Objektorienteret programmering

Introduktion

Plan

- Generelt om programmeludvikling
- Objekter og klasser (begreber)
- Objektorienteret programmeludvikling
- Programmering i Java

Aforisme

Ordene er håndtag på tingene.
Hvis vi ikke passer på,
holder de os virkeligheden fra livet.

Piet Hein

Programmeludvikling

Megen udvikling baseres på trial-and-error.

Problemerne træder tydeligst frem ved udvikling af store systemer.

Der er behov for kodificering af viden om, hvad man skal gøre, og hvad man ikke skal gøre (designviden).

Ønskværdige egenskaber ved programmel

- Brugbart
- Rettidigt
- Korrekt
- Robust
- Brugervenligt
- Effektivt
- Let at vedligeholde
- Genbrugeligt

Normalt kan alle egenskaber ikke opfyldes samtidigt.

Objektorienteret programmering

Fokus på

- Vedligeholdelse
- Genbrug

Vedligeholdelse forenkles ved

- Enkelhed
- Fleksibilitet
- Læsbarhed

To muligheder for fokusering

(1) **Algoritmer**
Fokusering på "control flow"
(procedure-orienteret tilgang)

(2) **Datastrukturer**
Fokusering på "data flow"

Objektorienteret programmering er en balance imellem disse to yderpunkter.

Om sprogs betydning

The limits of my language
mean the limits of my world.

Ludwig Wittgenstein

Objektorienterede sprog

Simula	1967	
Smalltalk	1970	
Objective C	1982	
C++	1983	(Bjarne Stoustrup, Danmark)
Eiffel	1985	
Object Pascal	1986	
Beta	1990	
Java	1995	
C#	2000	(Anders Hejlsberg, Danmark)

og mange flere

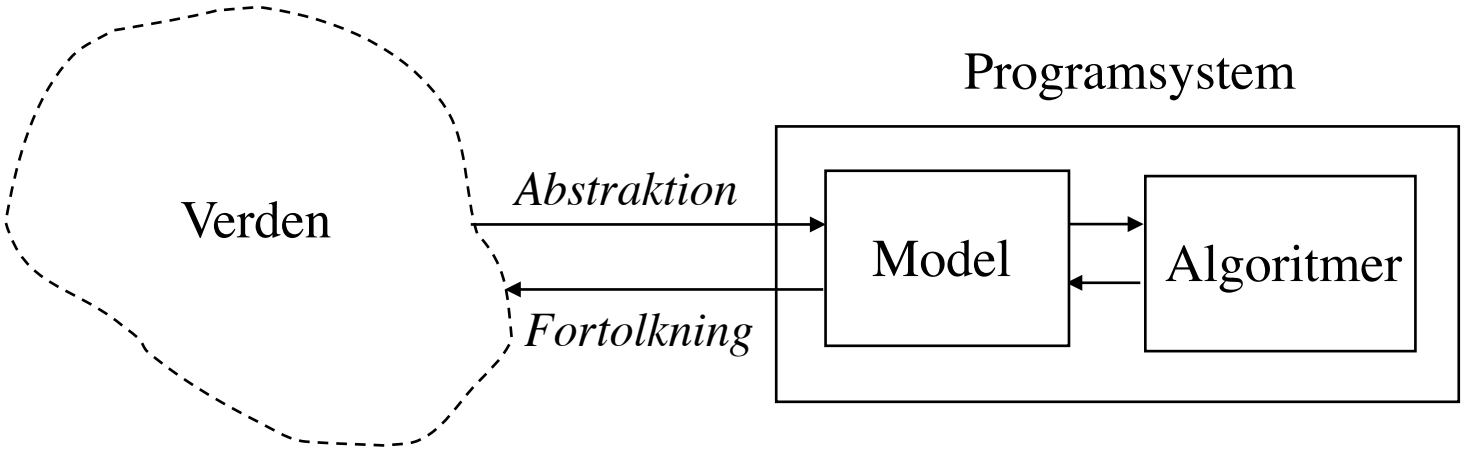
Citat

Alan J. Perlis

“You think you know when you learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program.”

-Alan J. Perlis

Programmeludvikling



Abstrahere: at se bort fra noget

Målet for programmeludvikling er konstruktion af programsystemer, der hjælper mennesker med problemløsning i den virkelige verden.

Grundlæggende begreber i objektorienteret programmering

Objekt:

Fortolkning i den virkelige verden:

Et *objekt* kan repræsentere alt, hvad der tydeligt kan identificeres.

Repræsentation i model:

Et *objekt* har en identitet, en tilstand og en adfærd.

Tilstand og adfærd

Tilstanden for et objekt defineres ved *felter* med tilhørende værdier.

Adfærden for et objekt defineres ved *metoder*, der kan aflæse eller ændre objektets tilstand.

Klasser

Klasse:

Fortolkning i den virkelige verden:

Et *klasse* repræsenterer en mængde af objekter med fælles karakteristika. Objekterne kaldes for *instanser* af klassen.

Repræsentation i model:

En *klasse* beskriver den struktur af tilstande og adfærd, der er fælles for alle klassens instanser.

Eksempel på en klasse

```
class Point {     // Klassenavn
    int x, y;     // Felter

    void move(int dx, int dy) {    // Metode
        x += dx;
        y += dy;
    }
}
```

En klasse er en **skabelon** til brug for skabelse af objekter.

Syntaks for felter og metoder

Felt:

[*Visibility*] *Type Identifier* [**=** *InitialValue*]

Metode:

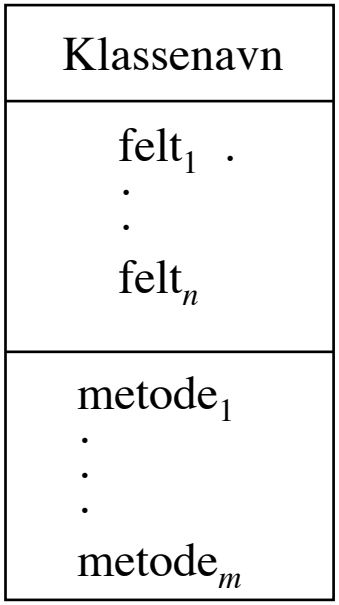
[*Visibility*] *Type Identifier*([*ParameterList*])

Visibility ::= **`public`** | **`private`** | **`protected`**

Grafisk notation

UML (Unified Modelling Language)

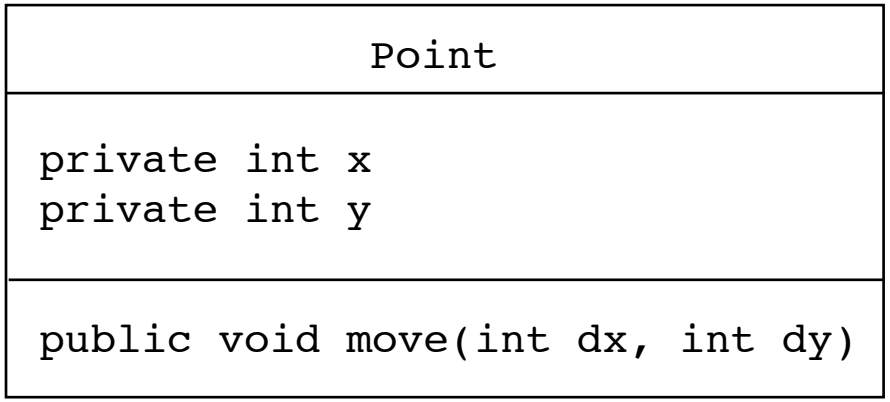
Klasse:

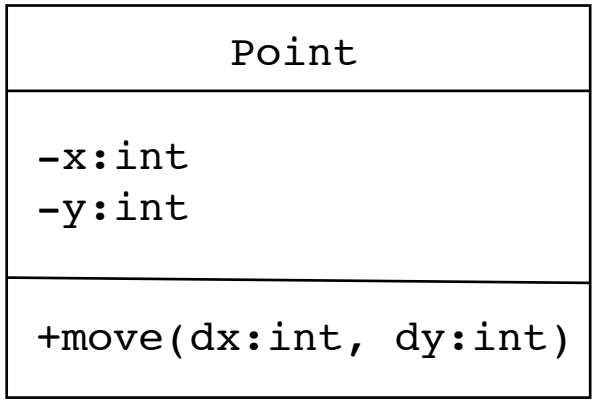


De to nederste dele kan udelades.

Eksempel

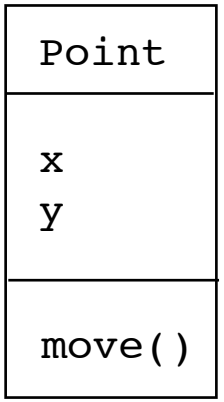
```
class Point
```





UML-syntaks

Forkortede former



UML-notation for objekter

Objektnavn : Klassenavn
$felt_1$ = $værdi_1$ . . . $felt_n$ = $værdi_n$

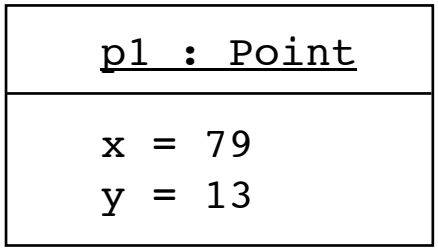
Bemærk understregningen

Den nederste del kan udelades.

Objektnavnet eller klassenavnet kan udelades.

Eksempel

Et `Point`-objekt




```
Point p1 = new Point();
p1.x = 79;
p1.y = 13;
```

Et metode kan kalde et objekts metoder.

Eksempel på kald: `p1.move(3, 4)`

3 nyttige principper ved programmeludvikling

1. **Modularisering**
2. **Abstraktion**
3. **Indkapsling**

1. Modularisering

Modularitet er nøglen til god programmering.

Opdel programmet i små moduler, som interagerer med hinanden igennem snævre, veldefinerede grænseflader.

Fordele:

- En programdel kan forstås, uden at hele programmet forstås.
- En fejl i en programdel kan rettes, uden at resten af koden bliver inddraget.

Modularisering

Princip:

> Et komplekst system dekomponeres til en mængde af samhørende, men løst koblede komponenter, kaldet *moduler*

Modulerne er typisk organiseret i et hierarki (d.v.s. et modul kan indeholde andre moduler).

Ved objektorienteret programmeludvikling udgøres modulerne af klasser og pakker (samhørende klasser).

2. Abstraktion

At *abstrahere*: at se bort fra noget.

Abstraktion hjælper til med at bestemme en hensigtsmæssig samling af komponenter.

Vi kan betragte forskellige ting som ens, selv om de er forskellige.

Abstraktion

Princip:

> Modulets adfærd karakteriseres ved hjælp af en kortfattet og præcis beskrivelse

Beskrivelsen kaldes modulets *kontraktlige grænseflade*.

Kontrakter

Et modul yder tjenester til (servicerer) sine brugere (klienter). Et modul skal overholde sine kontraktlige forpligtigelser.

En kontrakt beskriver *hvad*, men **ikke** *hvordan*.
Klienter behøver kun at kende kontrakten for at bruge modulet.

Udformning af kontrakter for klasser

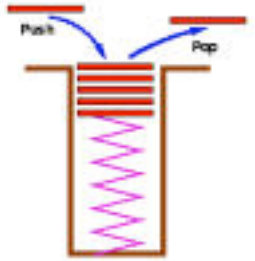
En kontrakt specificerer for hver af klassens offentlige metoder metodens ***pre**betingelse* og metodens ***post**betingelse*.

Prebetingelsen fastlægger *forudsætningerne* for at kalde metoden.

Postbetingelsen fastlægger *virkningen* af et kald.

En betingelse, som gælder såvel før som efter kald af enhver af klassens offentlige metoder, kaldes for en *klasseinvariant*.

Eksempel på specifikation




```
/**
 * @invariant 0 <= size() <= capacity
 */
public class Stack {
    /** Place an object at the top of the stack
     * @pre !isFull()
     * @post size() = size()@pre + 1
     */
     public void push(Object obj) { ... }

    /** Remove and return the top object of the stack
     * @pre !isEmpty()
     * @post size() = size()@pre - 1
     */
     public Object pop() { ... }
}
```

Nytten af kontrakter

- **Design**: Et godt værktøj til programspecifikation
- **Implementering**: Angiver retningslinjer for programmering
- **Dokumentation**: Udgør en væsentlig del af dokumentationen
- **Afprøvning**: Angiver retningslinjer for ekstern afprøvning

3. Indkapsling

Princip:

> Implementationen af et modul bør adskilles fra dets kontraktlige grænseflade og være skjult for modulets klienter

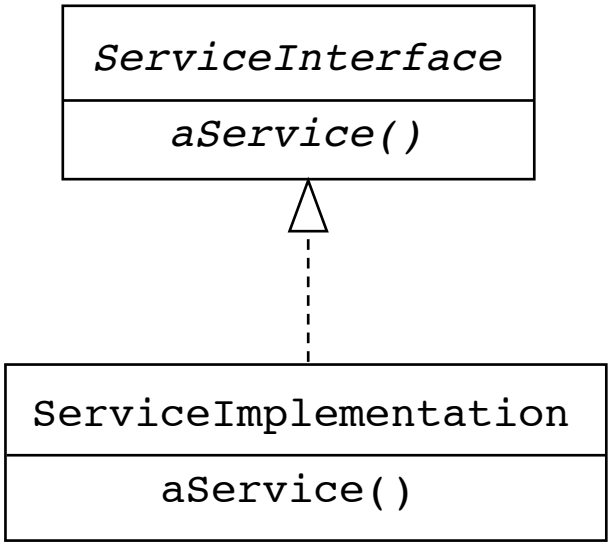
Jo mindre klienten ved om implementationen, desto løsere bliver koblingen imellem modulets og dets klienter.

Implementationen kan skiftes ud, uden at klienterne bliver berørt (så længe grænsefladen ikke ændres).

Grænseflader

En grænseflade uden implementation kaldes en *abstrakt datatype* (i Java-terminologi: et *interface*).

Grafisk notation:



Bemærk at grænsefladenavnet er skrevet i kursiv, og at linjen på pilen er stiplet

Eksempel

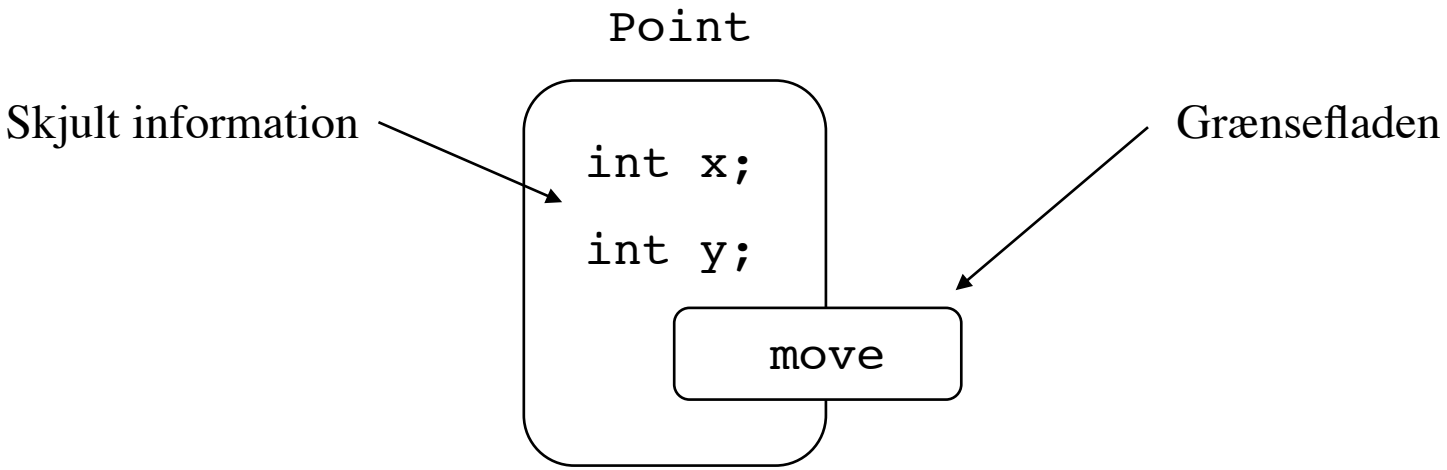
```
interface Movable {
   void move(int dx, int dy);
}
```

```
class Point implements Movable {
    int x, y;

    void move(int dx, dy) {
        x += dx;
        y += dy;
    }
}
```

Indkapsling

En klasse indeholder en række egenskaber.

Nogle egenskaber er skjulte (indkapslede). De øvrige udgør klassens grænseflade.

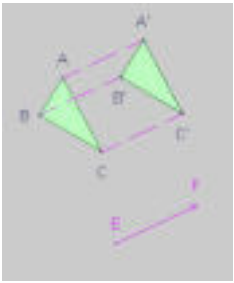


Polymorfi

En kontraktlig grænseflade kan overholdes af mange mulige implementeringer.

Muligheden for dynamisk at udskifte moduler, uden at klienterne berøres, kaldes for *polymorfi* (fra græsk: mange former).

Eksempel på polymorfi



```
class Line implements Movable {
    Point p1, p2;

    void move(int dx, dy) {
        p1.move(dx, dy);
        p2.move(dx, dy);
    }
}
```

```
Movable m;

...

m.move(3, 4);
```

`m` er polymorf

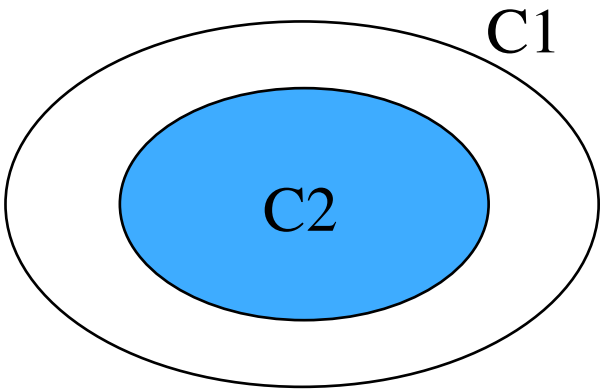
Virker, hvad enten `m` aktuelt er et `Point`-objekt eller et `Line`-objekt.

Relationer imellem klasser

- **Nedarvning**
- **Associering**
  - **Aggregering**
    - **Komposition**

Nedarvning

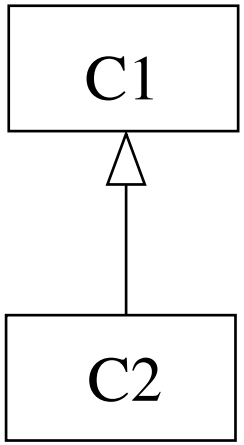
En klasse C2 siges at *nedarve* fra en anden klasse C1, hvis enhver instans af C2 også er en instans af C1.



C2 siges at være *underklasse* af C1.

C1 siges at være *overklasse* for C2.

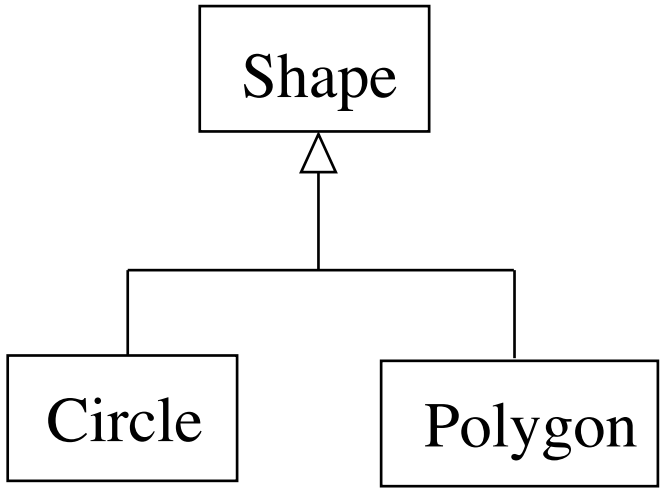
Notation for nedarvning




```
class C2 extends C1 {
    ...
}
```

C2 er en *udvidelse* af C1: alt hvad der gælder for C1, gælder også for C2.

Eksempel




```
class Shape {}

class Circle extends Shape {
    Point center;
    double radius;
}

class Polygon extends Shape {
    Point[] points;
}
```

Nedarvning repræsenterer en *er*-relation (engelsk: *is-a* relation).

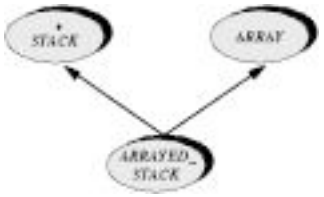
Tolkning af nedarvning

En underklasse udgør en *specialisering* af sin overklasse.

En overklasse er en *generalisering* af sine underklasser.

En underklasse udgør en *udvidelse* af sin overklasse.
En overklasses felter og metoder deles af dens underklasser (metoderne genbruges af underklasserne).

Multipel nedarvning



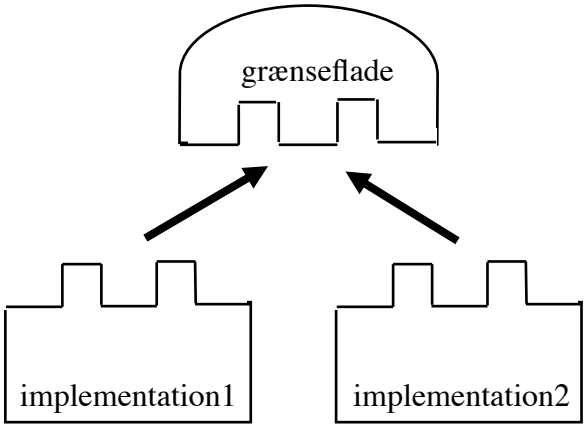
En klasse kan i princippet nedarve direkte fra mange overklasser. Dette kaldes for *multipel nedarvning*.

Java understøtter **ikke** nedarvning i sin fulde generalitet.

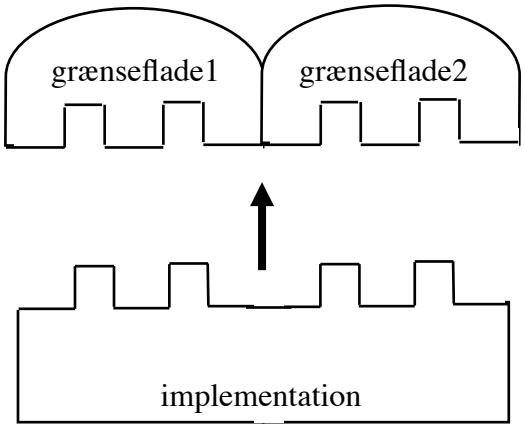
I Java kan en klasse højst nedarve direkte fra én anden klasse.

En begrænset form for multipel nedarvning er dog mulig, idet en klasse gerne må implementere mere end ét interface.

Relation imellem grænseflade og implementering

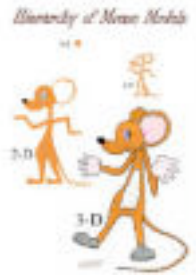


Multiple implementeringer



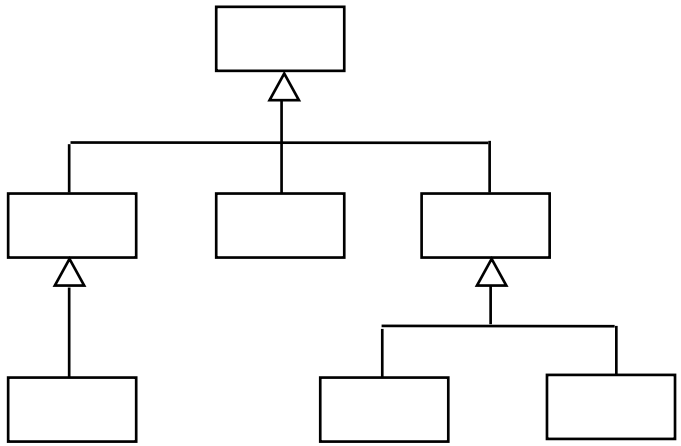
Multiple grænseflader

Abstraktionsniveauer



Princip:

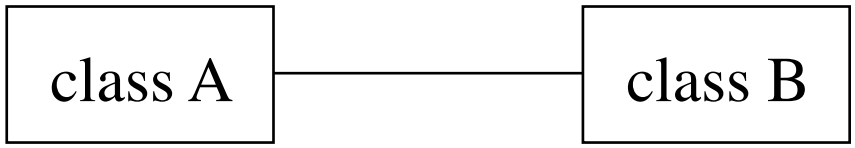
Abstraktioner kan organiseres i niveauer (lag)
Jo højere niveau, desto mere generel er abstraktionen
Jo lavere niveau, desto mere specifik er abstraktionen



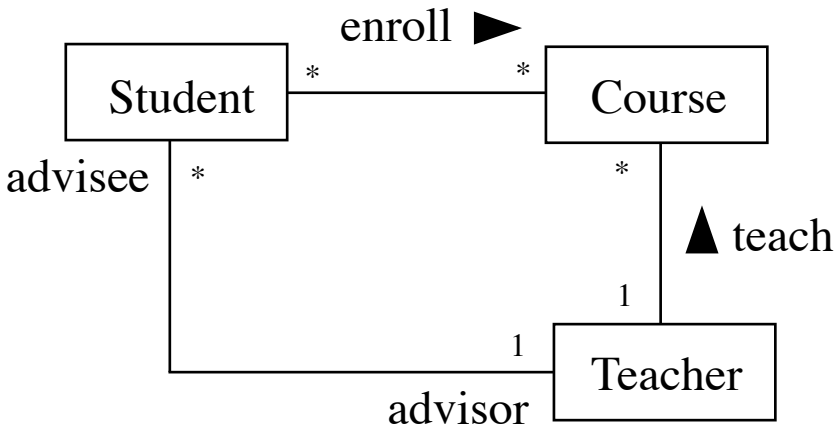
Associering

En associering repræsenterer en relation imellem to klasser.

Angives i UML med en kant:



Associeringer kan navngives, og deres retning kan angives.



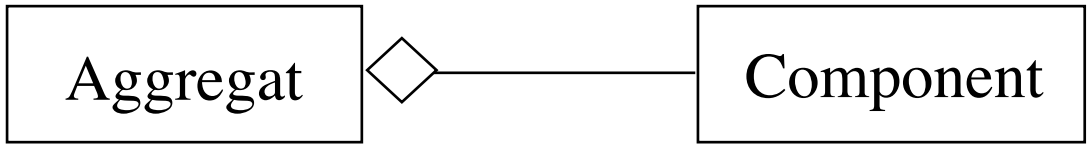
En kants ender kan forsynes med heltalsintervaller:

$l...u$	$l, l+1, ..., u$
i	i
*	0, 1, 2, ...

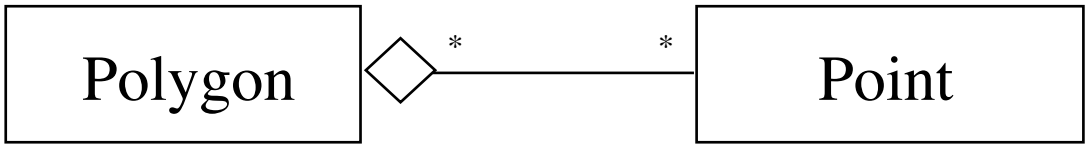
Aggregering

Aggregering er en speciel form for associering og betegner en *har*-relation (engelsk: *has-a* relation).

Grafisk notation:



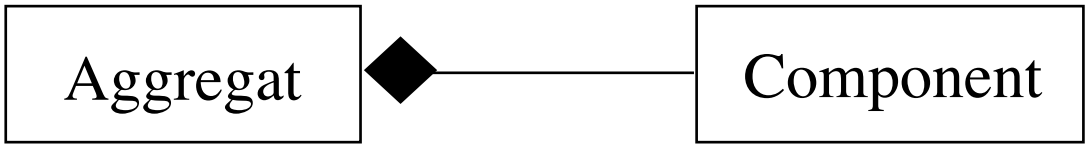
Der kan knyttes intervalangivelser på en kants endepunkter.
Eksempel:



Komposition

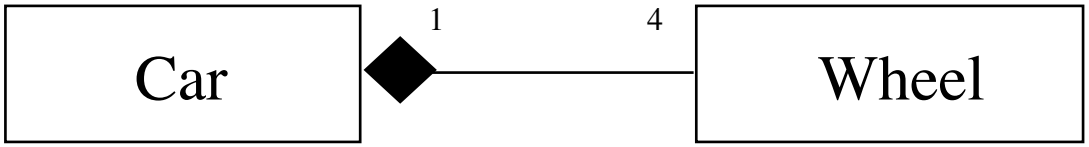
Komposition er en stærkere form for aggregering, der benyttes, når der er tale om eksklusivt ejerskab.

Grafisk notation:

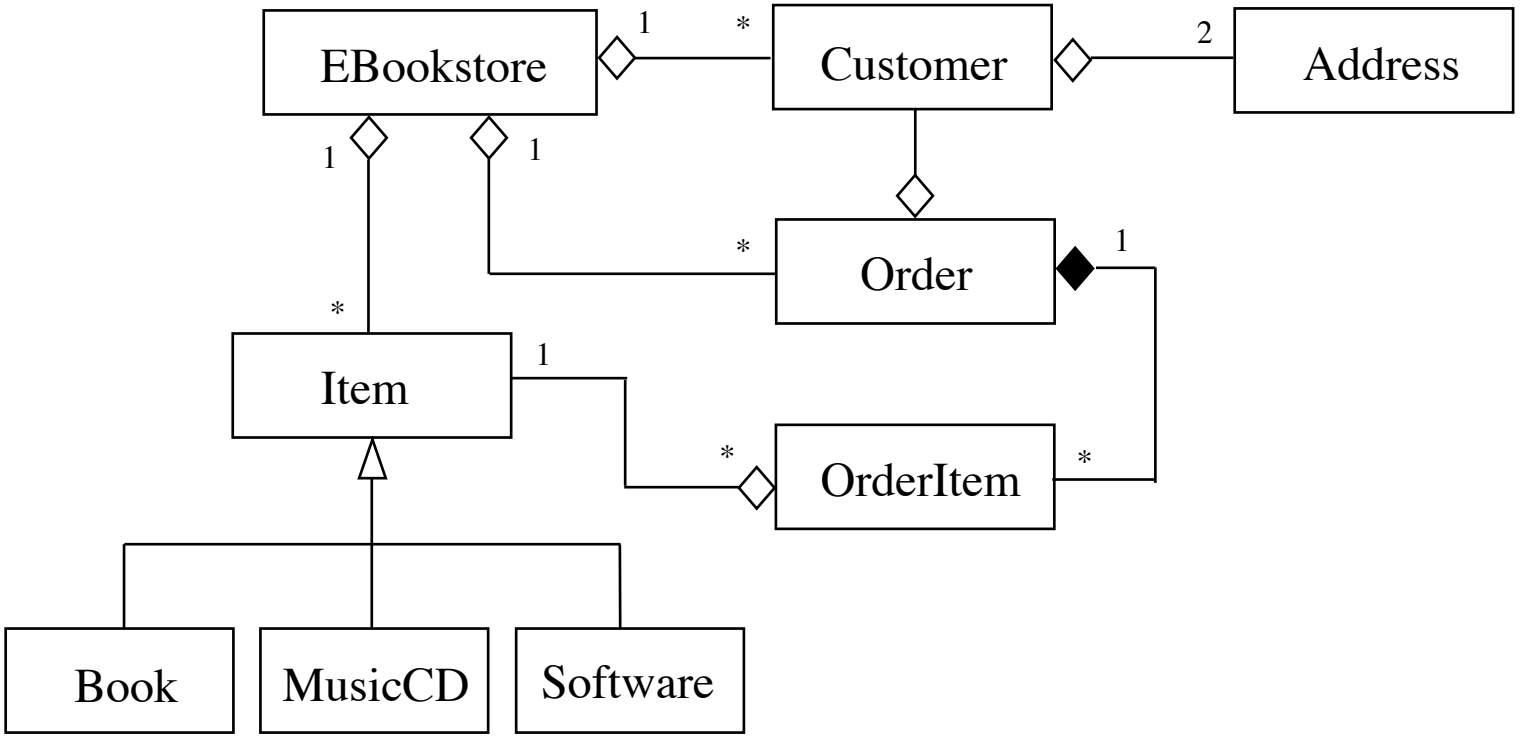


Der kan knyttes intervalangivelser på en kants endepunkter.

Eksempel:



UML-diagram for en E-boghandel



Java-klasser for online-boghandelen

```
class Item {
    String title;
    String publisher;
    int yearPublished;
    int ISBN;
    double price;
}

class Book extends Item {
    String  author;
    int edition;
    int volume;
}

class Software extends Item {
    int version;
}

class MusicCD extends Item {
    String artist;
    int volume;
}
```

fortsættes

```
class OrderItem {
    Item item;
    int quantity;
}

class Order {
    Customer customer;
    Collection<OrderItem> orderItems;
    double salesTax;
    double shippingFee;
    double total;
    int methodOfPayment;
}
```

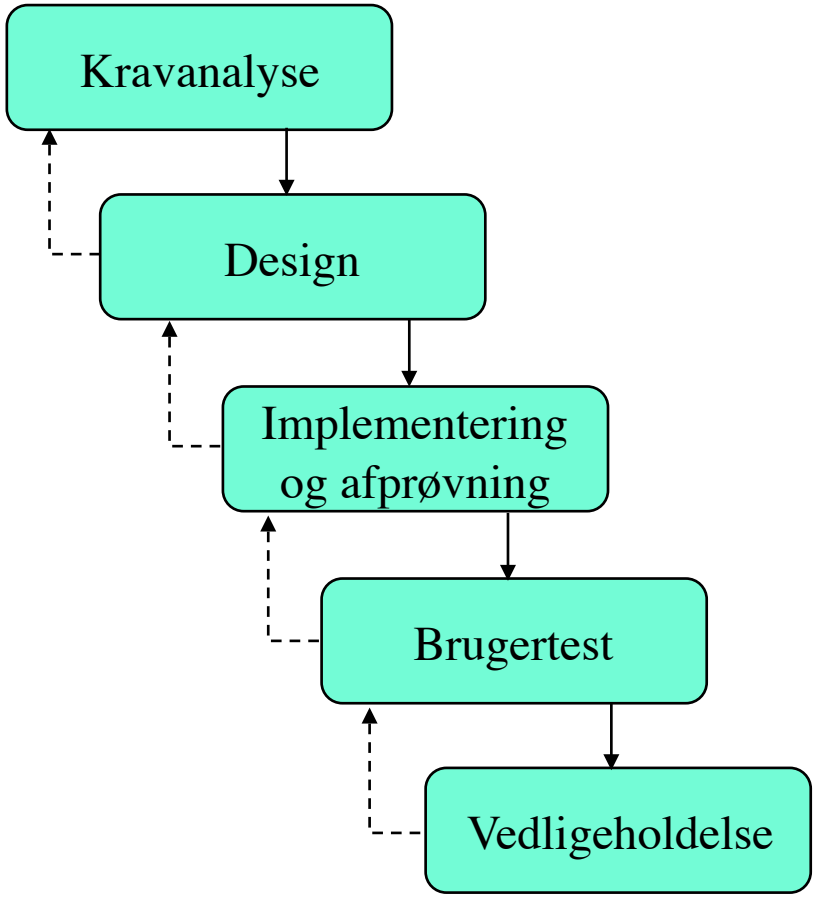
fortsættes

```
class Customer {
    String name;
    int ID;
    String password;
    Address shippingAdrees;
    Address billingAdress;
}

class Address {
    String street;
    String city;
    String state;
    String country;
    String postalCode;
}

class EBookstore {
    Collection<Customer> customers;
    Collection<OrderItem> orders;
    Collection<Item> items;
}
```

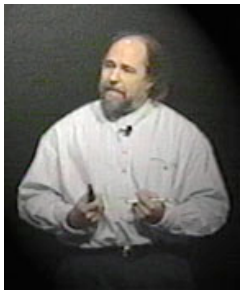
Traditionel systemudvikling

Vandfaldsmodellen:

faseopdelt



Objektorienteret systemudvikling
(Booch)

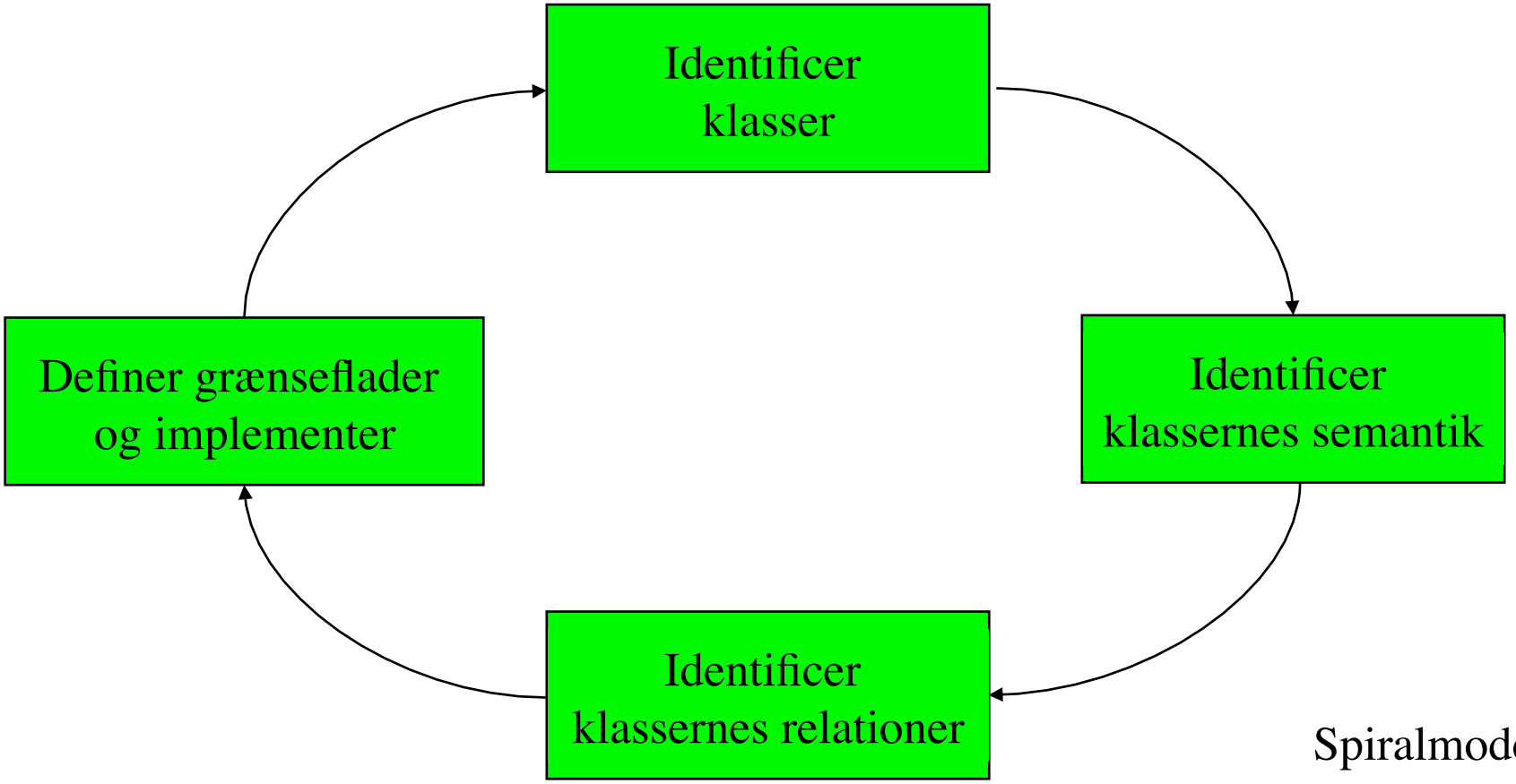


Udviklingsprocessen er *iterativ*.
Den består af successive iterationer, der som mål har, at

- identificere klasserne
- identificere klassernes *semantik* (deres felter og metoder)
- identificere klassernes indbyrdes *relationer*
- definere *grænseflader*
- *implementere* klasserne

Systemet vokser i relativt små afgrænsede skridt.

Mikroprocessen



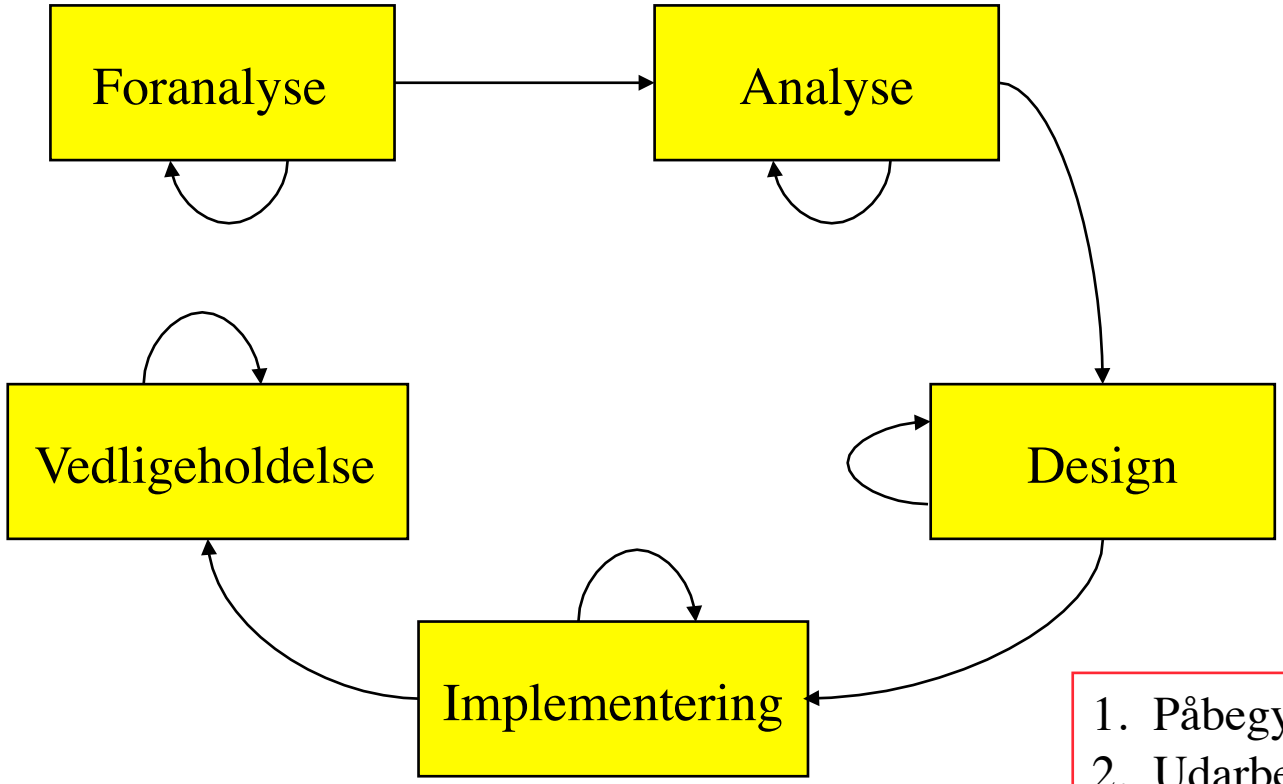
Spiralmodel

Makroproces

Førnævnte tilgang kaldes for **mikroproces**.
For at *styre* mikroprocessen benyttes følgende faser:

- **Begrebsliggørelse (foranalyse)**
  Forståelse af projektet, fastsættelse af rammer
- **Analyse**
  Udvikling af en model af systemets ønskede opførsel
- **Design**
  Skabelse af en arkitektur
  (objekter/klasser og deres relationer)
- **Implementering**
  Kodning i et objektorienteret programmeringssprog
- **Vedligeholdelse**
  Fejlretning og videreudvikling

Makroprocessen



1. Påbegyndelse
2. Udarbejdelse
3. Konstruktion
4. Overdragelse

Videreudviklet i **R**ational **U**nified **P**rocess

Retningslinjer for udviklingsprocessen

(1) Begrebsliggørelse

Nedskriv de basale krav til programmellets funktionalitet.

(2) Modellering af krav

Foretag *scenarie-analyse*:

Opskriv mulige brugerhandlinger og angiv systemets ønskede reaktioner. Start med scenarier, der omfatter handlinger, der ikke er fejlbehæftede.

(3) Identifikation af klasser

Foretag *navne-udsagnsord-analyse*:

Understreg alle navneord og udsagnsord i kravspecifikationen. Navneordene er kandidater for klasser. Udsagnsordene er kandidater for metoder.

(4) Identifikation af klassernes semantik

Identificer klassernes felter og metoder. Find fællestræk for klasserne og lav overklasser, der indeholde disse fællestræk.

(5) Identifikation af klassernes indbyrdes relationer

Klassediagram
Brugstilfældediagram
Tilstandsdiagram
Sekvensdiagram

Extreme programming (XP)

www.xprogramming.com

- Udviklingen foretages i iterationer

  Resultatet af hver iteration er et kørende program, der enten er en udvidelse eller en omstrukturering af programmet fra forrige iteration

- Skriv tests først

  Fremtvinger en klar definition af grænsefladen

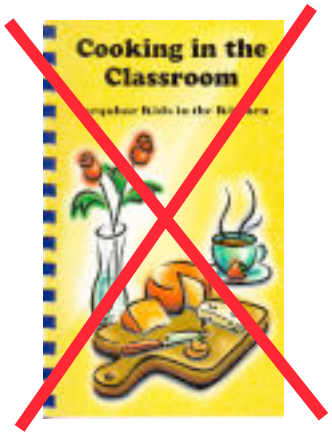
  Fanger fejl tidligt i forløbet

- Programmér i par

  Den ene koder, mens den anden tænker

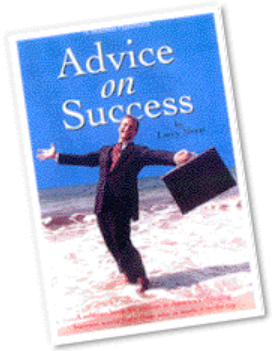
Det er OK med diagrammering. Men brug ikke for megen tid på det, og vær villig til at kassere diagrammerne.

Om udviklingsmetoder



Der er ingen kogebogsmetoder, der kan erstatte intelligens, erfaring og god smag ved design og programmering.

Nogle designråd



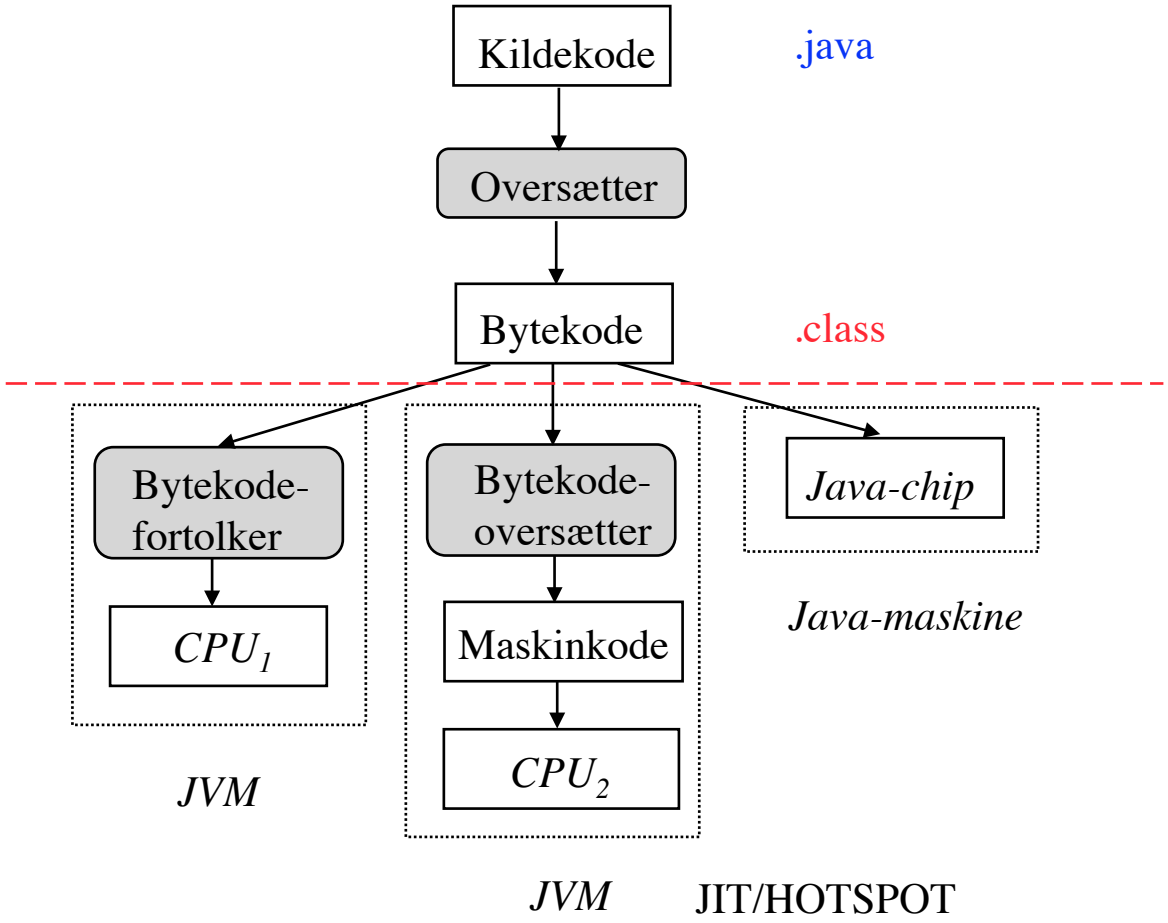
1. Gør hverken designet for specifikt eller for generelt
2. Undgå unaturlige klasser
3. Respekter de naturlige abstraktionsniveauer
4. En klasse bør ikke have for mange forpligtelser (1-3)
5. En klasse bør ikke have urelaterede forpligtelser

Javas egenskaber

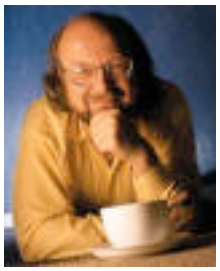
- Objektorienteret
- Distribueret
- Sikkert
- Platformsuafhængigt

Udførelse af Java-programmer

Et kompromis imellem oversættelse og fortolkning



Bytekode



James Gosling

Opcode (1 byte)
$operand_1$
$operand_2$
...

JVM's instruktionscyklus:

```
do {
    hent opcode-byten for den aktuelle instruktion;
    hent operanderne;
    udfør instruktionen;
} while (!done);
```

Applikationer og appletter

Applikation: et normalt program med adgang til alle systemressourcer.

Applet: et program indlejret i en Web-side med begrænset adgang til systemressourcer.

Sikkerhed

- Lagerbeskyttelse
- Kodeverifikation
- Ressourcebeskyttelse

  En applet kan normalt **ikke**

  - læse og skrive filer på værtsmaskinen
  - kommunikere med andre maskiner end den, hvorfra appletten er hentet
  - starte andre programmer

En simpel applikation


```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello from Venus");
    }
}
```

Hello From Venus!

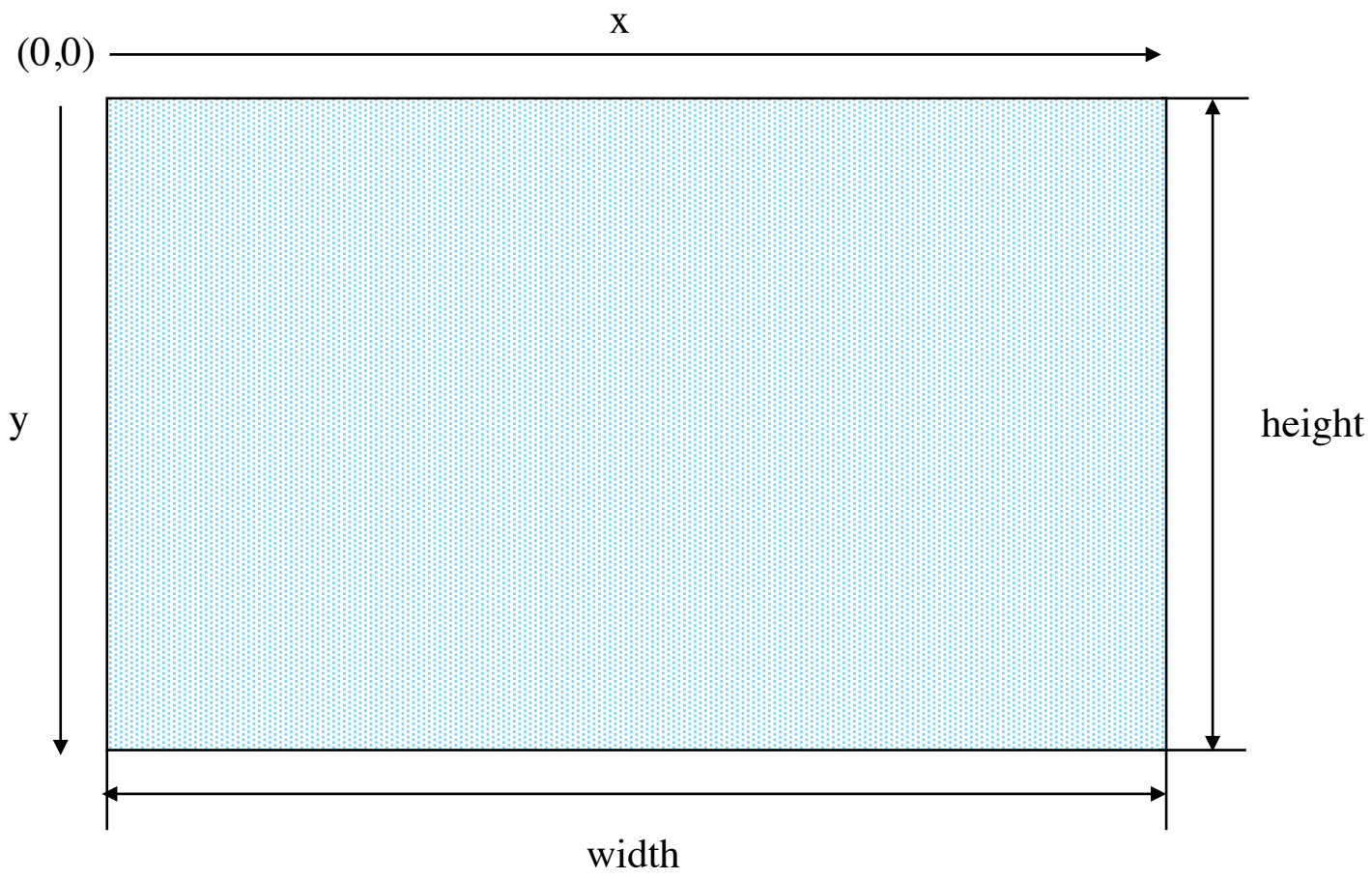
---

the source.

En simpel applet

```
import java.awt.*;
import java.applet.Applet;

public class HelloFromVenus extends Applet {
    public void paint(Graphics g) {
        Dimension d = getSize();
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, d.width, d.height);
        g.setFont(new Font("Helvetica", Font.BOLD, 24));
        g.setColor(new Color(255, 215, 0)); // gold color
        g.drawString("Hello From Venus!", 40, 25);
        g.drawImage(getImage(getCodeBase(), "Venus.gif"),
                    20, 60, this);
    }
}
```

Det grafiske koordinatsystem



Konkret eksempel

(0,0) (300,0)

(0,350) (300,350)

HTML-kode

```
<html>
  <head>
    <title>
      HelloFromVenus applet
    </title>
  </head>

  <body bgcolor=black text=white>
    <center>
      <applet code="HelloFromVenus.class"
              width=300 height=350>
      </applet>
    </center>
    <hr>
    <a href="HelloFromVenus.java">the source.</a>
  </body>
</html>
```

Brug af Java-arkiv

Hvis et program består af flere filer - klassefiler, såvel som billede- og lydfiler - kan disse med fordel komprimeres og samles i et Java-arkiv.

```
jar cf Hello.jar HelloFromVenus.class Venus.gif
```

Specificering af anvendelse i HTML:

```
<applet code="HelloFromVenus.class"
        archive="Hello.jar"
        width=300 height=350>
</applet>
```

Ugeseddel 1

31. august - 7. september

- Læs kapitel 4 i lærebogen (side 75 - 157)
- Løs opgaverne 3.1 og 3.2