Föreläsning 4

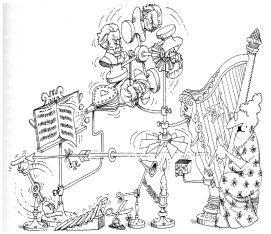
Top-Down Design
Metoder
Parameteröverföring

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 verktyget vi har 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.

> ***A person can only keep 7 plus or minus 2 items in mind at one time. (George Miller)***

Top-Down Design

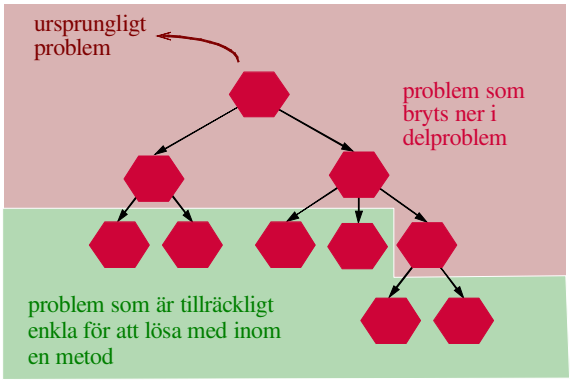
En problemlösningsmetodik som bygger på användning av abstraktioner är ***top-down design***.

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



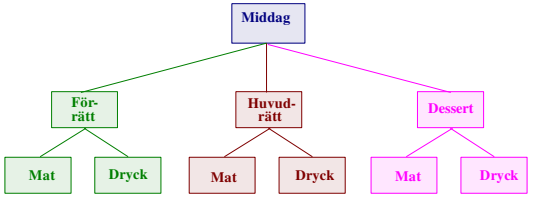
Med top-down design blir det möjligt att lösa det ursprungliga problemet steg för steg istället för att direkt göra en fullständig lösning.

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 trerätters 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.

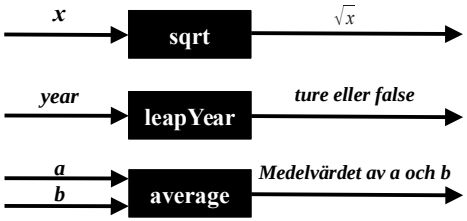
Modulär design

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.*

”Black box” - tänkande



Är ett sätt att uttrycka vissa steg i ett program på en *högre abstraktionsnivå*.

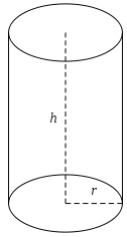
På en hög abstraktionsnivå är det *vad* som görs som är det intressanta, *inte hur* det görs.

- Ger direkt stöd för *stegvis förfining*.
- Varje abstraktion (black box) skall endast *göra en sak* och göra denna *bra*.

Vad en abstraktion gör och hur abstraktionen *används* definerar abstraktionens gränsitt (interface).

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

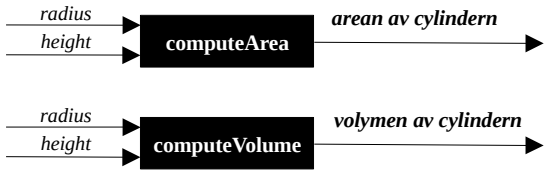
Nackdel:

Måste beakta alla detaljer samtidigt!

```
import javax.swing.*;
import java.util.*;
public class Cylinder {
    public static void main (String[] arg) {
        boolean done = false;
        while (!done) {
            String input = JOptionPane.showInputDialog("Ange cylinderns radie och höjd:");
            if (input == null)
                done = true;
            else {
                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 betraktas som var sin ”black -box”:



Vi implementera dessa som var sin metod.

```
//before: radius >= 0 && height >= 0
//return: The area of a cylinder with assigned raduis and height.
public static double computeArea(double radius, double height) { //todo }
```

```
//before: radius >= 0 && height >= 0
//return: The volume of a cylinder with assigned raduis and height.
public static double computeVolume(double radius, double height) { //todo}
```

Lösning 2: fortsättning

```
import javax.swing.*;
import java.util.*;
public class Cylinder2 {
   public static void main (String[] arg) {
       boolean done = false;
       while (!done) {
         String input = JOptionPane.showInputDialog("Ange cylinderns radie och höjd:");
         if (input == null)
             done = true;
          else {
             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
```

Anropa metoder för att utföra beräkningarna

Lösning 2: fortsättning

```
   //before: radius >= 0 && height >= 0
   //return: The area of a cylinder with assigned raduis and height.
    private static double computeArea(double radius, double height) {
       return  2*Math.PI*radius*height + 2*Math.PI*Math.pow(radius, 2);
    }//computeArea

   //before: radius >= 0 && height >= 0
   //return: The volume of a cylinder with assigned raduis and height.
    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 Utils {
    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
}//Utils
```

Vad är fördelarna med denna design?

Lösning 4: fortsättning

Vårt huvudprogram får då följande utseende:

```
import javax.swing.*;
import java.util.*;
public class Cylinder4 {
   public static void main (String[] arg) {
       boolean done = false;
       while (!done) {
          String input = JOptionPane.showInputDialog("Ange cylinderns radie och höjd:");
          if (input == null)
             done = true;
          else {
             Scanner sc = new Scanner(input);
             double radius  = sc.nextDouble();
             double height = sc.nextDouble();
             double area = Utils.computeArea(radius, height);
             double volume = Utils.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 Utils ligger i samma mapp som klassen Cylinder4.

Förvillkor och eftervillkor

För att kunna använda sig av en metod måste man känna till gränssnitt för metoden. Av gränsnittet 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 metoden skall fungera på avsett sätt beskrivs som ***förvillkor*** (***preconditions***).

Om klienten uppfyller förvillkoren lovar metoden uppfylla eftervillkoren.

Förvillkor och eftervillkor

```
public class Utils {
    //before: radius >= 0 && height >= 0
    //return: 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

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

    //before: radius >= 0 && height >= 0
    //return: 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

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

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

En cirkel kan inte ha en radie som är negativ.

javadoc

```
public class Utils {
    /**
         @before radius >= 0 && height >= 0
         @return 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

    /**
         @before radius >= 0 && height >= 0
         @return The volume of a cylinder with assigned raduis and height.
    */
    public static double computeVolume(double radius, double height) {
        return computeCircleArea(radius) * height;
    } //computeVolume

    /**
         @before radius >= 0 && height >= 0
         @return 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

    /**
         @before radius >= 0
         @return The area of a circle with assigned raduis.
    */
    public static double computeCircleArea(double radius) {
        return  Math.PI*Math.pow(radius, 2);
    }//computeCircleArea
}//Utils
```

@return är en fördefinierad annotation

@before är en *egendefinierad* annotation

Kommandot

```
javadoc -tag before:a:"Before:"  Utils.java
```

skapar en dokumentation av klassen Utils i form av en uppsättning html-filer.

computeArea

```
public static double computeArea(double radius,
                                 double height)
```

Returns:
the area of a cylinder with assigned raduis and height
Before:
radius >= 0 && height >= 0

computeVolume

```
public static double computeVolume(double radius,
                                   double height)
```

Returns:
the volume of a cylinder with assigned raduis and height
Before:
radius >= 0 && height >= 0

Exempel på andra fördefinerade annotationer:

@author
@version
@exception
@param

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;
}
```

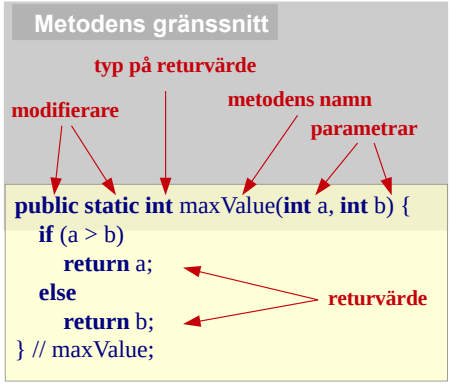
```
//Utseende på metoder som inte lämnar returvärde
modifierare void namn(parameterlista) {
    dataattribut och satser
}
```

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

exekveringsordning

```
. . .
int big = maxValue(value1, value2);
. . .
big = maxValue(big, value3);
. . .
```

```
public static int  maxValue(int a, int b) {
   if (a  > b)
      return a;
   else
      return b;
} //maxValue
```


Parameteröverföring

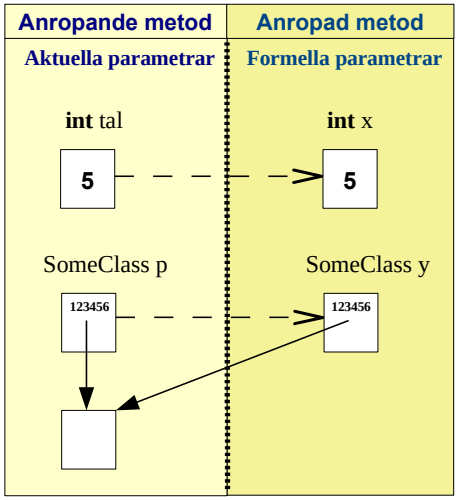
Alla primitiva datatyper och alla existerande klasser kan ges i parameterlistan och/eller som resultattyp.

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.

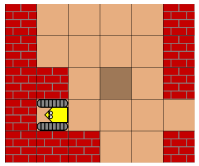
Värdet av den aktuella parametern p kopieras till den formella parametern y.

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

Laboration 2

I laboration 2 skall ni programmera en robot som modelleras av den givna klassen Robot.
En robot vistas i en enkel värld, och kan utföra några enkla operationer:



Följande operationer finns (samt några till):

void move() förflyttar sig ett steg framåt. Om roboten hamnar utanför världen eller i en mur fås ett exekveringsfel.

boolean frontIsClear() returnerar **true** om det är möjligt för roboten att göra move() utan att ett exekveringsfel erhålls, annars returnera **false**

void turnLeft() vrider sig 90° åt vänster

void makeLight() färgar rutan den står på till ljus. Om rutan redan är ljus fås ett exekveringsfel.

boolean onDark() returnerar **true** om roboten står på en mörk ruta, annars returneras **false**

int getDirection() returnerar robotens riktning. Roboten kan ha fyra riktningar: Robot.North, Robot.WEST, Robot.SOUTH och Robot.EAST.

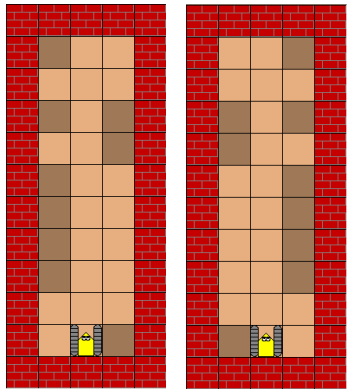
Syftet med laborationen är att ni bryta ner de uppgifter som roboten skall utföra i delproblem och utveckla kraftfullare abstraktioner än de operationer som roboten tillhandahåller.

Problem

I världen som skapas av klassen Swapper befinner sig roboten i ett scenario enligt den vänstra av de två figurerna bredvid. Roboten är placerad i början av en korridor och är riktad mot korridoren.

Vår uppgift är att implementer en metod swapAll som byter plats på färgerna på cellerna som finns på ömse sidor om korridoren (se högra bilden bredvid). Efter slutförd uppgift skall roboten återvända till sig ursprungliga startposition.

Vi skall bryta ner uppgiften i delproblem och utveckla kraftfullare abstraktioner (metoder) än de operationer som roboten tillhandahåller.



Före Efter

Klassen Swapper

```
public class Swapper {
   private Robot robot;
   public static void main(String[] args) {
      Swapper swapper = new Swapper();
      swapper.createEnviroment();
      swapper.swapAll();
   }//main

   public void createEnviroment() {
      RobotWorld world = RobotWorld.load("swap.txt");
      robot = new Robot(world.getNumRows() - 2, 2, Robot.NORTH, world);
      robot.setDelay(100);
   }//createEnviroment

   public void swapAll() {
      //todo
   }//swapAll
}//Swapper
```

Skapar världen som definieras i filen swap.txt

Skapar roboten och Placerar den I värden på en viss position och med en riktning

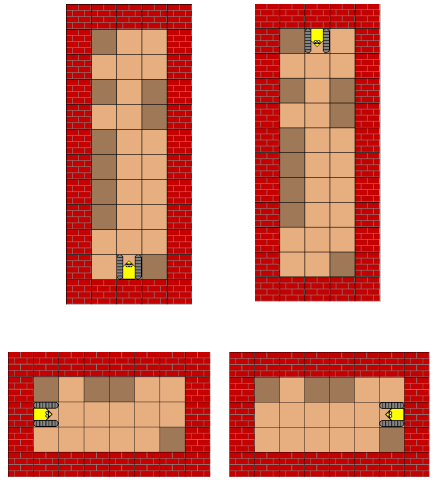
Sätter robotens rörelsehastighet

Problem

Roboten skall kunna utföra sin arbetsuppgift i en godtycklig värld med samma *principiella uppbyggnad* som världen i figurerna ovan.

När vi definierar en metod (abstraktion) är det viktigt att dokumentera vad metoden gör och vilka för- respektive eftervillkor som gäller (t.ex. vilken riktning roboten måste ha innan metoden anropas, och vilken riktning roboten får efter anropet). Det är också viktigt att välja lämpliga och beskrivande namn på metoderna.

När man löser ett problem är det sällan man finner den optimala lösningen direkt. När man väl har en lösning måste man reflektera över om det finns andra bättre lösningar. I det problem som vi har här, kan antalet operationer som roboten behöver utföra vara ett mått på hur bra lösningen är. Ju färre operationer desto bättre.



Lös problemet

Handfallen? Rådvill? Kommer inte igång?

- Gör en analys av uppgiftsformuleringen.
- Fokusera på vad som skall göras, inte på hur det skall göras! Finn abstraktioner!
- Förenkla problemet! Titta på en mindre instans av problemet!

Vi börjar med att specificera gränssnittet för metoden swapAll:

- Vad skall metoden göra?
- Vilka förvillkor gäller?
- Vilka eftervillkor skall gälla?

```
//Swapping colors on all across cells in the corridor.
//before: The robot is located at the beginning of the corridor, facing the corridor.
//after:    The robot has the same location and facing the same direction.
public void swapAll() {
      //todo
}//swapAll
```


Lös problemet

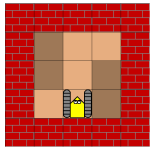
Skall vi byta färgerna på alla par av motstående celler i korridoren, måste vi kunna byta färgerna på ett par av motstående celler.

Vi inför en sådan abstraktion som vi kallar swapAcrossCells.

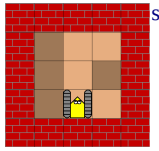
```
//Swapping colours of two across cells.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void swapAcrossCells()  {
    //todo
}//swapAcrossCells
```


Simulera ett enklet exempel:

1)

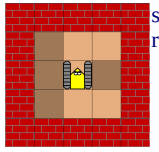


2)



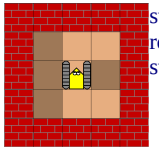
swapAcrossCells();

3)



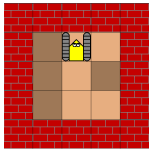
swapAcrossCells();
robot.move();

4)



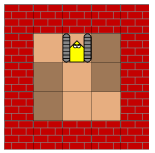
swapAcrossCells();
robot.move();
swapAcrossCells();

5)



swapAcrossCells();
robot.move();
swapAcrossCells();
robot.move();

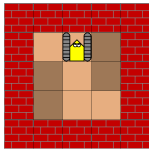
6)



swapAcrossCells();
robot.move();
swapAcrossCells();
robot.move();
swapAcrossCells();

Simulera ett enklet exempel:

6)



swapAcrossCells();
robot.move();
swapAcrossCells();
robot.move();
swapAcrossCells();

Ser du något mönster?

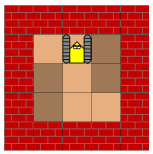
När roboten står på en position i korridoren byter den först färgen på motstående celler i korridoren, sedan går den fram en position om den inte står i slutet av korridoren.

Vi behöver en abstraktion isAtEndOfCorridor för att avgöra om roboten är i slutet av korridoren:

```
//returns true if the robot is at the end of the corridor, otherwise false
public boolean isAtEndOfCorridor()  {
      //todo
}//isAtEndOfCorridor
```

Simulera ett enklet exempel:

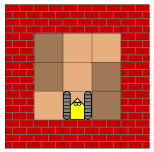
6)



```
boolean finished = false;
while (!finished) {
    swapAcrossCells();
    if (!atEndOfCorridor())
        robot.move();
    else
        finished = true;
}
```

Nu återstår att roboten skall återvända till sin ursprungliga startposition i början av korridoren. Vi behöver en abstraktion returnToStartPosition:

7)



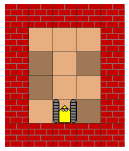
```
//before: The robot is at the end of the corridor, facing the wall.
//after:    The robot is at the beginning of the corridor, facing the corridor.
private void returnToStartPosition() {
        //todo
}//returnToStartPosition
```


Metoden swapAll

Metoden swapAll får följande implementation:

```
//Swapping colors on all across cells in the corridor.
//before: The robot is located at the beginning of the corridor, facing the corridor.
//after:     The robot has the same location and facing the same direction.
public void swapAll() {
    boolean finished = false;
    while (!finished) {
        swapAcrossCells();
        if (!isAtEndOfCorridor())
            robot.move();
        else
            finished = true;
    }
    returnToStartPosition();
}//swapAll
```


Metoden swapAcrossCells



När behöver färgerna på cellerna bytas?

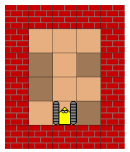
När cellerna har olika färg!

Vi inför en abstraktion isLeftCellDark för att ta reda på om färgen på den vänstra cellen är mörk, och en abstraktion isRightCellDark för att ta reda på om färgen på den högra cellen är mörk.

```
//return:  true if the cell on left side of the robot is dark, otherwise false.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private boolean isLeftCellDark() {
    //todo
}//isLeftCellDark
```

```
//return:  true if the cell on right side of the robot is dark, otherwise false.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private boolean isRightCellDark() {
    //todo
}//isRightCellDark
```


Metoden swapAcrossCells



Färgerna på de motstående celler i korridoren behöver bytas om

isLeftCellDark() != isRightCellDark()

För att byta färgerna på cellerna inför vi en abstraktion changeColorsOfAcrossCells:

```
//Change color of the cell on left side and of the cell on right side.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorsOfAcrossCells() {
    //todo
}//changeColorsOfAcrossCells
```

Metoden swapAcrossCells får då följande implementation:

```
//Swapping colors of two across cells.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void swapAcrossCells() {
        if (isLeftCellDark() != isRightCellDark())
            changeColorsOfAcrossCells();
}//swapAcrossCells
```

Metoden isLeftCellDark

För att ta reda på om cellen till vänster är mörk måste roboten gå in i cellen, kontrollera cellens färg och gå tillbaks till sin ursprungliga position. Vi inför abstraktion turnAround som vrider roboten 180 grader:

```
//before: None.
//after:    The robot is facing the opposite direction.
private void turnAround() {
    //todo
}//turnAround
```

Metoden isLeftCellDark får då följande implementation:

```
//Returns true if the cell on left side of the robot is dark, otherwise false.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private boolean isLeftCellDark() {
        robot.turnLeft();
        robot.move();
        boolean isDark = robot.onDark();
        turnAround();
        robot.move();
        robot.turnLeft();
        return isDark;
}// isLeftCellDark
```


Metoden isRightCellDark

För att svänga roboten till höger inför abstraktion turnRigth som vrider roboten 90 grader åt höger:

```
//before: None.
//after:    The robot has turned 90 degree to right.
public void turnRight() {
    //todo
}//turnRight
```

Metoden isRightCellDark får då följande implementation:

```
//Returns true if the cell on right side of the robot is dark, otherwise false.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private boolean isRightCellDark() {
        turnRight();
        robot.move();
        boolean isDark = robot.onDark();
        turnAround();
        robot.move();
        turnRight();
        return isDark;
}//isRightCellDark
```


Metoden turnAround och turnRight

Metoden turnAround innebär att roboten skall svänga två gånger åt väster:

```
//before: None.
//after:    The robot is facing the opposite direction.
private void turnAround() {
   robot.turnLeft();
   robot.turnLeft();
}//turnAround
```

Metoden turnRight innebär att roboten skall svänga tre gånger åt väster. Vi nytjar metoden turnAround i implementationen:

```
//before: None.
//after: The robot has turned 90 degree to right.
public void turnRight() {
    turnAround();
    robot.turnLeft();
}//turnRight
```


Metoden changeColors

Att ändra färgerna på motstående celler innebär att den vänstra cellen skall byta färg och att den högra cellen skall byta färg. Vi inför abstraktionerna changeColorOfLeftCell och changeColorOfRightCell:

```
//Change color of the cell on left side of the robot.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorOfLeftCell() {
    //todo
}//changeColorOfLeftCell
```

```
//Change color of the cell on right side of the robot.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorOfRightCell() {
    //todo
}//changeColorOfRightCell
```

Metoden changeColorsOfAcrossCells

Implementationen av metoden changeColorsOfAcrossCells får följande utseende:

```
//Change colors of the cells on left side and on right side of the robot.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorsOfAcrossCells() {
        changeColorOfLeftCell();
        changeColorOfRightCell();
}//changeColorsOfAcrossCells
```


Metoderna changeColorOfLeftCell och changeColorOfRightCell

För att byta färgen på en cell inför vi abstraktionen switchColor:

```
//Switch color of the cell.
//before: None.
//after:    If the cell, the robot is located at, was dark it has become light,
//          and if the cell was light it has become dark.
private void switchColor() {
        //todo
}//switchColor
```


Metoderna changeColorOfLeftCell och changeColorOfRightCell

Implementationerna av changeColorOfLeftCell och changeColorOfRightCell får följande utseende:

```
//Change color of the cell on left side of the robot.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorOfLeftCell() {
        robot.turnLeft();
        robot.move();
        switchColor();
        turnAround();
        robot.move();
        robot.turnLeft();
}//changeColorOfLeftCell
```

```
//Change color of the cell on right side of the robot.
//before: The robot is in the corridor facing the corridor.
//after:    The robot is in the corridor, facing the same direction.
private void changeColorOfRightCell() {
        turnRight();
        robot.move();
        switchColor();
        turnAround();
        robot.move();
        turnRight();
}//changeColorOfRightCell
```


Metoden switchColor

Implementationen av switchColor får följande utseende:

```
//Switch color of the cell.
//before: None.
//after:    If the cell, the robot is located at, was dark it has become light,
//          and if the cell was light it has become dark.
private void switchColor() {
        if (robot.onDark())
                robot.makeLight();
        else
                robot.makeDark();
}//switchColor
```

Metoden returnToStartPosition

För att återvända till sin startposition måste roboten vända sig om, gå till andra ändan av korridoren och vända sig om igen. Vi inför abstraktionen goToEndOfCorridor för att förflytta roboten från en given position till slutet av korridoren:

```
//before: The robot is in the corridor, facing the cooridor.
//after:    The robot is at the end of the corridor, facing the wall.
private void goToEndOfCorridor() {
   //todo
}//goToEndOfCorridor
```

Implementationen av returnToStartPosition får då följande utseende:

```
//before: The robot is at end of the corridor, facing the wall.
//after:    The robot is at beginning of the corridor, facing the corridor.
private void returnToStartPosition() {
        turnAround();
        goToEndOfCorridor();
        turnAround();
}//returnToStartPosition
```


Metoden is atAndOfCorridor och goToOfCorridor

Roboten är i slutet av korridoren om den är riktad mot en vägg (eller nått slutet av världen). Implementationen av metoden isAtEndOfCorridor får följande utseende:

```
//returns true if the robot is at the end of the corridor, otherwise false
public boolean isAtEndOfCorridor()  {
        return !robot.frontIsClear();
}//isAtEndOfCorridor
```

Att gå till slutet av korridoren innebär att förflytta sig längs korridoren tills slutet av korridoren har nåtts. Implementationen av goToEndOfCorridor får följande utseende:

```
//before: The robot is in the corridor, facing the cooridor.
//after:    The robot is at the end of the corridor, facing the wall.
private void goToEndOfCorridor() {
        while (!isAtEndOfCorridor()) {
                robot.move();
        }
}//goToEndOfCorridor
```