Klasserelationer

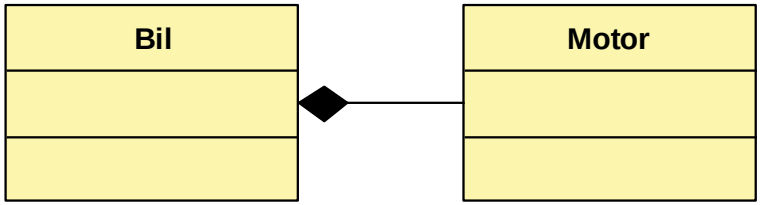
OOP – Lektion 2

Klasserelationer

- Der findes 4 typer klasserelationer
  - Komposition
  - Aggregering
  - Association
  - Arv

Komposition

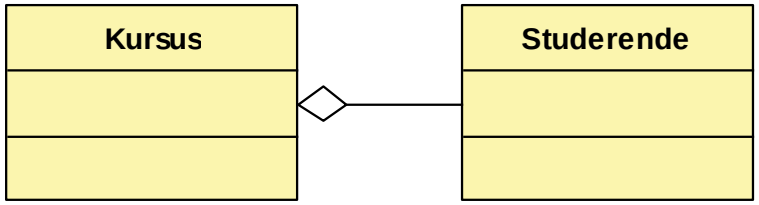
- Er en "*har* en/et"-relation (eller "består af") *med* ejerskab
- Eksempel:
  - En `Bil` *har* en `Motor`
  - En `Bil` *består af* en `Motor`
  - En `Motor` *er en del af* en `Bil`



- Bilen *ejer* motoren – motoren kan ikke bruges af andre biler
- De har samme levetid – når bilen nedlægges, nedlægges motoren også
- Komposition er den stærkeste relation
- Implementeringen kender du fra 1. semester ("Komposition.pdf")

Aggregering

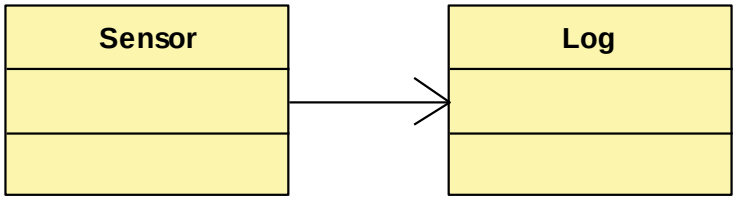
- Er en "*har* en/et"-relation (eller "består af") *uden* ejerskab
- Eksempel:
  - Et `Kursus` *har* en `Studerende`
  - Et `Kursus` *består af* `Studerende`
  - En `Studerende` *er en del af* et `Kursus`



- Kurset *ejer ikke* den studerende – en studerende kan deltage i flere kurser
- De har *ikke* samme levetid – hvis kurset nedlægges, nedlægges den studerende ikke
- Aggregering er en svagere relation end komposition
- Implementeringen foretages vha. pointere – du kan se det i dokumentet "Aggregering.pdf"

Association

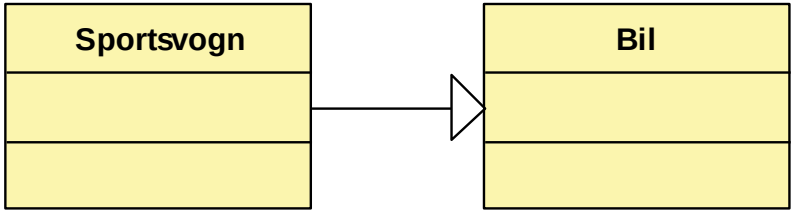
- En ”anvender/bruger/aflæser/osv.”-relation
- Eksempel:
  - En Sensor ***skriver til*** en Log
  - Et Sensor ***består ikke*** af en Log
  - En Log er ***ikke*** en ***del af*** en Sensor



- Sensoren ***ejer ikke*** logen – flere sensorer kan skrive til logen
- De har ***ikke*** samme levetid – hvis sensoren nedlægges, nedlægges logen ikke
- Association er en svagere relation end aggregering
- Vi kigger nærmere på denne relation om lidt

Arv

- En ”***er*** en/et”-relation
- Eksempel: En `Sportsvogn` ***er*** en `Bil`
- Lærer du om senere i dette kursus



Repetition af klasserelation komposition

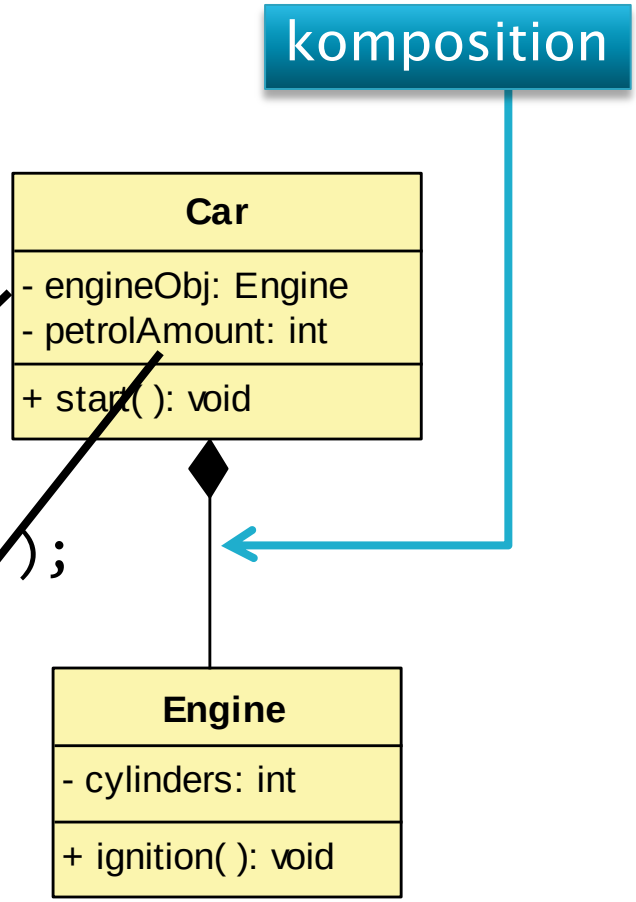
Komposition – 1

- Eksempler
  - En `Bil` ***har*** en `Motor`
    - Dvs. et Motor objekt er ***medlem*** af Bil klassen
  - En `Cirkel` ***har*** et `Punkt`
    - Dvs. et Punkt objekt er ***medlem*** af Cirkel klassen
  - Et `Kontrolpanel` ***har*** en `Knap`
    - Dvs. et Knap objekt er ***medlem*** af Kontrolpanel klassen
- Komposition implementeres som ”forventet”
  - Eksempel:
    - Klassen `Bil` har et `Motor` objekt som privat attribut

Komposition – 2

- Eksempel:

```
class Car
{
public:
        Car( int pa=0, int cyl=0 );
        void start();
private:
        Engine engineObj_;
        int petrolAmount_;
};
```

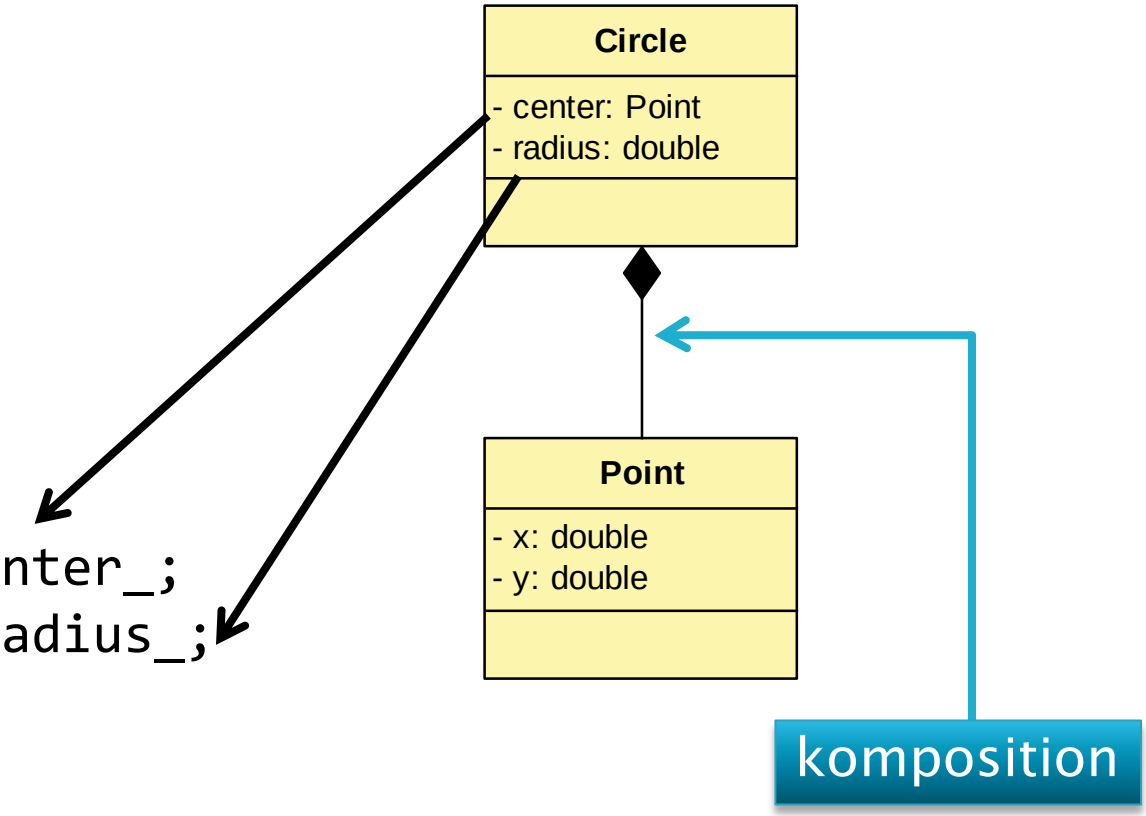


Bemærk, at motor-objektet er en privat del af bil-objektet – det kan derfor ***ikke*** bruges af andre objekter. Hvert bil-objekt har ***sit eget*** motor-objekt.

Komposition – 3

- Eksempel:

```
class Circle
{
public:
        .
        .
        .
private:
        Point center_;
        double radius_;
};
```



Bemærk, at Point objektet oprettes når der oprettes et Circle objekt og det nedlægges når Circle objektet nedlægges.

Repetition af pointere

Pointere – 1

- Datatyper:
  - `int` – 16/32 bit – kan indeholde hele tal
  - `float` – 32 bit – kan indeholde decimaltal
  - `double` – 64 bit – kan indeholde decimaltal
  - `char` – 8 bit – kan indeholde karakterer (ASCII)
  - * (pointer) – 32 bit – kan indeh. lageradr. (hextal)

Pointere – 2

- MEN……lageradressen er kun ***1. byte*** af det den refererer til.
- DERFOR…… SKAL en pointer have en ***type*** (for at ”den ved” hvor meget den ***ialt*** refererer til).
- Dvs.

```
char myChar = ‘a’;

char *myCharPtr = &myChar;
```

En lille repetitions opgave

- Hvad udskrives i følgende linier?
- Diskutér 3 minutter med din sidemand (m/k).

```
6     char myChar = ‘a’;
7     char *myCharPtr = &myChar;
8
9     cout << myChar << endl;
10
11    cout << &myChar << endl;
12
13    cout << myCharPtr << endl;
14
15    cout << *myCharPtr << endl;
```

Pointere – 3

- ALTSÅ:
  - i en ***erklæring*** betyder * at det er en ***pointer***

```
char *myCharPtr = &myChar;
```

  - andre steder betyder * *"det som pointeren peger på"*

```
cout << *myCharPtr << endl;
char c = *myCharPtr;
```

Pointere – 4

- Hvad bruges pointere til?
  - Til at gennemløbe arrays (1.semester)
  - Til at ”sende” information til en funktion – f.eks. information om et array (for at undgå kopiering) – det kaldes ”call-by-reference” (1.semester)
  - Til at implementere relationen aggregering
  - Til at implementere relationen association
  - Til dynamisk lagerallokering (kommer senere i kurset)

Pointere – 5

- Når pointere bruges til ”call-by-reference”, er det vigtigt at du anvender const, de steder hvor det er relevant – dvs. når der kun må ***læses*** via pointeren
- Nedenstående pointer referere til en `const int`. Det, den refererer til, kan derfor ***læses*** men ***ikke ændres*** via pointeren:

```
const int *myPtr = &x;

*myPtr = 4;      // ERROR
```

- Dette er **vigtigt** og bruges ***meget ofte*** i metoders parameterliste

Funktionskald – ”call-by-value”

```
int triple( int );                  // ”call-by-value”-funktion

int main()
{
    int x = 4, y;

    y = triple( x );                // der sendes en KOPI af x
                                    // y ændres
}

int triple( int a )                 // a = 4
{
    a *= 3;                         // x ændres IKKE
    return a;                       // 12 returneres
}
```

Funktionskald – ”call-by-reference”

```
void triple( int * );          // ”call-by-reference”-funktion

int main()
{
    int x = 4;

    triple( &x );              // der sendes ADRESSEN af x
                               // x er ændret !!!
}

void triple( int *aPtr )       // aPtr = &x
{
    (*aPtr) *= 3;              // x ændres !!!
}
```

Dagens emne

Klasserelationen association

Association – 1

- En ”anvender/bruger/aflæser/osv.”-relation
- Eksempel:
  - En Sensor ***skriver til*** en Log
  - Et Sensor ***består ikke*** af en Log
  - En Log er ***ikke*** en ***del af*** en Sensor
- Sensoren ***ejer ikke*** logen – flere sensorer kan skrive til logen
- De har ***ikke*** samme levetid – hvis sensoren nedlægges, nedlægges logen ikke
- Association er en svagere relation end aggregering
- Vi kigger nærmere på denne relation om lidt

Association – 2

- Flere eksempler:
  - En `Sensor` ***skriver*** til en `Log` (andre kan skrive til samme log)
  - En `Monitor` ***anvender*** et `Videokamera` (billedet fra kameraet kan vises på flere monitorer)
  - En `Tid` ***vises*** på et `Display` (der kan vises andre info på displayet – f.eks. `Dato`)
- Altså……der er ***ingen*** ejerskab ved association – og objekterne har ***ikke*** samme levetid !!!
- Det er en svag relation

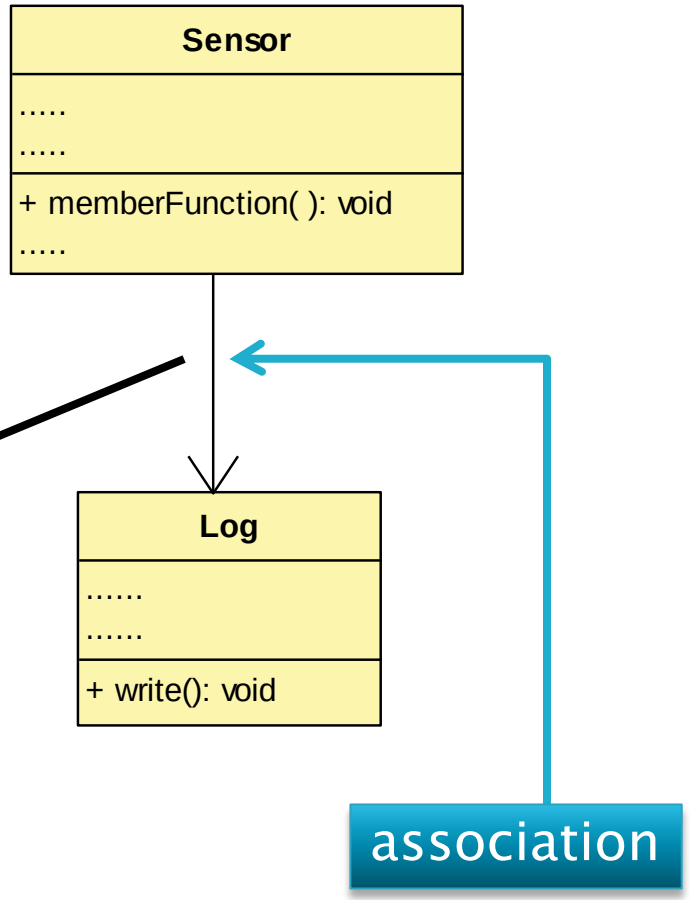
Association – 3

- I modsætning til komposition er Log objektet derfor ***ikke*** medlem af Sensor klassen
- Log objektet og relationen til Sensor objektet oprettes derfor ***ikke*** automatisk
- Derimod er en Log ***pointer*** medlem af Sensor klassen
- ***Begge*** objekter oprettes eksplicit (af dig) i koden (kunne være i main()) og relationen skabes via en parameter til constructoren som bruges til at initialisere member-pointeren

Association – 4

- Eksempel:

```
class Sensor
{
public:
        Sensor( Log * );
        .
        .
private:
        Log *myLogPtr_;
        .
};
```

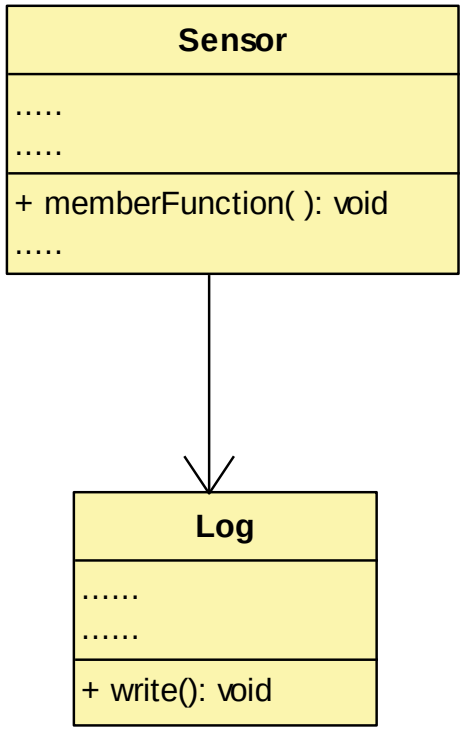


Association – 5

- Eksempel – fortsat:

```
Sensor::Sensor( Log *LPtr )
{
        myLogPtr_ = LPtr;
}


void Sensor::memberFunction()
{
        myLogPtr_->write();
}
```



Kald af metode via POINTER

Skaber forbindelsen Til Log-objektet

Association – 6

- NB!
  - Dot-operatoren kan ***kun*** bruges sammen med et ***objekt***
  - Da `myLogPtr` ***ikke*** er et objekt men en ***pointer*** til et objekt kan du ***ikke*** skrive

```
myLogPtr_.write()  // Error!!!
```

  - Derimod er ”det som pointeren peger på” et objekt. Du er derfor nødt til at skrive

```
(*myLogPtr_).write()
```

  - ***Men***...der findes en speciel operator der ***erstatter*** dette:

```
myLogPtr_->write()
```

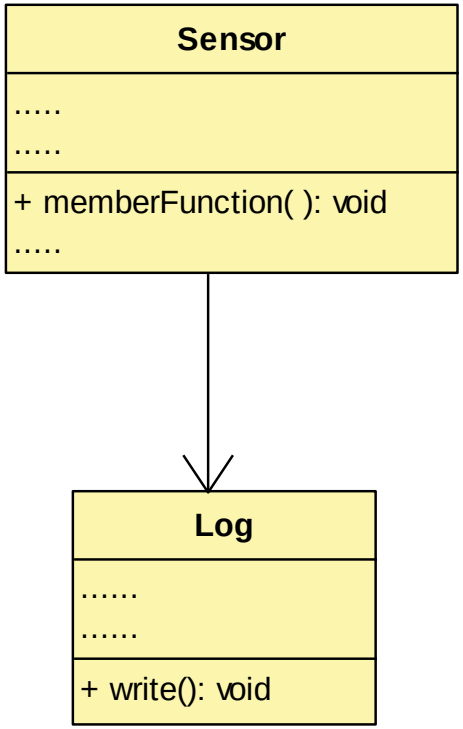
Association – 7

- Eksempel – fortsat:

```
int main()
{
        Log myLog;
        Sensor mySensor( &myLog );

        .
        .
        .

        return 0;
}
```



Adressen på Log-objektet sendes til constructoren

Eksempel

Association.pdf

Eksempel

SDS klassediagram.pdf