Föreläsning 4

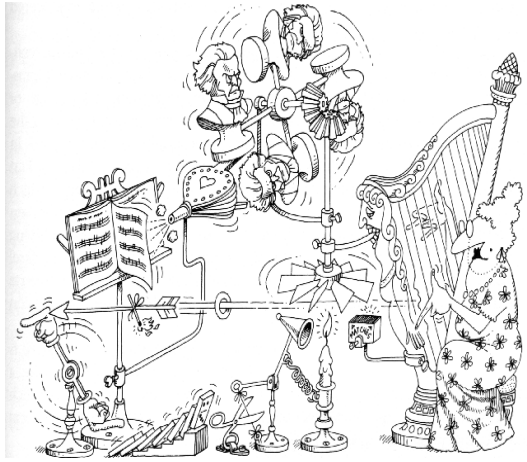
Top-Down Design
Metoder
Parameteröverföring

Programmering = modellering

Ett datorprogram är en ***modell*** av en verklig eller tänkt värld. Ofta är det komplexa system som skall modelleras

I objektorienterad programmering består denna värld av ett antal objekt som tillsammans löser den givna uppgiften.

- De enskilda objekten har specifika ansvarsområden.
- Objekten samarbetar genom att kommunicera med varandra via meddelanden.
- Ett meddelande till ett objekt är en begäran från ett annat objekt att få något utfört.



Att göra en bra modell av verkligheten, och därmed möjliggöra en bra design av programmet, är en utmaning.

Abstraktion

För att lyckas utveckla ett större program måste man arbeta efter en ***metodik***.

En mycket viktig princip vid all problemlösning är att använda sig av ***abstraktioner***.

En abstraktion innebär att man bortser från vissa omständigheter och detaljer i det vi betraktar, för att bättre kunna uppmärksamma andra för tillfället mer väsentliga aspekter.

Abstraktion är det viktigaste vi har verktyg för att hantera komplexitet och för att finna gemensamma drag hos problem och hitta generella lösningar.

Betraktas alla detaljer *ser man inte skogen för alla träden* och två problem kan synas helt olika, medan de på en hög abstraktionsnivå är identiska.

Top-Down Design

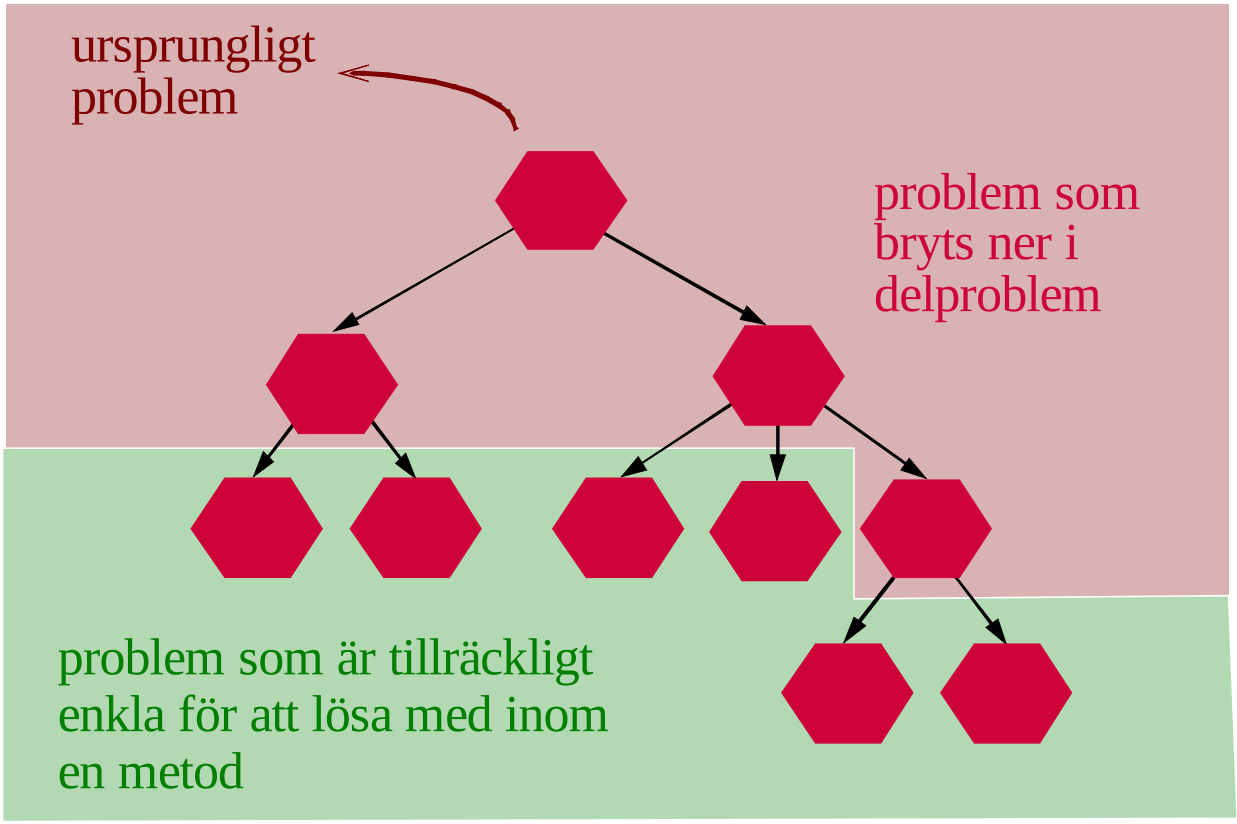
En problemlösningsmetodik som bygger på användning av abstraktioner är ***top-down design***.

Top-down design innebär att vi betraktar det ursprungliga problemet på en hög ***abstraktionsnivå*** och bryter ner det ursprungliga problemet i ett antal delproblem.

Varje delproblem betraktas sedan som ett separat problem, varvid fler aspekter på problemet beaktas, dvs vi arbetar med problemet på en lägre abstraktionsnivå än vi gjorde med det ursprungliga problemet.

Om nödvändigt bryts delproblemen ner i mindre och mer detaljerade delproblem. Denna process upprepas till man har delproblem som är enkla att överblicka och lösa. Top-down-design bygger på principen ***divide-and-conquer***.

Top-Down Design



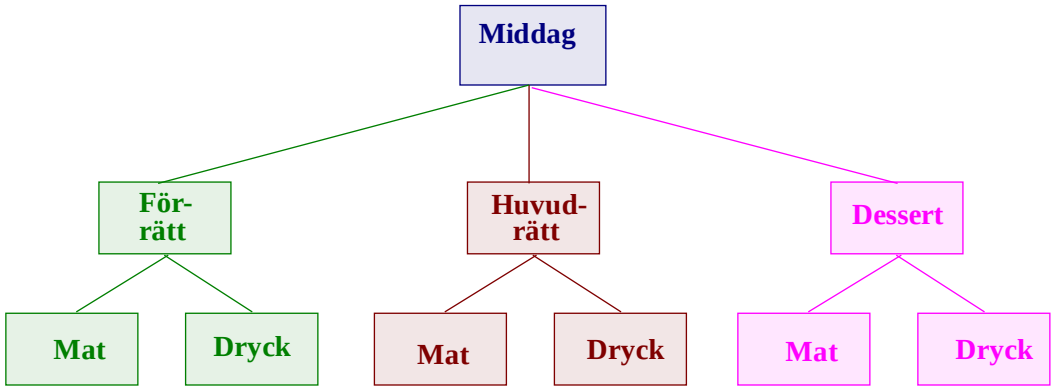
Med top-down design blir det möjligt att lösa det ursprungliga problemet steg för steg istället för att direkt göra en fullständig lösning.

Top-Down Design

Allteftersom ett problem bryts ner i mindre delproblem, betraktar man allt fler detaljer. Vi går således från en abstrakt nivå mot allt mer detaljerade nivåer. Denna process brukar kallas för ***stegvis förfining***.

Exempel:

Att ordna en tre-rätters middag enligt "top-down design"



Modulär Design

Vid utveckling av Javaprogram är klasser och metoder (tillsammans med paket) de ***abstraktionsmekanismer*** som används för att dölja detaljer och därmed *öka överblickbarhet och förståelse*.

Att utveckla ett Javaprogram med hjälp av top-down design innebär således att dela in programmet i lämpliga klasser och metoder, vilka i sin tur delas upp i nya klasser och metoder.

Man skall eftersträva en ***modulär design*** där varje delproblem (= klass eller metod) handhar en *väl avgränsad uppgift* och att varje delproblem är så *oberoende* av de andra delproblemen som möjligt.

Modulär design

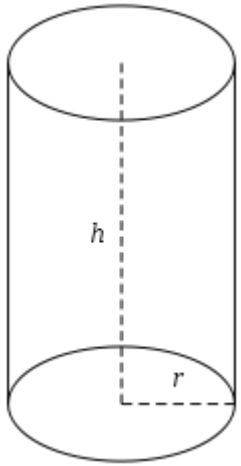
En *välgjord* modulär design innebär att programsystemet är uppdelat i *tydligt identifierbara abstraktioner*. Fördelarna med ett sådant system är:

- det går lätt att utvidga
- komponenterna går att återanvända
- komponenterna har en tydlig uppdelning av ansvar
- komplexiteten reduceras
- komponenterna går att byta ut
- underlättar testning
- tillåter parallell utveckling.

*Designa programsystemet runt **stabila abstraktioner** och **utbytbara komponenter** för att möjliggöra små och stegvisa förändringar.*

Exempel:

Skriv ett program som läser in radien och höjden av en cylinder, samt beräknar och skriver ut cylinderns area och volym.



Arean *A* och volymen *V* av en cylinder fås av följande formler:

$$A = 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2 \text{ och } V = \pi \cdot r^2 \cdot h$$

, där *r* är radien och *h* är höjden av cylindern.

Utkast till lösning:

1. Läs cylinderns radie *r*.
2. Läs cylinders höjd *h*.
3. Beräkna cylinderns area *A* mha formeln $A = 2 \cdot \pi \cdot r \cdot h + 2 \cdot \pi \cdot r^2$.
4. Beräkna cylinderns volym *A* mha formeln $V = \pi \cdot r^2 \cdot h$.
5. Skriv ut cylinderns area *A* och volym *V*.

Lösning 1:

Var och ett stegen i vår lösningsskiss är mer eller mindre triviala varför programmet kan skrivas som ett enda huvudprogram:

```
/*Programmet läser in radien och höjden av en cylinder, samt beräknar och skriver ut arean och
  volymen av cylindren */
import javax.swing.*;
import java.util.*;
public class Cylinder {
    public static void main (String[] arg) {
        String input = JOptionPane.showInputDialog("Ange cylinderns" + " radie och höjd:");
        Scanner sc = new Scanner(input);
        double radius  = sc.nextDouble();
        double height = sc.nextDouble();
        double area = 2*Math.PI*radius*height + 2*Math.PI*Math.pow(radius, 2);
        double volume = Math.PI * Math.pow(radius, 2) * height;
        JOptionPane.showMessageDialog(null, "Arean av cylindern är "+ area
                                              + "\nVolymen av cylindern är " + volume);
    } //main
} //Cylinder
```

Lösning 2:

I lösningsskissen utgör beräkningen av arean respektive beräkningen av volymen var sitt delproblem och kan därmed implementeras som var sin metod!

```
/*Programmet läser in radien och höjden av en cylinder, samt beräknar och skriver ut arean och
  volymen av cylindren */
import javax.swing.*;
import java.util.*;
public class Cylinder2 {
   public static void main (String[] arg) {
      String input = JOptionPane.showInputDialog("Ange cylinderns" + " radie och höjd:");
      Scanner sc = new Scanner(input);
      double radius  = sc.nextDouble();
      double height = sc.nextDouble();
      double area = computeArea(radius, height);
      double volume = computeVolume(radius, height);
      JOptionPane.showMessageDialog(null, "Arean av cylindern är "+ area
                                          + "\nVolymen av cylindern är " + volume);
   } //main
```

Lösning 2: fortsättning

```
    private static double computeArea(double radius, double height) {
        return  2*Math.PI*radius*height + 2*Math.PI*Math.pow(radius, 2);
    }//computeArea

    private static double computeVolume(double radius, double height) {
        return Math.PI * Math.pow(radius, 2) * height;
    } //computeVolume
} //Cylinder2
```

Kommentar:

Vi har deklarerat metoderna computeArea och ComputeVolume **private** eftersom de är ***hjälpmetoder*** för att huvudprogrammet skall kunna göra sin uppgift.

Lösning 3:

Arean av en cylinder beräknas med hjälp av cylinderns mantelyta samt cylinderns cirkelyta, och även volymen beräknas med hjälp av cirkelytan. Därför kan vi bryta ner problemen att beräkna cylinderns area och volym i ytterligare delproblem.

```
private static double computeArea(double radius, double height) {
   return  computeSideArea(radius, height) + 2*computeCircleArea(radius);
}//computeArea

private static double computeVolume(double radius, double height) {
   return computeCircleArea(radius) * height;
} //computeVolume

private static double computeSideArea(double radius, double height) {
   return  2*Math.PI*radius*height;
}//computeSideArea

private static double computeCircleArea(double radius) {
   return  Math.PI*Math.pow(radius, 2);
}//computeCircleArea
```

Lösning 4:

I föregående lösning har vi en klass som innehåller både ett huvudprogram och de *privata* klassmetoderna computeArea, computeVolume, computeSideArea och computeCircleArea. Det är även möjligt (och lämpligt) att lägga dessa metoder i en annan klass och än huvudprogrammet. Metoderna måste då göras *publika*.

```
public class Formulas {
    public static double computeArea(double radius, double height) {
        return  computeSideArea(radius, height) + 2*computeCircleArea(radius);
    }//computeArea

    public static double computeVolume(double radius, double height) {
        return computeCircleArea(radius) * height;
    } //computeVolume

    public static double computeSideArea(double radius, double height) {
        return  2*Math.PI*radius*height;
    }//computeSideArea

    public static double computeCircleArea(double radius) {
        return  Math.PI*Math.pow(radius, 2);
    }//computeCircleArea
}//Formulas
```

Vad är fördelarna med denna design?

Lösning 4: fortsättning

Vårt huvudprogram får då följande utseende:

```
/*Programmet läser in radien och höjden av en cylinder, samt beräknar och skriver ut
   arean och volymen av cylindren */
import javax.swing.*;
import java.util.*;
public class Cylinder4 {
   public static void main (String[] arg) {
      String input = JOptionPane.showInputDialog("Ange cylinderns" + " radie och höjd:");
      Scanner sc = new Scanner(input);
      double radius  = sc.nextDouble();
      double height = sc.nextDouble();
      double area = Formulas.computeArea(radius, height);
      double volume = Formulas.computeVolume(radius, height);
      JOptionPane.showMessageDialog(null, "Arean av cylindern är "+ area
                                          + "\nVolymen av cylindern är " + volume);
   } //main
}//Cylinder4
```

Anm: För att programmet skall fungera måste klassen Formulas ligger i samma mapp som klassen Cylinder4.

Bottom-Up Design

Ett alternativ till top-down design är ***bottom-up design***.

Bottom-up design innebär att man startar med att utveckla små och *generellt användbara* programenheter och sedan bygger ihop dessa till allt större och kraftfullare enheter.

En viktig aspekt av objektorienterad programmering, som ligger i linje ned bottom-up design, är ***återanvändning***. Återanvändning innebär en strävan att skapa klasser som är så generella att de kan användas i många program.

I Java finns ett ***standardbibliotek*** som innehåller ett stort antal sådana generella klasser. Standardbiblioteket kan således ses som en *"komponentlåda*" ur vilken man kan plocka komponenter till det programsystem man vill bygga.

Vid utveckling av Javaprogram kombinerar man vanligtvis top-down design och bottom-up design.

Uppbyggnaden av en metod

```
//Utseende på metoder som lämnar returvärde
 modifierare typ namn(parameterlista) {
      dataattribut och satser
      return uttryck;
 }
```

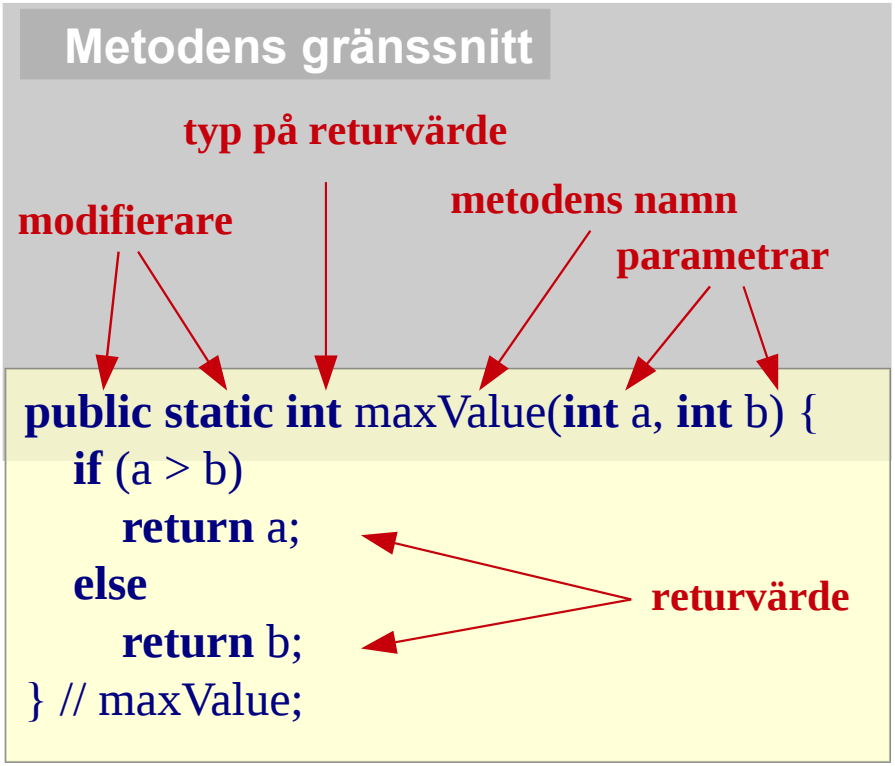
```
//Utseende på metoder som inte lämnar returvärde
 modifierare void namn(parameterlista) {
      dataattribut och satser
 }
```

Metoder kan antingen vara klassmetoder eller instansmetoder.

Metoder kan antingen lämna ett returvärde eller inte lämna ett returvärde.

Metoder kan bl.a. vara **private** eller **public**.

Uppbyggnaden av en metod



Satsen

return *uttryck*;

terminerar metoden och värdet *uttryck* blir resultatet som erhålls från metoden.

En metod som inte lämnar något värde (en ***void-metod***) har ingen **return**-sats eller har **return**-satser som saknar *uttryck*.

Uppbyggnaden av en metod

För att kunna använda en metod måste man känna till och kunna använda ***metodens gränssnitt*** på ett korrekt sätt. En metods gränssnitt bestäms av

- metodens *namn*
- metodens *returtyp*
- metodens *parameterlista* avseende antal parametrar samt parametrarnas typer och ordning
- huruvida metoden är en *klassmetod eller instansmetod*

Ett metodanrop kan ses som att en avsändare skickar ett meddelande till en mottagare. Parameterlistan beskriver vilken typ av data avsändaren kan skicka i meddelandet och resultattypen beskriver vilken typ av svar avsändaren får i respons från mottagaren.

Formella och aktuella parametrar

```
import javax.swing.*;
import java.util.*;
public class Exempel {
   public static int  maxValue(int a, int b) {
      if (a  > b)
         return a;
      else
         return b;
   } //maxValue
   public static void main(String[] args) {
      String input = JOptionPane.showInputDialog("Ge tre heltal");
      Scanner sc = new Scanner(input);
      int value1 = sc.nextInt();
      int value2 = sc.nextInt();
      int value3 = sc.nextInt();
      int big = maxValue(value1, value2);
      big = maxValue(big, value3);
      JOptionPane.showMessageDialog(null, "Det största av talen " + value1 + ", "
                                          + value2 + " och " + value3 + " är " + big);
   } // main
} //Exempel
```

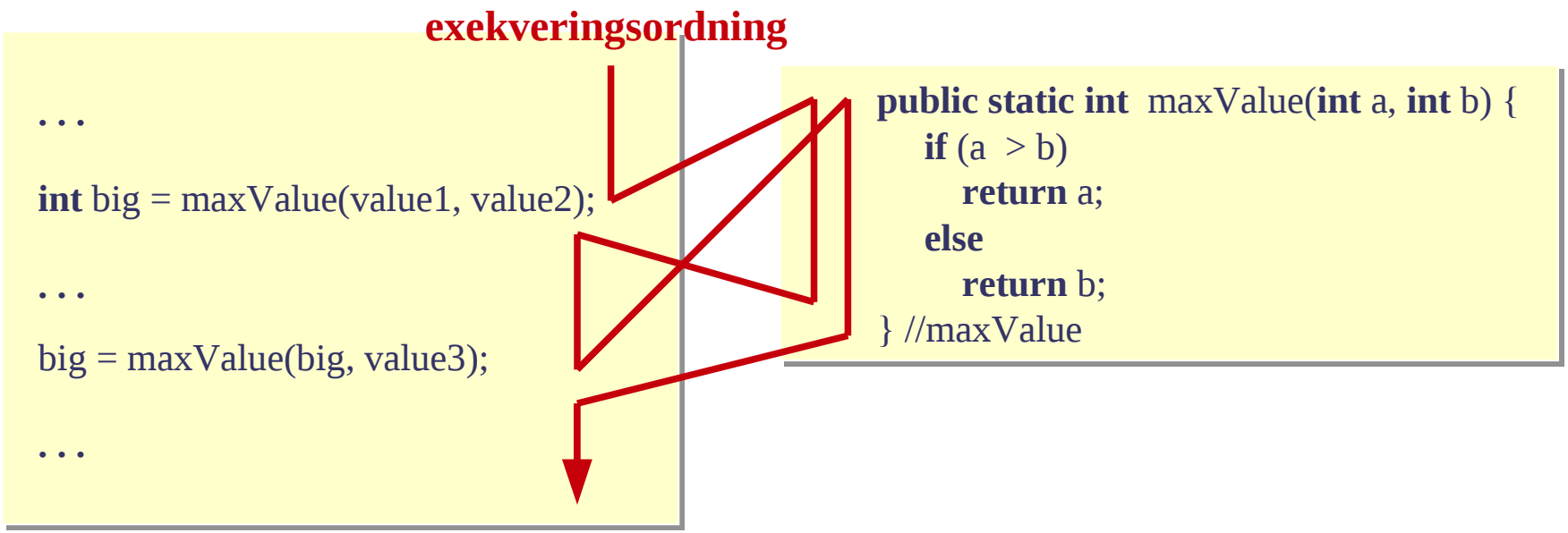
formella parametrar

aktuella parametrar

Metodanrop

Vid anrop av en metod sker följande:

- värdet av de aktuella parametrarna kopieras till motsvarande formell parameter
- exekveringen fortsätter med den första satsen i den anropade metoden
- när exekveringen av den anropade metoden är klar återupptas exekveringen i den metod där anropet gjordes



Parameteröverföring

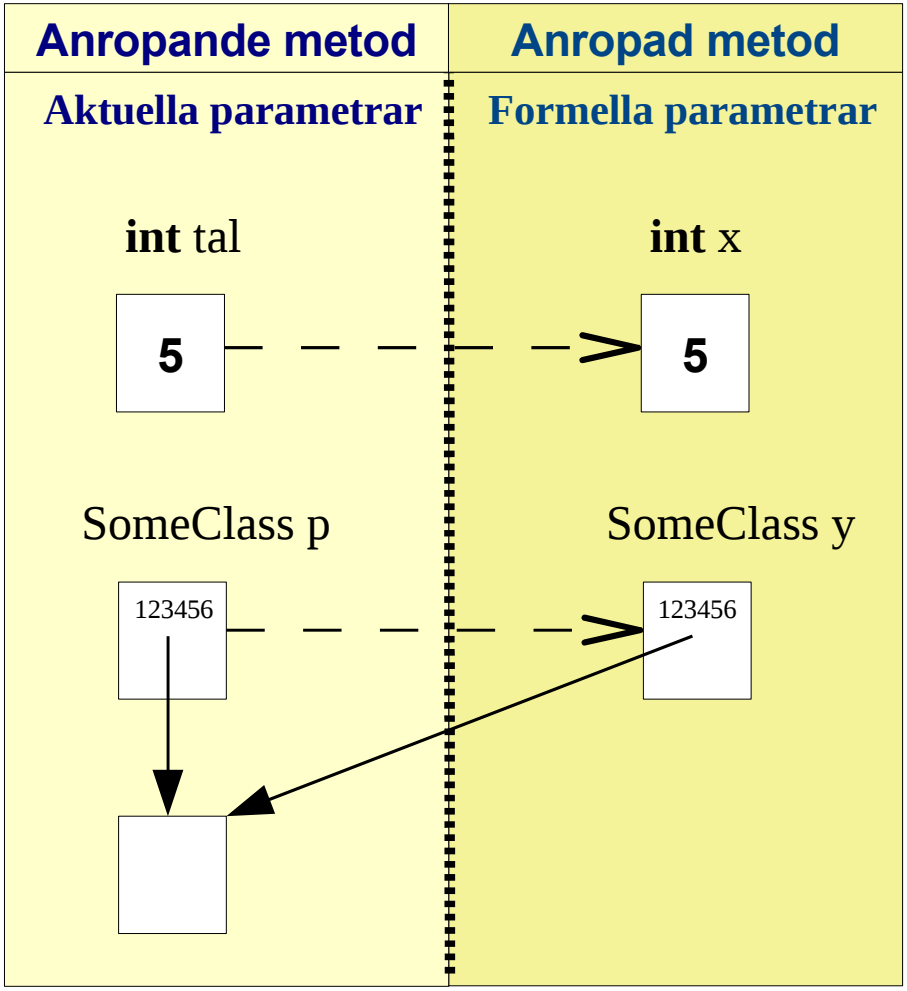
Alla primitiva datatyper och alla existerande klasser kan ges i parameterlistan och/eller som resultattyp.

Parameterlistan kan innehålla ett godtyckligt antal parametrar

I Java sker alltid parameteröverföring via ***värdeanrop***, vilket betyder att *värdet av den aktuella parametern kopieras över till den formella parametern*.

- När den aktuella parametern är en *primitiv typ* kommer därför den aktuella parametern och den formella parametern att ha access till *fysiskt åtskilda objekt*.
- När parametern är ett *objekt* (dvs en instans av en klass) är parametern en referensvariabel, varför den aktuella parametern och den formella parametern kommer att ha access till *samma fysiska objekt*.

Parameteröverföring



Värdet av den aktuella parametern tal kopieras till den formella parametern x.

tal och x är *åtskilda fysiska objekt*.

En förändring av värdet i variabeln x påverkar inte värdet i variabeln tal.

Värdet av den aktuella parametern p kopieras till den formella parametern y.

p och y kommer att referera till *samma fysiska objekt*.

En förändring *i objektet* som refereras av variabeln y påverkar därför objektet som refereras av variabeln p, eftersom det är samma objekt

Testning

Hur förvissar vi oss om att bredvidstående program är korrekt?

Genom testning!!!

Modulär design underlättar testning, eftersom varje metod kan testas individuellt.

Gör en modulär design av programmet!

```
import javax.swing.*;
public class Postage {
   public static void main( String[] arg) {
      String input = JOptionPane.showInputDialog("Ange vikten:");
      double weight = Double.parseDouble(input);
      String output;
      if (weight <= 0.0)
         output = "Du har angivit en ogiltig vikt!!";
      else if (weight <=20.0)
         output = "Portot är 5.50 kronor.";
      else if (weight <= 100.0)
         output = "Portot är 11.00 kronor.";
      else if (weight <=250.0)
         output = "Portot är 22.00 kronor.";
      else if (weight <=500.0)
         output = "Portot är 33.00 kronor.";
      else
         output = "Måste gå som paket.";
      JOptionPane.showMessageDialog(null, output);
   } // main
} // Postage
```

Modulär design av Postage

```
import javax.swing.*;
public class Postage {
   public static void main( String[] arg) {
      String input = JOptionPane.showInputDialog("Ange vikten:");
      double weight = Double.parseDouble(input);
      JOptionPane.showMessageDialog(null, getPostage(weight));
   } // main

   public static String getPostage(double weight) {
      if (weight <= 0.0)
         return "Du har angivit en ogiltig vikt!!";
      else if (weight <=20.0)
         return "Portot är 5.50 kronor.";
      else if (weight <= 100.0)
         return "Portot är 11.00 kronor.";
      else if (weight <=250.0)
         return "Portot är 22.00 kronor.";
      else if (weight <=500.0)
         return "Portot är 33.00 kronor.";
      else
         return "Måste gå som paket.";
   }//getPostage
} // Postage
```

Fel i program

När man skriver program uppkommer alltid fel. Felen kan indelas i följande kategorier:

- Under kompileringen upptäcker kompilatorn fel som handlar om att man använt konstruktionerna i programspråket på ett felaktig sätt (***kompileringsfel***). Kompileringsfelen kan indelas i ***syntaktiska fel*** och ***semantiska fel***.

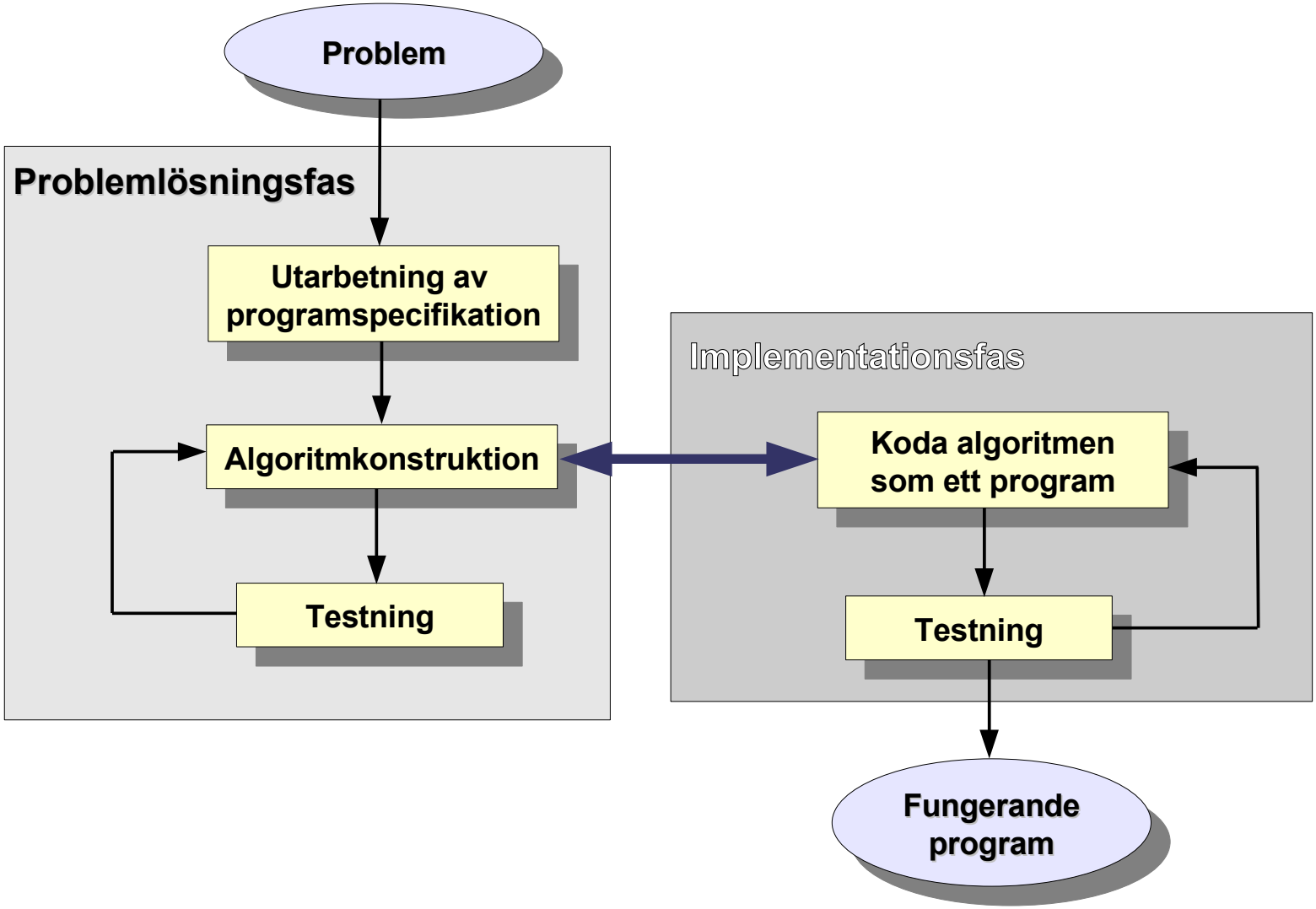
- I ett exekveringsbart program kan det förekomma två typer av fel - ***exekveringsfel*** och ***logiska fel***.

  - Exekveringsfel uppkommer t.ex. på grund av att det under exekveringen någonstans i programmet sker en evaluering av ett uttryck som resulterar i att ett värde erhålls som ligger utanför det giltiga definitionsområdet för uttryckets datatyp. Felen uppträder vanligtvis inte vid varje körning utan endast då vissa specifika sekvenser av indata ges till programmet.
  - Logiska fel är rena tankefel hos programmeraren. Exekveringen lyckas, men programmet gör inte vad det borde göra.

Testning av program

- Testning är en vedertagen metod inom all ingenjörsmässig verksamhet för att fastställa om en hypotes, konstruktion eller produkt är korrekt och fungerar som avsett.
- Till grund för all testning av program ligger *programspecifikationen*. En dålig eller felaktig specifikation leder naturligtvis till ett undermåligt eller felaktigt program.
- En ***testplan***, som anger hur det färdiga programmet skall fungera, bör utarbetas samtidigt med programspecifikationen.
- Testning är ett sätt att minimera antalet fel i ett program.
- Testning kan ***enbart påvisa förekomsten av fel, aldrig frånvaron av fel!***
- Testning skall ske i samtliga faser av programutvecklingen.

Verklig modell för programutveckling



Testning av program

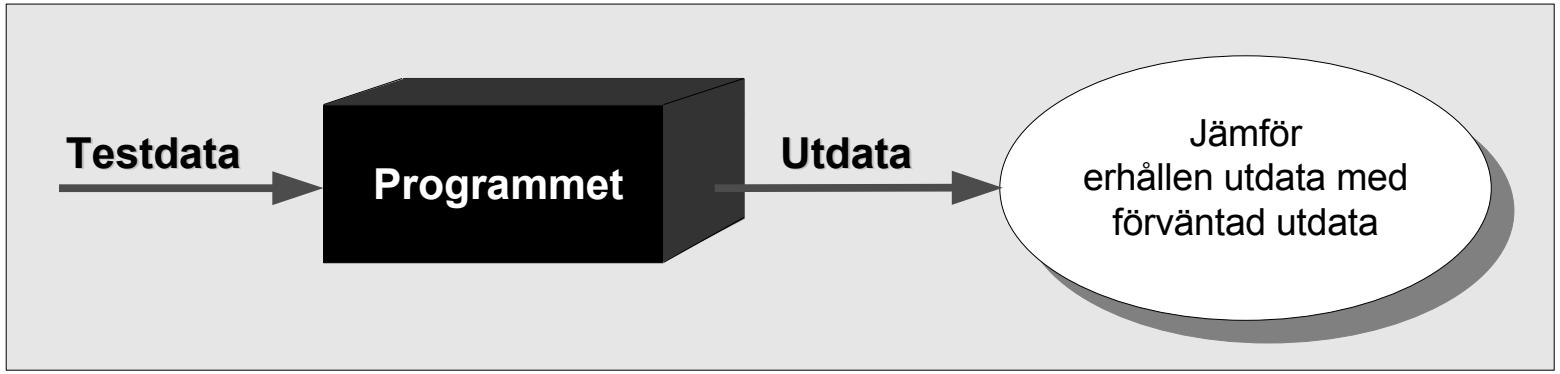
Ju senare i utvecklingsarbetet man upptäcker ett fel ju svårare och dyrare är det att lokalisera och korrigera felet.

En algoritm testas manuellt genom att gå igenom algoritmen och utför de olika stegen i algoritmen. Syftet är att verifiera att algoritmen inte innehåller några *logiska fel*.

Under testning av algoritmen kan det visa sig att programspecifikationen är felaktig eller ofullständig, då måste man backa tillbaks till program-specifikationen och göra nödvändiga modifieringar eller kompletteringar.

Testning av program

Vid leveranstestning görs ***black-box testning***, vilket innebär att testningen sätts upp utan kunskaper om hur programmet är implementerat.



Om felaktigheter upptäcks under testningen, skall felen naturligtvis åtgärdas. Därefter skall alla testfallen i testplanen återupprepas, eftersom korrigeringar av fel kan introducera nya fel.

Testning av program

Det är omöjligt att göra ***uttömmande testning***, dvs testa alla uppsättningar av möjliga indatasekvenser.

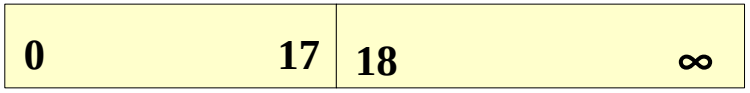
Men vi måste ha en uppsättning testfall som är så övertygande att man kan *anta* att programmet är korrekt.

Den metod som används är att indela de möjliga indatasekvenserna i olika ***ekvivalens kategorier***.

Testning av program

Exempel:

Anta att vi har ett program som skall skriva ut om en person får rösta eller inte. Röståldern är 18 år. Vi får då två ekvivalens kategorier enligt nedanstående figur:



Det finns dock ytterligare en ekvivalens kategorier, nämligen den som består av ogiltig data.

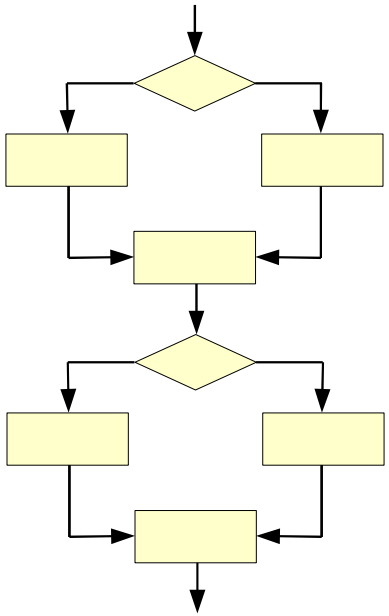
I testplanen väljs (minst) ett testfall från varje ekvivalens kategorier, samt testfall med värden från övergångarna mellan ekvivalens kategorierna. Vi får således följande testplan:

Test nr	Indata	Förväntat resultat
1	12	Får inte rösta
2	21	Får rösta
3	17	Får inte rösta
4	18	Får rösta
5	-10	Felutskrift
6	0	Får inte rösta
7	-1	Felutskrift

Testprogram för metoden getPostage:

```
import javax.swing.*;
public class TestGetPostage {
   public static void main( String[] arg) {
      System.out.print("Testfall 1: Förväntat värde: Du har angivit en ogiltig vikt!! ");
      System.out.println("Erhållet värde: " + Postage.getPostage(0));
      System.out.print("Testfall 2: Förväntat värde: Portot är 5.50 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(0.5));
      System.out.print("Testfall 3: Förväntat värde: Portot är 5.50 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(20.0));
      System.out.print("Testfall 4: Förväntat värde: Portot är 11.00 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(20.5));
      System.out.print("Testfall 5: Förväntat värde: Portot är 11.00 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(100.0));
      System.out.print("Testfall 6: Förväntat värde: Portot är 22.00 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(100.5));
      System.out.print("Testfall 7: Förväntat värde: Portot är 22.00 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(250.0));
      System.out.print("Testfall 8: Förväntat värde: Portot är 33.00 kronor. ");
      System.out.println("Erhållet värde: " + Postage.getPostage(250.5));
      System.out.print("Testfall 9: Förväntat värde: Portot är 33.00 kronor.");
      System.out.println("Erhållet värde: " + Postage.getPostage(500.0));
      System.out.print("Testfall 10: Förväntat värde: Måste gå som paket.");
      System.out.println("Erhållet värde: " + Postage.getPostage(500.5));
   }//main
}//TestGetPostage
```

White-box-testning



När man testar sina programkomponenter under implementationsfasen har man kännedom om hur implementationen är gjord.

Testmetoder som drar nytta av att implementationen är tillgänglig kallas för ***white-box testning***.

Vid white-box testning används kunskapen om programmets strukturella uppbyggnad när testdata väljs.

Det man eftersträvar vid white-box testning är att köra programmet med en uppsättning testfall som valts på så sätt att varje sats i programmet blir exekverad under testningen.

I stora program finns det enormt många olika exekveringasvägar, varför det i praktiken är omöjligt att göra en fullständig white-box testning.

Testning av komponenter och system

Varje programenhet skall testas separat från övriga enheter innan enheten integreras i programsystemet. Man då har större möjlighet att lokalisera och åtgärda uppkomna fel.

Att testa en programkomponent kallas för ***enhetstesting***.

Att testa det kompletta programsystemet kallas för ***systemtesting***.

Det finns två olika metoder för att testa de enskilda enheterna i ett programsystem, *bottom-up* och *top-down*.

Vid bottom-up utvecklas och testas de minsta och mest grundläggande enheterna först, varefter dessa kan användas som komponenter i större och mera kraftfulla enheter.

Vid top-down utvecklas och testas de största och mest abstrakta enheterna först. Då top-down är en vedertagen princip för utveckling av större program är det också lämpligt att testa stora program enligt samma princip.

Normalt användes en kombination av top-down och button-up testning.

Det är vedertagen praxis att utföra enhetstester och systemtest. Man bör dock inte gå direkt från enhetstestning till systemtestning, utan istället successivt utöka systemet med nya programdelar och utföra testningar allteftersom programdelarma integreras i systemet. Detta kallas för ***inkrementell testning***.

Förvillkor och eftervillkor

Av *specifikationen för en metod* skall framgå

- metodens namn
- metodens parameterlista
- metodens returtyp
- vad metoden gör
- vilka förutsättningar som måste gälla för att metoden skall fungera på ett korrekt sätt

Vad metoden gör beskrivs som ***eftervillkor*** (***postconditions***). Eftervillkoren anger dels vilket resultat metoden returnerar, dels vilka ***sidoeffekter*** metoden har.

Vilka förutsättningar som måste gälla för att fungera på avsett sätt beskrivs som ***förvillkor*** (***preconditions***).

Om klienten uppfyller förvillkoren metoden att uppfylla eftervillkoren.

Förvillkor och eftervillkor

```
public class Formulas {
    //Precondition: radius >= 0 && height >= 0
    //Returns: the area of a cylinder with assigned raduis and height
    public static double computeArea(double radius, double height) {
        return  computeSideArea(radius, height) + 2*computeCircleArea(radius);
    }//computeArea

    //Precondition: radius >= 0 && height >= 0
    //Returns: the volume of a cylinder with assigned raduis and height
    public static double computeVolume(double radius, double height) {
        return computeCircleArea(radius) * height;
    } //computeVolume

    //Precondition: radius >= 0 && height >= 0
    //Returns: the side area of a cylinder with assigned raduis and height
    public static double computeSideArea(double radius, double height) {
        return  2*Math.PI*radius*height;
    }//computeSideArea

    //Precondition: radius >= 0
    //Returns: the area of a circle with assigned raduis
    public static double computeCircleArea(double radius) {
        return  Math.PI*Math.pow(radius, 2);
    }//computeCircleArea
}//Formulas
```

En cylinder kan inte ha en radie inte ha en radie som är negativ och inte en höjd som är negativ.

En cirkel kan inte ha en radie som är negativ.