02100 Indledende Programmering og Softwareteknologi

Uge 6 eftermiddagen om bogens kapitel 8: Klasser m.m.

Målet med formiddagens lektion var at blive i stand til:

- **At kunne håndtere en-dimensionale arrays af primitive typer og objekt-typer.**
- **At kunne håndtere fler-dimensionale arrays.**
- **At kunne skrive statiske metoder som tager arrays som parametre og/eller giver arrays som returværdier.**
- **At kunne operere med objekt-værdien null.**

Ugens hovedemne: Arrays & Klasser
02100 Indledende Programmering og Softwareteknologi

Pensum for uge 6: Kapitel 7 & 8.

Målet med eftermiddagens lektion er at blive i stand til:

- At kunne redegøre for begreberne klasse og objekt, samt relationerne mellem dem.
- At kunne forklare begrebet objekt-orienteret programmering.
- At kunne operere med felter, instans-metoder, konstruktører og indkapsling.
- At kunne designe og implementere simple klasser i Java.
- At kunne forklare begreberne kohæsion og kobling i relation til objekt-orienteret programmering.

Typer af klasser

Der findes to typer af klasser i Java:

1. Selvstændige moduler der kan køres som programmer.
   Disse klasser har altid en main-metode.
2. Skabeloner for en type af objekter.

Indtil videre har alle de klasser som I er blevet bedt om at lave i kurset været af den første type. Vi har til gengæld set på klasser af den anden type i Java's bibliotek, f.eks. klasserne `Random` og `Scanner`. Disse 2 bibliotheksklasser indeholder skabeloner til at oprette hhv. `Random`- og `Scanner`-objekter.

Når en klasse definerer en type af objekter kalder vi ofte disse objekter for **instanser** (eng.: **instances**) af klassen. Klassen er den skabelon som fortæller hvordan objekterne (instanserne) ser ud.

Klasserne `Math`, `Character` og `Arrays` er specielle. De indeholder først og fremmest en række statiske metoder.

Objekter

Objekt (eng.: **object**): En enhed i et program som indeholder en tilstand (data) og opførsel (metoder).

Eksempel. Vi betragter punkter i planen.
Ethvert `Point`-objekt indeholder en x- og en y-koordinat, hvilket er objektets *tilstand*.
Desuden "indeholder" objektet forskellige metoder såsom `distance`, `setLocation` og `translate`. De repræsenterer objektets *opførsel*.

Objekt-orienteret programmering

Objekt-orienteret programmering (eng.: **Object-Oriented Programming** / **OOP**): Det at skrive programmer som primært fungerer igennem interaktion mellem objekter.

Java er et objekt-orienteret programmeringssprog, men indtil videre har vi ikke skrevet nogen objekt-orienterede programmer. Det skal vi igang med nu...

Pointen med objekt-orienteret programmering er at give endnu mere struktur til vores programmer end det vi kan opnå ved blot at splitte op i statiske metoder. Mere struktur gør programmerne lettere at overskue, vedligeholde og udvide, og det gør det lettere at genbruge kode i andre programmer.

Point-objekter

`Point`-klassen indeholder konstruktører til at oprette `Point`-objekter med:

```
Point <navn 1> = new Point(<x>, <y>);  // nyt pkt. i (x,y)
Point <navn 2> = new Point();  // nyt pkt. i (0,0)
```

Ethvert `Point`-objekt som bliver oprettet indeholder både en *tilstand* og en *opførsel*:

- *Tilstand*: `x`- og `y`-felterne i objektet.
- *Opførsel*: Metoder som `distance`, `setLocation` og `translate`.

Tilstand og felter

Et objekts tilstand er beskrevet igennem det der i Java kaldes **felter**.

Felt (eng.: **field**): En særlig variabel i et objekt som udgør en del af objektets tilstand.

Vi kan oprette vores egen simple `Point`-klasse med følgende kode:

```
public class Point {
    int x;
    int y;
}
```

Ovenstående kode deklarerer `x` og `y` som felter. Det betyder at ethvert `Point`-objekt så vil få en tilstand der består af en `x`- og en `y`-værdi. Bemærk at de to variable ovenfor er deklareret helt i begyndelsen af klassens definition i stedet for inde i en af klassens metoder. Det er sådan felter skelnes fra sædvanlige variable.

Anvendelse af felter

```
public class Point {
   int x;
   int y;
}
```

Antag vi lægger ovenstående kode i en fil `Point.java`. Så kan vi fra andre programmer oprette objekter af typen `Point`:

```
1  public class UsePoints {
2     public static void main(String[] args) {
3        Point p1 = new Point();  // opretter objekt p1
4        p1.x = 5;  // opdaterer tilstand af p1
5        p1.y = 7;  // opdaterer tilstand af p1
6        Point p2 = new Point();  // opretter objekt p2
7        p2.x = 2;  // opdaterer tilstand af p2
8        p2.y = 3;  // opdaterer tilstand af p2
9     }
10 }
```

`p1` og `p2` har hver deres kopi af felterne `x` og `y`, så `p1.x` og `p2.x` er uafhængige af hinanden.

Syntaks for deklaration af felter

Den generelle syntaks for deklaration af felter i en klasse er nøjagtig som deklaration af sædvanlige variable:

```
<type> <navn>;
```

Forskellen er blot at felterne til forskel fra almindelige variable skal deklareres direkte inde i klassens yderste tuborg-parantes (ikke inde i en metode).

Eksempel. En simpel klasse for studerende, hvor en studerendes tilstand er givet ved navn og studienummer:

```
public class Studerende {
   String navn;
   String studienummer;
}
```

Ethvert objekt af typen `Studerende` vil da have sit eget `navn` og sit eget `studienummer`.

Syntaks for reference til felter

Den generelle syntaks for at returnere værdien af et felt `<feltnavn>` i et objekt `<objektnavn>` er følgende:

```
<objektnavn>.<feltnavn>
```

Eksempler.

```
p1.x
p1.y
array.length
```

Den generelle syntaks for at tildele en værdi til et felt `<feltnavn>` i et objekt `<objektnavn>` er følgende:

```
<objektnavn>.<feltnavn> = <værdi>;
```

Eksempel.

```
p1.x = 5;
```

Eksempel med felter

```
1  public class UsePoints {
2    public static void main(String[] args) {
3      Point p1 = new Point();  // opretter objekt p1
4      p1.x += 5;  // opdaterer tilstand af p1
5      p1.y += 7;  // opdaterer tilstand af p1
6      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
7      p1.x += 3;  // opdaterer tilstand af p1
8      p1.y += 4;  // opdaterer tilstand af p1
9      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
10     p1.x += 2;  // opdaterer tilstand af p1
11     p1.y += 3;  // opdaterer tilstand af p1
12     System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
13   }
14 }
```

Ovenstående kode opretter et `Point`-objekt initialiseret til $(0, 0)$, og forskyder dernæst dets koordinater flere gange. Koden har en del redundans...

Eksempel med felter – fortsat

Antag at vi mange gange vil forskyde et `Point`-objekts koordinater med kode i følgende stil:

```
1 p1.x += 3;
2 p1.y += 4;
```

For at undgå redundans kan vi lave en statisk metode der foretager forskydningen:

```
public static void translate(Point p, int dx, int dy) {
   p.x += dx;
   p.y += dy;
}
```

Koden fra linje 1 og 2 ovenfor kan så forkortes til:

```
translate(p1,3,4);
```

Instans-metoder

Løsningen med at lave en statisk metode som ovenfor er ikke optimal. Hele ideen med objekter er at de selv skal indeholde både *tilstand* og *opførsel*. Det at forskyde et punkt er en del af punktets opførsel, og burde således være indeholdt i objektet.

Et objekts opførsel beskrives igennem *instans-metoder*.

Instans-metode (eng.: **instance method**): En metode som definerer en opførsel af et objekt. Metoden deklareres uden nøgleordet `static`, og metodens krop kan frit referere til objektets felter og andre instans-metoder fra objektet.

Syntaks for deklaration af instans-metoder:

```
public <returtype> <metodenavn>(<parametre>) {
   <liste af kommandoer>
}
```

Bemærk fraværet af nøgleordet `static`.

Mere om instans-metoder

En instans-metode til at forskyde et `Point`-objekt kan deklareres på følgende måde:

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

Bemærk at vi ikke længere har `Point`-objektet som parameter til metoden. Det er fordi vi opfatter instans-metoder som del af objekterne selv, så vi kan lade objektet være en *implicit parameter*.

Syntaksen for at benytte instans-metoder er følgende:

```
<objektnavn>.<metodenavn>(<parametre>)
```

Eksempel. Hvis `p1` er et `Point`-objekt kan vi forskyde det med koordinaterne $(2, 4)$ med følgende metode-kald:

```
p1.translate(2,4);
```

Statiske metoder vs. instans-metoder

Den *statiske metode* til forskydning af punkter blev deklareret på følgende måde:

```
public static void translate(Point p, int dx, int dy) {
   p.x += dx;
   p.y += dy;
}
```

Den tilsvarende *instans-metode* blev deklareret på følgende måde:

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

Bemærk at instans-metoden adskiller sig fra den statiske metode ved at lade `Point`-parameteren være implicit. Når vi skriver `x` og `y` i instans-metodens krop er det underforstået at det er felterne `x` og `y` i det `Point`-objekt som kalder metoden.

Statiske metoder vs. instans-metoder

Vi har nu to forskellige typer af metoder i Java:

- **Instans-metoder.** Disse metoder er altid knyttet til objekter af en bestemt type, og bliver altid kaldt af objekter af denne type. Syntaksen for at anvende en instans-metode er:

  `<objektnavn>.<metodenavn>(<parametre>)`

- **Statiske metoder.** Disse metoder er ikke knyttet til objekter af en særlig type. Det "input" som en statisk metode opererer på er givet udelukkende ved de parametre som metoden tager som argument. Syntaksen for at anvende en instans-metode er:

  `<klassenavn>.<metodenavn>(<parametre>)`

  Her er `<klassenavn>` er navnet på klassen hvori den statiske metode er defineret. Hvis den statiske metode er defineret i samme klasse som den kaldes fra skriver man blot:

  `<metodenavn>(<parametre>)`

Eksempler med statiske metoder og instans-metoder

Eksempler på kald af **statiske metoder**:

```
Math.sqrt(5)
Arrays.sort(talListe)
```

Bemærk syntaksen: først navnet på klassen som indeholder den statiske metode, så navnet på selve metoden.

Eksempler på kald af **instans-metoder**:

```
p.translate(2,3)
s.length()
```

Bemærk syntaksen: først navnet på objektet som kalder metoden, så navnet på selve metoden.

Point-klassen

Når vi definerer en `Point`-klasse skal den både indeholde en definition af `Point`-objekters *tilstand* og deres *opførsel*. Hvis tilstanden er de to felter `x` og `y` og opførslen er instans-metoden `translate`, kommer klasse-definitionen til at se således ud:

```
1 public class Point {
2     int x;
3     int y;
4
5     public void translate(int dx, int dy) {
6         x += dx;
7         y += dy;
8     }
9 }
```

Når vi så opretter et `Point`-objekt `p` vil vi automatisk få adgang til objektets felter `p.x` og `p.y`, samt til objektets instans-metode `p.translate`.

Mere om Point-klassen

Betragt igen følgende klasse-definition:

```
1 public class Point {
2     int x;
3     int y;
4 
5     public void translate(int dx, int dy) {
6         x += dx;
7         y += dy;
8     }
9 }
```

Hvert nyt objekt af typen `Point` som bliver oprettet har ikke kun sin egen kopi af felterne `x` og `y`, men også sin egen kopi af metoden `translate`. Hvis `p1` og `p2` er forskellige `Point`-objekter er det derfor *ikke* det samme `x` der bliver ændret i nedenstående to metode-kald:

```
p1.translate(1,0);
p2.translate(1,0);
```

Eksempel med Point-objekter

Betragt igen følgende eksempel:

```
1  public class UsePoints {
2    public static void main(String[] args) {
3      Point p1 = new Point();  // opretter objekt p1
4      p1.x += 5;  // opdaterer tilstand af p1
5      p1.y += 7;  // opdaterer tilstand af p1
6      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
7      p1.x += 3;  // opdaterer tilstand af p1
8      p1.y += 4;  // opdaterer tilstand af p1
9      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
10     p1.x += 2;  // opdaterer tilstand af p1
11     p1.y += 3;  // opdaterer tilstand af p1
12     System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
13   }
14 }
```

Nu da vi har fået tilføjet `translate`-metoden til `Point`-klassen kan vi benytte denne til at forkorte ovenstående program...

Eksempel med Point-objekter – fortsat

Det modificerede program:

```
1  public class UsePoints {
2    public static void main(String[] args) {
3      Point p1 = new Point();  // opretter objekt p1
4      p1.translate(5,7);
5      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
6      p1.translate(3,4);
7      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
8      p1.translate(2,3);
9      System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
10   }
11 }
```

Ovenstående kode kalder metoden `translate` fra `Point`-klassen. Metoden kaldes direkte på `Point`-objektet. Vi opfatter det at forskyde sig at være en del af punktets egen *opførsel* eller *virkemåde*, og derfor er metoden placeret som en instans-metode i klassen der definerer `Point`-objekter.

Flere instans-metoder på Point-objekter

Vi kan også deklarere en metode til at returnere en streng-repræsentation af et `Point`-objekt, så vi slipper for hver gang at skulle skrive som ovenfor:

```
System.out.println("p1 er (" + p1.x + ", " + p1.y + ")");
```

Bør en sådan metode være en statisk metode eller en instans-metode? Egentlig er streng-repræsentationen af et punkt en naturlig del af punktet selv, så det bør skrives som en instans-metode i `Point`-klassen:

```
public String toString() {
   return "(" + x + ", " + y + ")";
}
```

Bemærk igen at `Point`-objektet er en implicit parameter i metodens deklaration. Vi refererer til det implicitte objekts `x`- og `y`-felter.

Hvis vi ønsker at få streng-repræsentationen af et konkret `Point`-objekt `p` kan vi nu skrive:

```
p.toString()
```

Klient-kode

Hvis vi tilføjer `toString`-metoden til `Point`-klassen, kan vi nu forkorte programmet fra før yderligere:

```
1  public class UsePoints {
2    public static void main(String[] args) {
3      Point p1 = new Point();  // opretter objekt p1
4      p1.translate(5,7);
5      System.out.println("p1 er " + p1.toString());
6      p1.translate(3,4);
7      System.out.println("p1 er " + p1.toString());
8      p1.translate(2,3);
9      System.out.println("p1 er " + p1.toString());
10   }
11 }
```

Kode der som ovenstående interagerer med en klasse og dens objekter kaldes **klient-kode**. Et fuldstændigt Java-program består ofte af en række klasser som definerer forskellige typer af objekter, og så klient-koden som benytter disse objekter.

Accessor og mutator

I programmet på foregående slide benyttede vi de to instans-metoder `toString` og `translate`. Disse metoder er henholdsvis en *accessor* og en *mutator*.

Accessor: En metode som giver information om et objekt uden at modificere det. *Eksempler*: `toString`, `distance`.

Mutator: En metode som modificerer et objekt. *Eksempler*: `translate`, `setLocation`.

Flere instans-metoder på Point-objekter

Vores `Point`-klasse har nu følgende instans-metoder:

```
1 public void translate(int dx, int dy) {
2     x += dx;
3     y += dy;
4 }
5
6 public String toString() {
7     return "(" + x + ", " + y + ")";
8 }
```

Vi kan naturligvis tilføje flere instans-metoder til klassen som vi nu finder det passende eller påkrævet for vores applikationer.

Opgave: Skriv en instans-metode `distanceFromOrigin` som tager et `Point`-objekt og beregner dets afstand til koordinatsættet $(0, 0)$.

Konstruktører

I klientkoden fra det tidligere eksempel konstruerede vi et nyt `Point`-objekt med følgende kommando:

```
Point p1 = new Point();
```

Dette punkt blev initialiseret med koordinaterne $(0, 0)$.

Metoder der opretter og initialiserer objekter kaldes som bekendt **konstruktører**. Enhver klasse definerer automatisk en *standard-konstruktør* hvis ikke vi definerer en. Standard-konstruktøren initialiserer alle felter i det konstruerede objekt til deres standard-værdi (samme værdi som ved automatisk initialisering af arrays).

Når vi skriver `new Point()` i vores `Point`-klasse fra før er det således denne standard-konstruktør som bliver benyttet.

Ofte ønsker vi at definere vores egen konstruktører, f.eks. så et punkt kan initialiseres med vilkårlige koordinater:

```
Point p1 = new Point(2,3);
```

Syntaks for deklaration af konstruktører

Et objekts konstruktører deklareres i klassen sammen med dets felter og instans-metoder.

Syntaks for deklaration af en konstruktør:

```
public <type>(<parametre>) {
   <liste af kommandoer>
}
```

Eksempel. Deklaration af en `Point`-konstruktør som tager to parametre til initialisering af x- og y-koordinaterne:

```
public Point(int x0, int y0) {
   x = x0;
   y = y0;
}
```

Når vi kalder `new Point(x0,y0)` vil vi så få returneret et nyt `Point`-objekt hvis `x`-felt er sat til værdien `x0` og hvis `y`-felt er sat til værdien `y0`.

En komplet Point-klasse

En komplet (men beskeden) `Point`-klasse som inkluderer både felter, en konstruktør og en instans-metode:

```
1  public class Point {
2      // felter
3      int x;
4      int y;
5
6      // konstruktør
7      public Point(int x0, int y0) {
8          x = x0;
9          y = y0;
10     }
11
12     // instans-metode
13     public void translate(int dx, int dy) {
14         x += dx;
15         y += dy;
16     }
17 }
```

Mere klient-kode

Klient-klassen `UsePoints` fra før kan nu omskrives til at benytte den nye version af klassen med vores egen konstruktør:

```
1  public class UsePoints {
2     public static void main(String[] args) {
3        Point p1 = new Point(5,7);
4        System.out.println("p1 er " + p1.toString());
5        p1.translate(3,4);
6        System.out.println("p1 er " + p1.toString());
7        p1.translate(2,3);
8        System.out.println("p1 er " + p1.toString());
9      }
10 }
```

Private felter

Det er ikke altid fordelagtigt at give ens klient-kode mulighed for at ændre direkte i et objekts felter. Hvis vi ønsker at hindre dette kan vi erklære felterne som **private**. Det gøres ved at tilføje nøgleordet `private` til deklarationen af felterne.

Eksempel. En variant af `Point`-klassen med private felter:

```
public class Point {
   private int x;
   private int y;

   ...
}
```

General syntaks for deklaration af private felter:

```
private <type> <navn>;
```

Indkapsling

Hvis `Point`-klassens felter er private kan vi ikke tilgå dem direkte.

Ofte skrives da accessor-metoder til at returnere felternes værdier:

```
public int getX() {
   return x;
}
```

Af og til skrives også mutator-metoder til at modificere felternes værdier:

```
public void setX(int newX) {
   x = newX;
}
```

Strategien med at gøre felter private så de ikke kan tilgås direkte udefra kaldes **indkapsling** (eng.: **encapsulation**). Indkapsling virker umiddelbart besværligt i forhold til blot at lade alle felter være direkte tilgængelige udefra. Der er dog et par vigtige pointer i det...

Fordele ved indkapsling

Vigtige fordele ved indkapsling:

- Ved at indkapsle felter (deklarere dem som private) kan vi undgå at klient-koden foretager ugyldige ændringer af et objekts tilstand. *Eksempel*: Antag vi ønsker at implementere en klasse `FirstQuadrant`, hvor objekterne repræsenterer punkter i første kvadrant af planen, dvs. punkter (x, y) med $x \geq 0$ og $y \geq 0$. Vi kan repræsentere sådanne punkter ved to felter af typen `double`. Men hvis felterne ikke er private kan vi risikere at klient-koden sætter negative koordinatværdier.
- Ved at indkapsle felter holder vi muligheden åben for at ændre på den interne repræsentation af objekternes tilstand. *Eksempel*: Måske finder vi på et tidspunkt ud af det er lettere at repræsentere `FirstQuadrant`-objekter internt i polære koordinater. Hvis klassens felter er indkapslede behøver vi kun ændre i `FirstQuadrant`-klassen, ikke alle klient-klasserne.

Eksempel med indkapsling

Hvis vi ønsker at implementere en `FirstQuadrant`-klasse må vi sørge for at objekter af typen `FirstQuadrant` aldrig kan få koordinater der ligger udenfor første kvadrant.

Hvis klassen skal indeholde en instans-metode `setLocation` der sætter et objekts x- og y-koordinater, skal vi altså sørge for at denne metode aldrig kan sætte hverken x- eller y-koordinaten til en negativ værdi. Hvis klient-koden forsøger at sætte ugyldige koordinatværdier kan vi lade klassen kaste en undtagelse:

```
1 public void setLocation(double newX, double newY) {
2     if (newX < 0 || newY < 0) {
3         throw new IllegalArgumentException();
4     }
5     x = newX;
6     y = newY;
7 }
```

Klasse-invarianter

```
1  public class FirstQuadrant {
2      private double x;
3      private double y;
4
5      public void setLocation(double newX, double newY) {
6          if (newX < 0 || newY < 0) {
7              throw new IllegalArgumentException();
8          }
9          x = newX;
10         y = newY;
11     }
12 }
```

En betingelse som en bestemt type af objekter er garanteret at opfylde kaldes en **klasse-invariant** (eng.: **class invariant**). Klassen ovenfor overholder invarianten at både `x` og `y` er ikke-negative.

Overholder standard-konstruktøren for klassen også invarianten?

Mere om `toString`

Enhver klasse deklarerer automatisk en standard `toString`-metode hvis ikke vi selv gør det. Men denne metode returnerer blot en streng repræsenterende adressen på objektet, hvilket sjældent er hvad vi ønsker. Derfor bør man altid definere sin egen `toString`-metode.

Metoden `toString` kaldes automatisk når et objekt skal printes eller konkateneres med en anden streng. Vi behøver altså ikke som vi gjorde tidligere at skrive:

```
System.out.println("p1 er " + p1.toString());
```

Vi kan i stedet nøjes med at skrive følgende, som Java automatisk fortolker på samme måde:

```
System.out.println("p1 er " + p1);
```

Nøgleordet `this`

Betragt følgende instans-metode i `Point`-klassen:

```
public void setLocation(int newX, int newY) {
    x = newX;
    y = newY;
}
```

Vi kunne i stedet have deklareret metoden på følgende ækvivalente måde:

```
public void setLocation(int newX, int newY) {
    this.x = newX;
    this.y = newY;
}
```

Ordet `this` er et nøgleord i Java som indeholder en reference til den implicitte parameter. Hvis den implicitte parameter er et `Point`-objekt kan dets felter `x` og `y` således altid refereres til som `this.x` og `this.y`.

Mere om nøgleordet `this`

Ved at gøre brug af nøgleordet `this` kan vi endda deklarere instans-metoden `setLocation` på følgende måde:

```
public void setLocation(int x, int y) {
   this.x = x;
   this.y = y;
}
```

Nøgleordet `this` benyttes her til at sørge for at vi kan skelne mellem metodens formelle parametre (`x` og `y`) og felterne i den implicitte parameter (`this.x` og `this.y`).

Antag `p` er et `Point`-objekt og betragt følgende metodekald:

```
p.setLocation(2,3);
```

I starten af udførslen af dette metodekald sættes `this` til at referere til det samme objekt som `p`. Det betyder at når vi i metodens krop modificerer værdien af `this.x`, så er det faktisk værdien af `p.x` vi modificerer.

Nøgleordet `this` og multiple konstruktører

En klasse kan have mange konstruktører, og en konstruktør kan kalde en anden konstruktør som i følgende eksempel.

```
public Point() {
   this(0, 0); // kalder Point(int, int) konstruktøren
}

public Point(int x0, int y0) {
   x = x0;
   y = y0;
}
```

Bemærk at det ikke er muligt at benytte `new Point(0, 0)` i stedet for `this(0, 0)`, eftersom dette blot ville skabe endnu et nyt objekt (som ikke kan erstatte det aktuelle objekt).

Kohæsion og kobling

Når man designer et større Java-program, opdeler man det typisk i en række forskellige klasser. Spørgsmålet er naturligvis *hvordan* denne opdeling skal foregå. Som tommefingerregel stræber man efter *stærk kohæsion* og *svag kobling*.

Kohæsion (eng.: **cohesion**): En klasses kohæsion er graden af indre sammenhæng i klassen. En klasse har stærk kohæsion hvis dens objekter repræsenterer en enkel abstraktion — hvis dens objekter har et enkelt overskueligt, velafgrænset og let forståeligt ansvarsområde.

Kobling (eng.: **coupling**): En klasses kobling er graden af afhængighed af andre klasser. Et program har svag kobling hvis dets klasser kun afhænger af hinanden i begrænset omfang.

Ved at stræbe mod stærk kohæsion og svag kobling opnår man at de enkelte klasser i størst mulig omfang kan genbruges, og at eventuelle ændringer skal foretages så få steder som muligt.