Arbetssätt och Programkonstruktion

TDA548/Joachim von Hacht

Arbetssätt

Ett genomtänkt arbetssätt skall hjälpa er att klara kursens laborationer (...och följande kursers)

- Genom att arbeta på ett visst sätt ökar vi vår förmåga att lösa problem.
- Generellt: Behärska komplexitet.

Minsta Steget

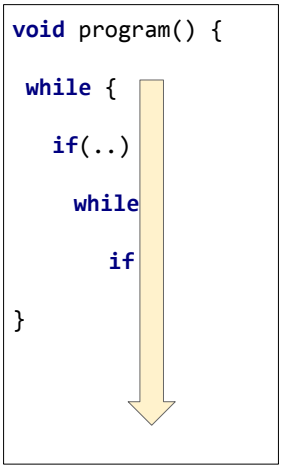
```
void program() {
   // Very small basic step
   out.println("Program started");

}
```

3

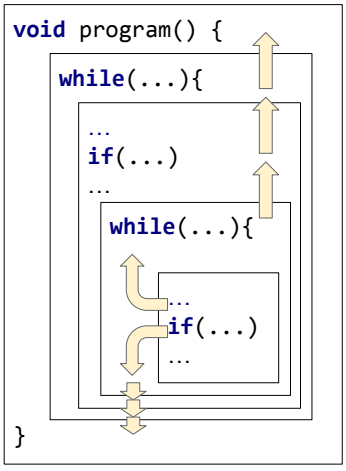
När vi skriver ett program (eller håller på med en del av) kan man inte skriva allt på en gång!

- Man börjar med att försöka hitta ett minsta steg som för oss närmare lösningen
- Kodar detta och kontrollerar om det fungerar … d.v.s. kör (eller testar) programmet.
- När det fungerar söker vi nästa minsta steg på samma sätt
- Vi ser till att alltid ha en körbar kärna som vi bygger på steg för steg.
- Bortse inledningsvis från alla specialfall, skriv “normalkoden” först
    - … MEN, notera specialfallen, … ofta där felen (buggarna) finns.

Inifrån Ut



Uppifrån och ner

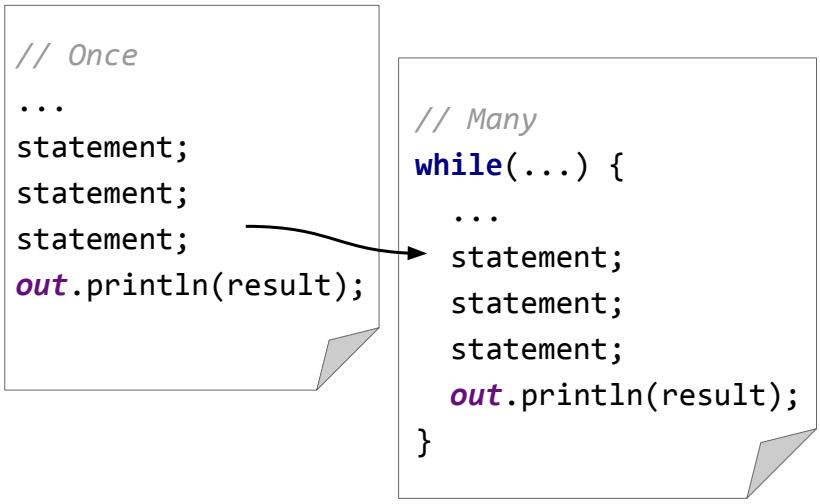


Inifrån ut

När man jobbar med minsta steget tekniken arbetar man ofta “inifrån-ut”

- Inifrån-ut innebär att man inte skriver programmet “rad för rad” uppifrån och ned.
- Istället börjar man “inifrån”
- När man fått till något minst steg utökar man vid behov programmet genom att lägga till ev. if/while etc. “runt”
- När detta fungerar utökar man med ev. nästa “lager” o.s.v.

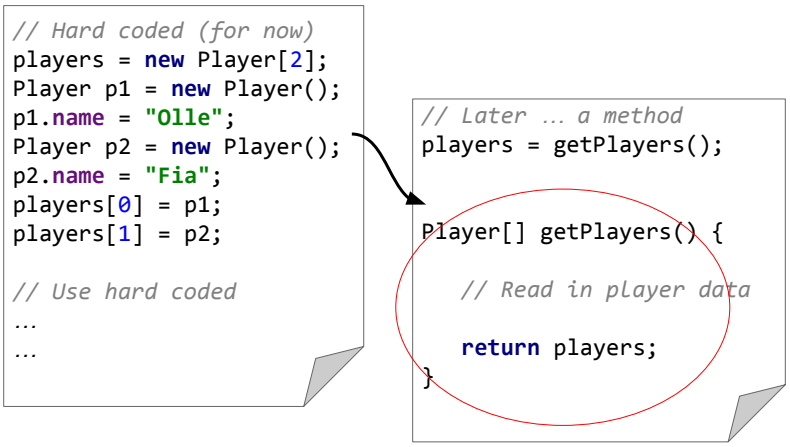
En Gång, flera Gånger



När man använder minsta steget och inifrån-ut:

- Om man kan göra något en gång, … är det lätt att göra det flera gånger … lägg en loop runt!
- Börja med att göra något en gång först!

Hårdkodad data



Vi vissa steg kan det behövas data som man inte har än. Om så:

- Hårdkoda data, d.v.s. skriv dit något tillfälligt (något enkelt). Bara så att vi kan fortsätta med nästa steg.
- Senare ersätts det hårdkodade med inläst och/eller beräknad data.

Kommentera och *//TODO*

```
/*coin = new ImageIcon(ImageIO
       .read(new
File("src/exercises/optional/gold_coin_single.png")))
       .getImage();*/
coin = new ImageIcon(this.getClass() // TODO better use Image?

       .getResource("gold_coin_single.png"))
       .getImage();
// getAudioInputStream() also accepts a File or InputStream
//AudioInputStream ais = AudioSystem.
// getAudioInputStream(new
File("src/exercises/optional/atari.wav"));
AudioInputStream ais = AudioSystem
     .getAudioInputStream(this.getClass()
     .getResourceAsStream("atari.wav"));
```

7

Under utvecklingen undviker man att ta bort kod, .. kanske koden inte var så dum ändå …!

- Använd kommentarer istället för att ta bort kod!
- Lätt att få tillbaka den gamla koden om den visade sig vara bra!
- När programmet är färdig: STÄDA UPP! Ta bort kommenterade stycken, indentera, fixa bättre namn,,...

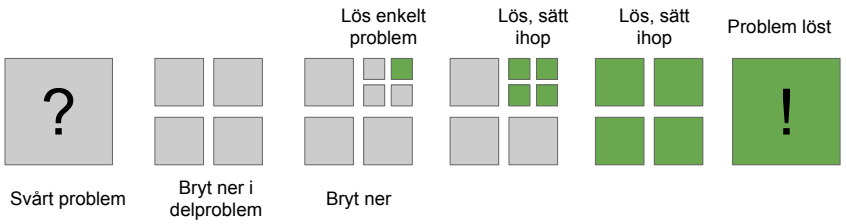
Ofta stöter man på saker som man borde fixa till (när man egentligen håller på med en annan sak).

- Ta för vana att lägga in TODO noteringar (sökbara i IntelliJ)

I bilden

- Gammal kod bortkommenterad (tills vi är säkra på att den nya är bättre, annars, kommentera ut den nya och avkommentera den gamla)
- TODO-notering för att senare kolla om Image är bättre än ImageIcon

Bryta ner Problem



8

Normalt är ett problem (program) för stort för att direkt kunna kodas med minsta steget metodiken.

- Vi måste dela upp problemet.

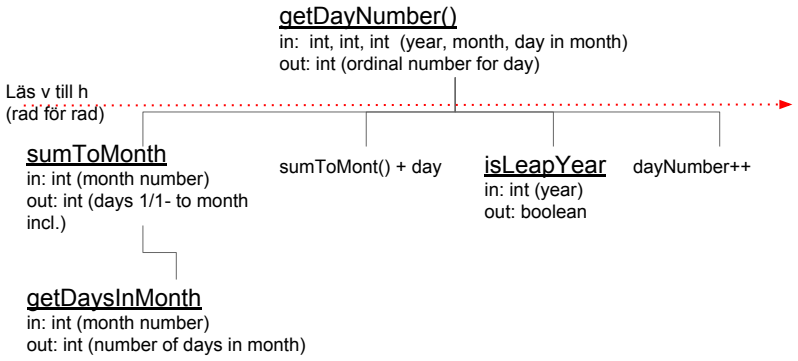
Ett klassiskt angreppssätt om man har ett stort/svårt problem.

- Bryt ner i mindre/enklare problem!
- Lös de enklare problemen (eller dela dessa ytterligare)
- Sätt ihop alla lösningar av delproblemen till en lösning för hela problemet.

Konkret gör vi detta genom att använda funktionell nedbrytning.

Funktionell Nedbrytning

Problem: Givet årtal och datum, beräkna ordningsnumret för dagen detta år (Exempel: 25/4, 2018 är dag 115 detta år)



9

Funktionell nedbrytning (functional decomposition) innebär att man:

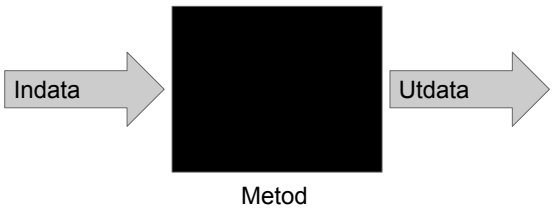
- Antar att man har en metod som löser hela problemet (i bilden: getDayNumber)
- Börjar skriva denna … när man stöter på ett nytt problem antar man att man har en metod som löser detta (sumToMonth, isLeapYear).
- O.s.v… tills metoderna man behöver är triviala t.ex. getDaysInMonth.
  - Implementera de triviala metoderna.
  - Implementera m.h.a. dessa metoderna "högre upp"
  - Genom att kombinera små enkla metoder har man löst ett stort problem. Klart!
- För att illustrera kan man rita ett diagram enligt bilden. Överst ritar man den första metoden man antog, därefter en ny nivå för varje nedbruten method

Funktionell nedbrytning är en top-down strategi. Man börjar med helheten och bryter ner i enklare delar.

Genom att använda funktionell nedbrytning (plus anteckningar/skisser) får vi en struktur på programmet.

- Ha alltid papper och penna uppe!

Funktionell Abstraktion



10

Under arbetet med funktionell nedbrytning söker vi metoder vi behöver.

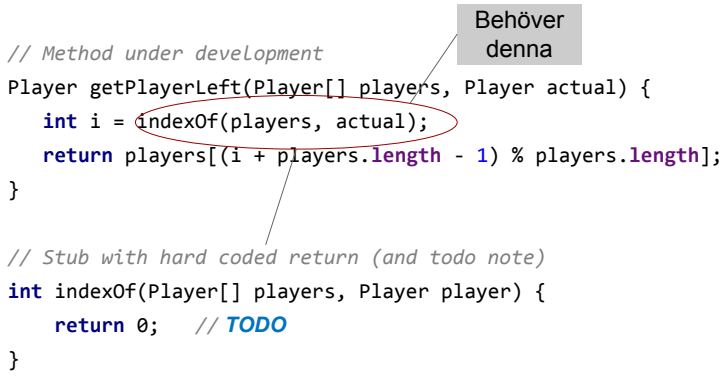
För att underlätta sökandet använder man **funktionell abstraktion**, en tankeform där man bara fokuserar på:

- Indata (vad har jag?)
- Utdata (vad vill jag ha?)
- … detta för att slippa alla detaljer om hur det skall gå till …
- Sikta på vad som skall göras inte hur!
    - Vi måste veta vad som skall göras innan vi börjar fundera på hur det skall göras.

Process för att skriva en metod:

1. Namnge metoden (med ett bra namn som säger vad den gör, ett verb är ofta inblandat).
    a. En metod skall vara bra på en sak, kan du inte hitta ett bra namn kan det bero på att metoden gör för mycket. Isf bryt ner den i mindre metoder.
2. Vilken data har du? Ange parametrar.
3. Vad vill du ha? Skriv metodens returtyp m.h.a. detta.
4. Skriv klart metoden m.h.a. minsta steget (och testa, se senare slide)

Metodstubbar



Då man arbetar med funktionell nedbrytning händer det t.ex. att man upptäcker flera metoder samtidigt.

- Eftersom vi bara kan implementera en metod i taget så skriver man "stubbar" för de övriga

En **metodstubbe** är en "tom" metod.

- Genom att använda stubbar kan vi gör anrop till metoder som inte är färdiga d.v.s programmet kompilerar och går att köra (men resultatet blir inte rätt)
- Genom att använda stubbar kan vi få ett körbart program.
  - Vid behov hårdkoda returdata, t.ex. return 0, return null, return "", o.s.v..

Testning

```
// Simple test for dices
int d = rollDice();
out.println(1 <= d && d <= 6);   // true?

// Test to get player to the left of first (there are 3 players)
Player[] players = {new Player("0lle", 3), new Player ...};
out.println(getPlayerLeft(players, players[0]) == players[2]);  // true?


// Check distribution of game chips given a known distribution and a result
String[] res = {"L", "C", "R"};  // The known result
actual = players[1];
distributeChips(res, players, actual); // Initially all have 3 chips
out.println(players[0].chips == 4 &&
            players[1].chips == 0 && players[2].chips == 4); // true?
```

12

Då vi använder funktionell nedbrytning kommer programmet att bestå av ett antal samverkande metoder.

- Att bara sätta ihop dessa utan att veta om de fungerar är dåligt!
  - För att få helheten att fungera måste delarna fungera!

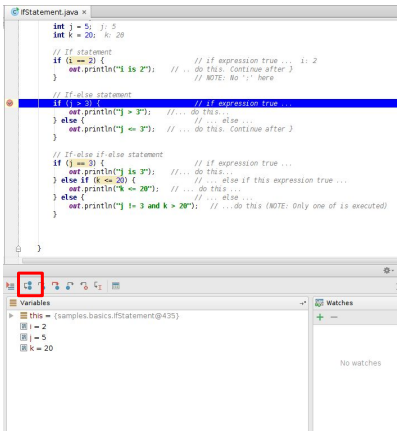
Vår strategi är följande:

- Så fort vi implementerat en metod så testar vi den.
- IO-metoder testar vi, vid behov manuellt, vi provkör helt enkelt
  - IO metoder är (skall vara) enkla.
- För logikmetoder automatiserar vi testerna genom att skriva ut (out.println) ett uttryck som skall ge värdet true
  - I uttrycket gör vi en jämförelse mellan resultatet av ett metodanrop och ett förväntat korrekt värde (som vi måste veta)
    - Här krävs viss kreativitet för att komma på vilka jämförelser vi vill göra
    - Testerna måste vara så enkla som möjligt, vi vill inte introducera nya fel i själva testerna.
  - Vi skriver alla tester i en egen metod med namnet test() eller …
    - … senare i en egen klass men namnet Test.
  - Vi behåller alltid alla tester, … när som helst kan vi köra dessa igen!

I senare kurser får du lära dig att använda speciella ""testramverk".

Felsökning (debug)

Avlusare (debugger)



Utskrift av värde

```
out.print(someValue);
```

14

En stor del av programmeringsarbetet består av felsökning. Det finns flera alternativ

- Använda out.println() och skriv ut värden
  - Kan var en bra metod för upprepad inspektion
    - Utskriften sker varje gång programmet (loopen) körs
  - Nackdelar:
    - Blir det för många utskrifter kan man till slut inte hänga med
    - Utskrifterna måste tas bort
- Använda en avlusare (Debugger)
  - En avlusare är ett program som kan köra ett annat (ditt) program sats för sats
  - Finns inbyggd i IntelliJ

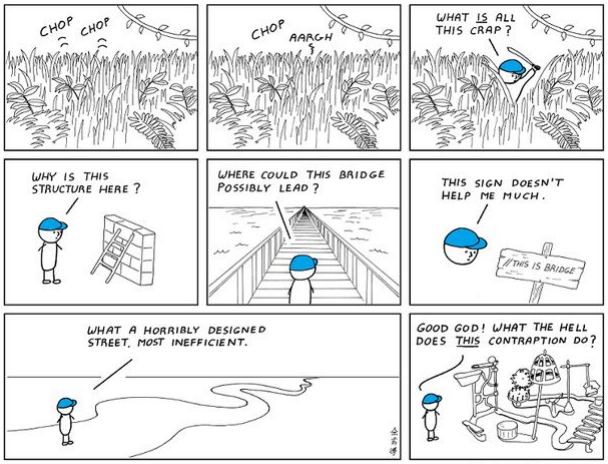
Avlusning (procedur)

- Klicka i vänstermarginalen för att få en brytpunkt (röd prick ovan).
  - En brytpunkt innebär att programmet kommer att stanna på markerad rad.
  - Klicka igen om du vill ta bort..
- Högerklicka i kodfönstret och välj Debug …
- Avlusaren startar och kör programmet fram till brytpunkten. Där stannar det.

- Därefter kan du köra sats för sats genom att klicka Step Over (röd fyrkant i bilden)
- Den blå raden skall hoppa en sats då du klickar
- Du kan hela tiden inspektera variabelvärden i det nedre fönstret
- Avsluta genom att klicka röd fyrkant ner till vänster (visas ej)
- Hakar något upp sig … börja om

OBS! Att felsöka/debuga i tester är mycket effektivt:
- Vi kan hårdkoda in data
- Vi arbetar med isolerade delar av programmet.

Programkonstruktion



I hate reading other people's code.

14

För att skapa högkvalitativ mjukvara finns ett stort antal regler eller rekommendationer för hur program skall konstrueras.

- I bilden: Om man upplever koden som i serien så har man definitivt problem med konstruktionen.
- Software construction.

Best Practices

```
if(!scene6TF1.getText().isEmpty() && scene6TF1.getText().length() < 21 && Pattern.matches("[\306\330\305\346\370\34
    if (!scene6TF2.getText().isEmpty() && scene6TF2.getText().length() < 51 && Pattern.matches("[\306\330\305\346\3
        if (!scene6TF3.getText().isEmpty() && scene6TF3.getText().length() < 81 && Pattern.matches("[\306\330\305\3
            if (!scene6TF4.getText().isEmpty() && scene6TF4.getText().length() == 8 && Pattern.matches("[0-9]+", sc
                if (!scene6TF5.getText().isEmpty() && scene6TF5.getText().length() == 4 && Pattern.matches("[0-9]+",
                    if (!scene6TF6.getText().isEmpty() && scene6TF6.getText().length() == 10 && Pattern.matches("[0
                        if(scene6CB1.getValue() != null) {
                            if(scene6CB2.getValue() != null)
                            {
                                System.out.println("Tillykke, alle felter er godkendt!");
                                // Opret medarbejder - det vil sige foretag et SQL statement der indsætter de forskl

                            } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Løntrin er ikke valgt i drop-down menu
                        } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Stilling er ikke valgt i drop-down menuen.'
                    } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Kontonummeret indtastet forkert. Feltet må kun
                } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Registreringsnummer indtastet forkert. Feltet må ku
            } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Telefonnumer indtastet forkert. Feltet må kun indeholde
        } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Adresse indtastet forkert. Feltet må kun indeholde bogstave
    } else { FejlMeddelelse.visFejlMeddelelse("Fejl: Efternavn indtastet forkert. Feltet må kun indeholde bogstaver,
} else { FejlMeddelelse.visFejlMeddelelse("Fejl: Fornavn indtastet forkert. Feltet må kun indeholde bogstaver, mel
});
```

Violating best practices!

17

Med best practices (beprövad erfarenhet) avses ett stort antal informella regler som ganska stor sannolikhet leder till bättre kodkvalitet. T.ex:

Namn:

- Beskrivande, lagom långa namn
- Standard förkortningar point = pts, m.fl.
- Singular/Pluralis
- Metoder inleds ofta med verb
- Booleska variabler/metoder namnges som ja/nej frågor

Arrayer:

- Arrayer används framförallt då man inte vill att strukturen skall ändras, vi kan ta bort element men index finns kvar.
- Finns inte det kravet föredra någon samling (t.ex. List)

Metoder

- En metod skall vara expert på en sak!
- Hellre många små specialiserade metoder än några stora
- “one-liners” metoder är helt ok.
- Skilj IO och logik
- Undvik utparametrar
- Undvik att returnera null (skicka t.ex. en tom array/List)

Klasser:

- En klass skall fånga ett koncept
- En klass skall ha ett ansvarsområde
- … på samma sätt som för metoder
- … annars blir de svåra att (åter)använda, felsöka, … det blir rörigt!
- Bättre att kombinera flera (små) tydliga klasser
- Använd informationsgömning!

Code Smells

```
// Bad redundant code!
if ( … ) {
    …
    dices = 1;
} else {
    …
    dices = 1;
}
```

```
// Non redundant
if ( …  ) {
   …
} else {
    …
}
dices = 1;
```

16

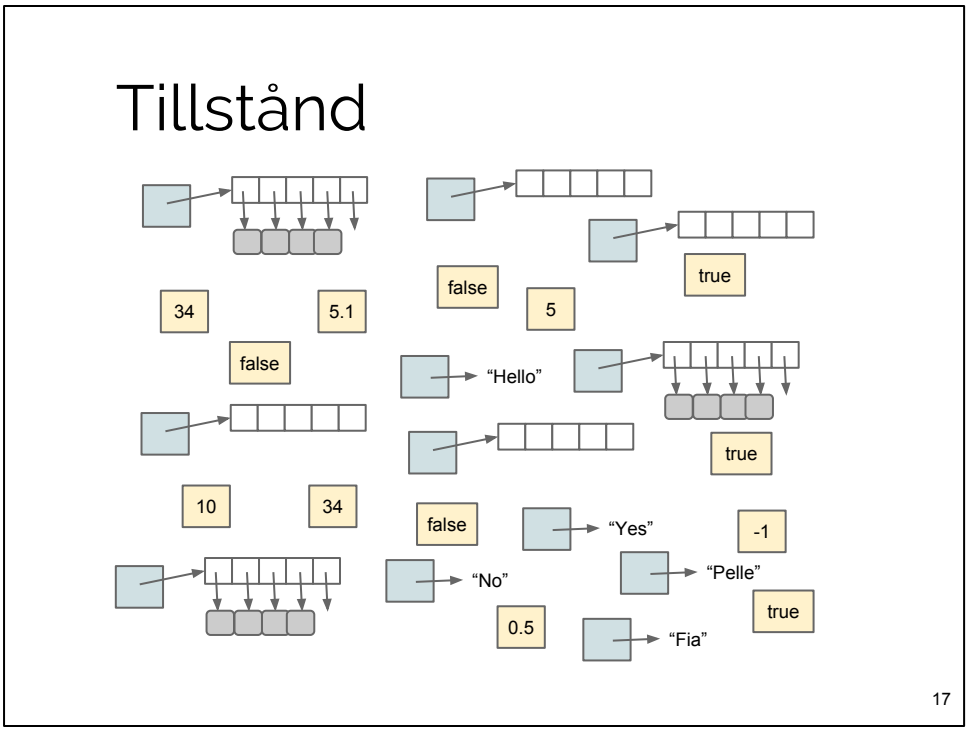
Code smells är tecken på att koden inte är bra (inversen till best practices)

Exemplet i bilden:
Redundant eller duplicerad kod är en code smell

- Mer (onödig) kod -> mer chanser till fel
- Kod måste hållas i synk, ändringar måste göras på flera ställen.
- Idealet är att allt finns/görs på exakt ett ställe i programmet.

Hur åtgärda:

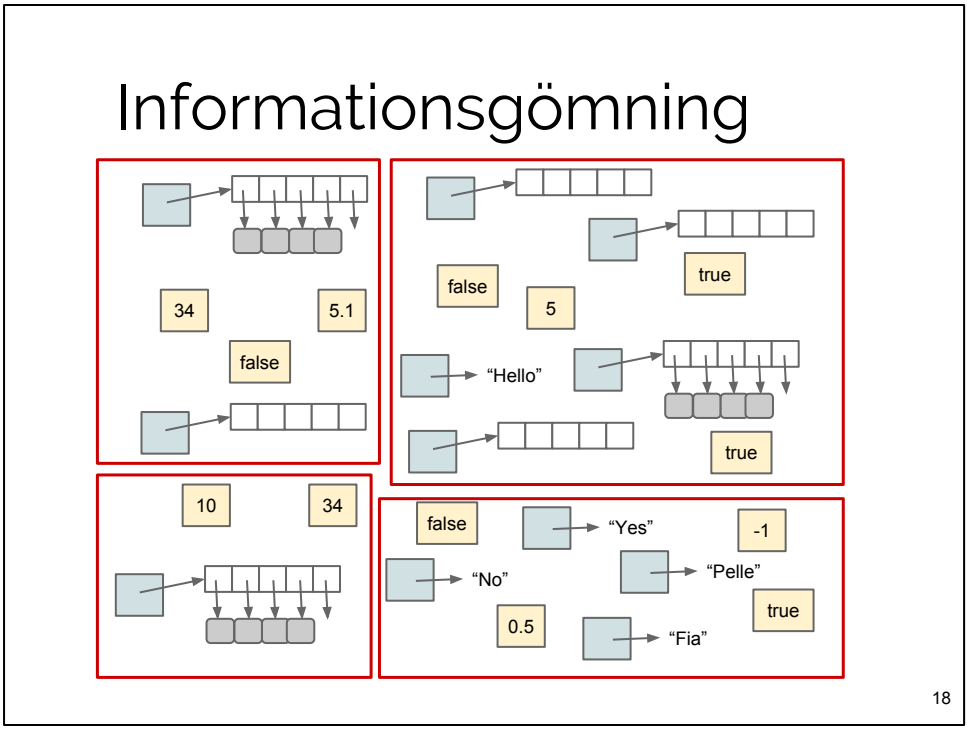
- När ni fått till ett fungerande kodavsnitt gör en "**code review**"!
- Känns någon code smell … t.ex. görs något i onödan, eller görs samma sak på flera ställen?



Ett fundamentalt problem inom imperativ programmering är att behärska tillståndet!

Tillstånd (state) mängden av alla värden för alla instansvariabler i programmet vid en viss tidpunkt under exekveringen

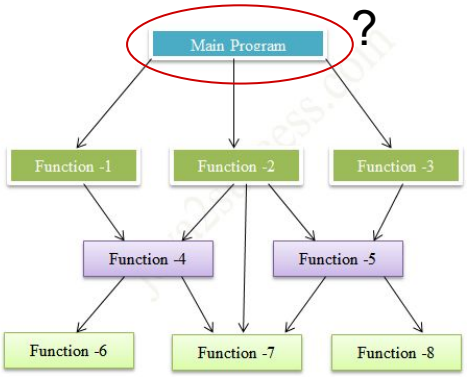
- Lokala variabler räknas ej (de kommer och går …)
- Om allt fungera som tänkt innehåller variablerna korrekta värden, programmet befinner sig i ett giltigt tillstånd … om EJ
- … har vi ett ogiltigt tillstånd. Något är fel
- Att naivt försöka behärska tillståndet övergår mänsklig förmåga …
  - … vi måste utveckla tekniker för detta
  - Innebär t.ex. att vi alltid föredrar lokala variabler (eftersom de inte ingår i tillståndet)
  - Vi försöker också konsekvent att minska synlighetsområdet för variabler.
  - Om möjligt kan vi använda icke-muterande objekt (de har tillstånd men tillståndet ändras inte).



Ett annat sätt att behärska tillståndet är att dela upp det totala tillståndet i mindre deltillstånd och på så sätt lättare kunna hålla dessa giltiga

- Kallas **informationsgömning** (information hiding)
- Konkret gör vi detta genom att dela upp tillståndet på olika klasser med private instansvariabler.
- Se Klasser

Procedurell Programmering



När man skapar ett program genom att använda funktionell nedbrytning, kallas detta procedurell programmering

- Programmet är uppbyggt av procedurer (= metoder/funktioner)

Eftersom man använder en top down strategi måste vi hitta top-metoden!

- Inte alltid så lätt .. .

Problem med Procedurell Programmering

Skriv ett procedurellt program som hanterar situationen nedan!

- Hmm, vilken är top metoden?

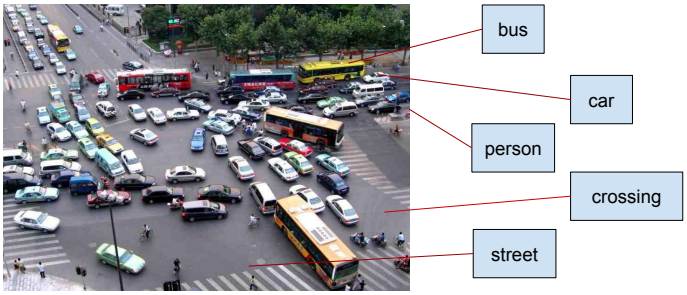
Vissa problem har visat sig svåra att arbeta med genom att enbart använda procedurell programmering.

- (Se bild) Vilken är top-metoden?
- Var skall vi börja???

Objektorientering

Skriv ett program som hanterar situationen nedan!

- Identifiera inblandade objekt!



Som en reaktion på problemen med procedurell programmering har objektorienterad programmering vuxit fram.

För att lösa problemet skapar man en objektmodell av detta (en OO-modell)

- Man försöker hitta objekt som finns i problemet (vilket är ofta är naturlig för oss människor)
- Försöker avgöra vilka egenskaper och beteenden objekten har.
- Man försöker se hur de samverkar.

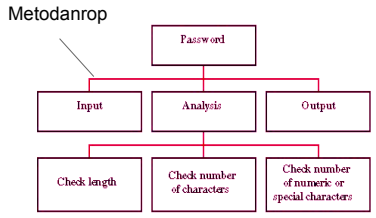
Därefter översätts modellen till klasser, instansvariabler och metoder.

Objektorientering innebär att man jobbar **bottom-up**

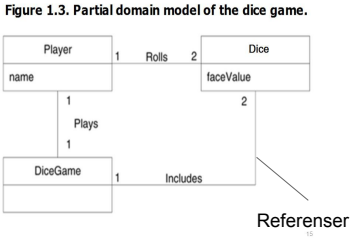
- D.v.s. vi börjar med delarna (objekten/klasserna).
- Precis tvärtemot procedurell programmering.

Eftersom OO-programmering också använder metoder, så gäller fortfarande funktionell nedbrytning och abstraktion (och minsta steget) då vi implementerar metoder.

Programstruktur



Procedurellt program



Objektorienterat program

Funktionell nedbrytning ger programmet en relativt enkel struktur att greppa.

- Strukturen förändras inte vid körning.
- Strukturen bygger på metodanrop

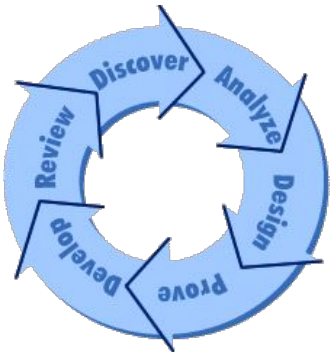
Objektorienterade program har en mer komplicerade struktur.

- Programmet är en väv (en graf) av objekt.
- Strukturen bygger på referenser mellan objekt.
- Grafen ändras under körning, objekt kommer och går, referenser byts o.s.v.

Ett av de stora problemen med objektorientering är att man lätt kan skapa en “object soup”, d.v.s. en obegriplig struktur av objekt och referenser.

- För att undvika detta måste man lära sig om objektorienterad design (i senare kurser)

Objektorienterad Metodik



I denna kurs är klasserna för det mesta givna.

- I verkligheten gör man en objektorienterad analys för att hitta klasser.

En metod för att skissa fram en objektmodell (trial and error)

1. Ange troliga attribut (instansvariabler) för klasserna.
2. Metoder som använder attributen skall placeras i klasserna, så att vi inte behöver skicka runt en massa data.
3. Utgå från användarens kommandon och skissa anropskedjor m.h.a. metoderna. Man kan se det som att den funktionella nedbrytningen är utspridd över flera objekt. Varje objekt bidrar med sin del.
    - Koppla ihop objekt som behöver varann m.h.a. konstruktorn. Viktigt för att göra det möjligt att testa klasser separat.
4. Om duplicerad data upptäcks, faktorisera ut till (abstrakta) basklasser (arv).
    - Flytta ev. gemensamma metoder till basklassen.
5. Testa klasserna, d.v.s. icke-triviala metoder i klasserna.