shape fås det *publika interfacet* och Shape's implementationer.
- Genom privat arv från XXCircle får man tillgång till XXCircle's implementation utan att dess interface blir publikt i Circle – ett s.k. *implementationsarv*.

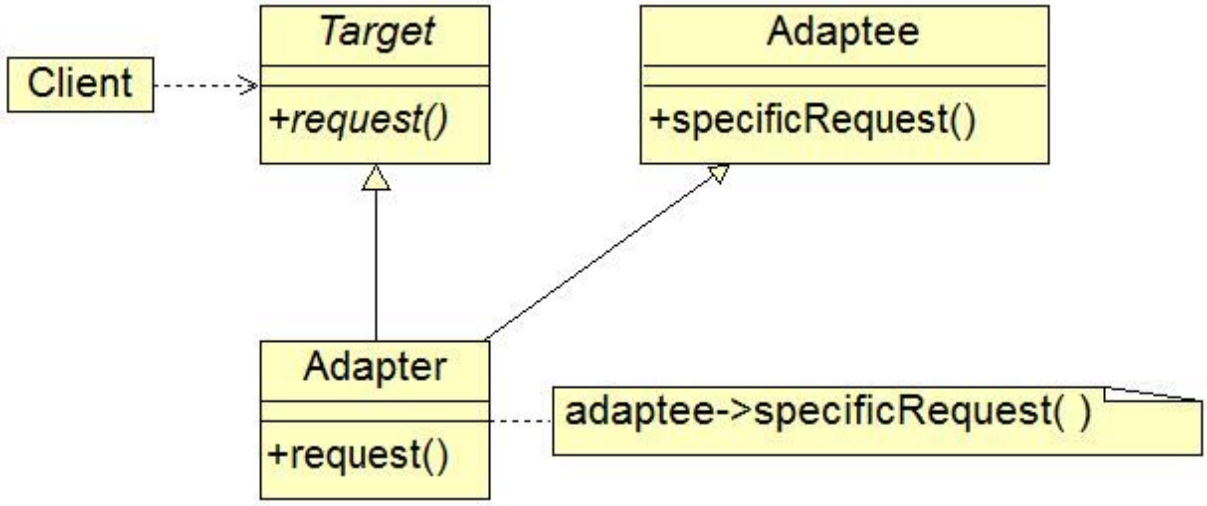
```
// Class adapter
class Circle : public Shape, private XXCircle {
public:
      void setLocation();
      void getLocation();
      void display();
      virtual void fill();
      virtual void undisplay();
      virtual void setColor();
};

// Implement the Shape interface using XXCircle's operations
...
void Circle::setLocation() {
      XXCircle::setLocation();
}

void Circle::display() {
      displayIt();
};
osv.
```

Strukturen kan generellt beskrivas som



Sammanfattning DP Adapter (även känt som ’Wrapper’)

- Struktur för Object Adapter



- Syfte
  - Att anpassa ett interface utanför din kontroll till ett givet interface.

- Problem
  - En klass vi vill utnyttja har önskvärda egenskaper men ett oönskat interface.

- Lösning
  - Adapter erbjuder en 'wrapper'-klass med önskat interface.

- Deltagare:
  - Adaptern anpassar interfacet hos det adapterade objektet (adaptee) så det överensstämmer med interfacet hos adapterns. Det tillåter klienter att använda det adapterade objektet som om det vore av adapterns typ.

- Konsekvenser
  - Adapter DP tillåter existerande klasser att passa in i nya klass-strukturer trots att interfacen inte är direkt kompatibla.

- Använd Adapter när
  - du vill använda befintliga klasser och deras interface inte matchar det du behöver
  - du vill skapa en återanvändbar klass som samarbetar med orelaterade klasser som kan ha icke kompatibla interface.

Facade vs Adapter

Både Facade och Adapter är ”wrappers” dvs. de kapslar in andra klassers funktionalitet i en ny klass. Vilken är egentligen skillnaden mellan dessa DP?

Kriterium	Facade	Adapter
Finns det existerande klasser att använda?	Ja	Ja
Finns det ett interface som vi måste anpassa oss till?	Nej	Ja
Ska objekten uppträda polymorfiskt?	Nej	Oftast
Är ett enklare interface nödvändigt?	Oftast	Nej

Studera medföljande exempel, framförallt exPowerSource där motorer med olika inkompatibla interface adapteras till ett gemensamt interface för att sedan användas på polymorfiskty sätt. Koden i exemplet ska längre fram användas i laboration 3.