

# Datorteknik TSEA82 + TSEA57

## Fö4

Strukturerad programmering med JSP

# Datorteknik Fö3 : Agenda

- Lägeskoll
- Strukturerad programmering – JSP
- Kodstil
- Lab 1
- Tid för frågor

# Lägeskoll

# Var är vi?

V13 V14 V15 V16 V17 V18 V19 V20 V21

|     |     |     |     |     |     |     |     |       |
|-----|-----|-----|-----|-----|-----|-----|-----|-------|
| Fö1 | Fö4 | Fö6 | Fö7 | Fö8 | Fö9 | Le  |     | TPVT2 |
| Fö2 | La0 | La1 | La2 |     | La3 | La4 | La5 | LAX!  |
| Fö3 | Fö5 |     |     |     |     |     |     |       |

# Strukturerad programmering

*Algorithms + Data structures = Programs*





# Spaghetti code

Flödesdiagram är väl bra, eller?

Överblick?

Var börjar det?

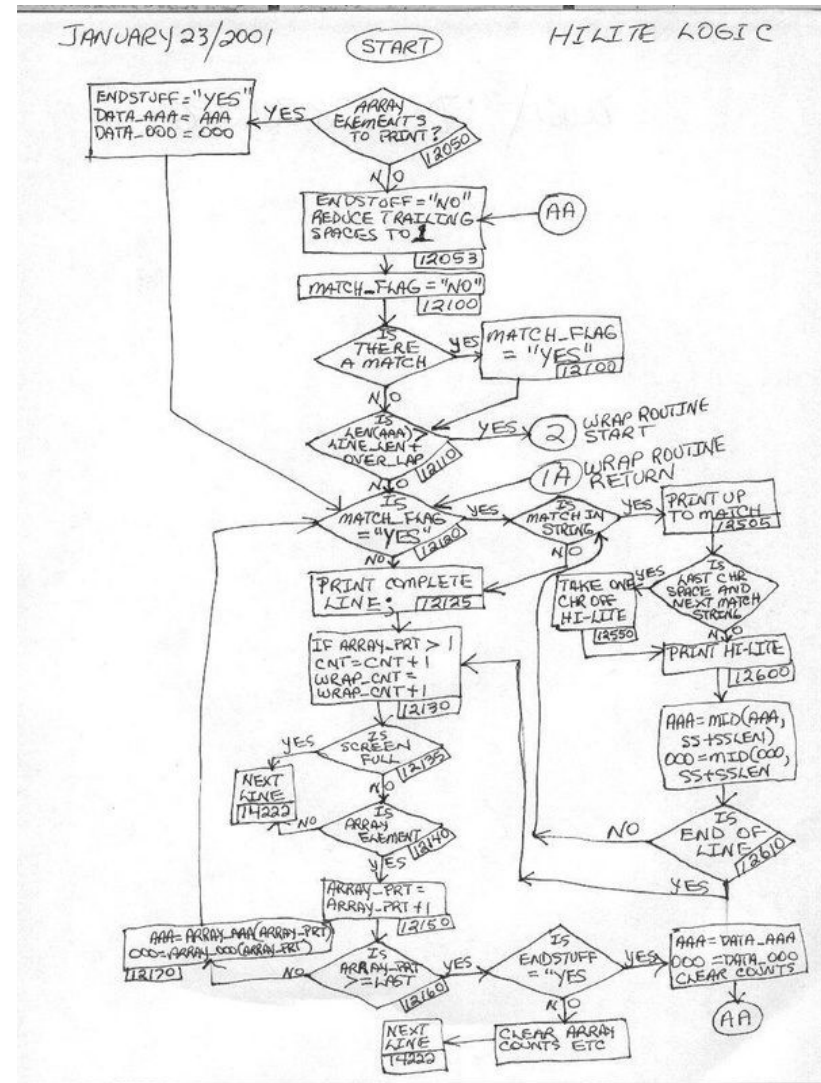
Var slutar det?

Vilka är programmets beståndsdelar?

Kan man återanvända kod som upprepar sig?

Går det ens att se var koden upprepar sig?

Det finns ingen STRUKTUR!!





## Strukturerad programmering

Att kunna ett programspråk  
är **INTE** detsamma  
som att kunna programmera!

[Jfr : Trafikregler vs Bilkörning]

# JSP – Jackson Structured Programming

## JSP-princip

- Data kan beskrivas i **strukturdiagram**
- Programmet ska följa datat.

Algorithms +  
Datastructures =  
Programs

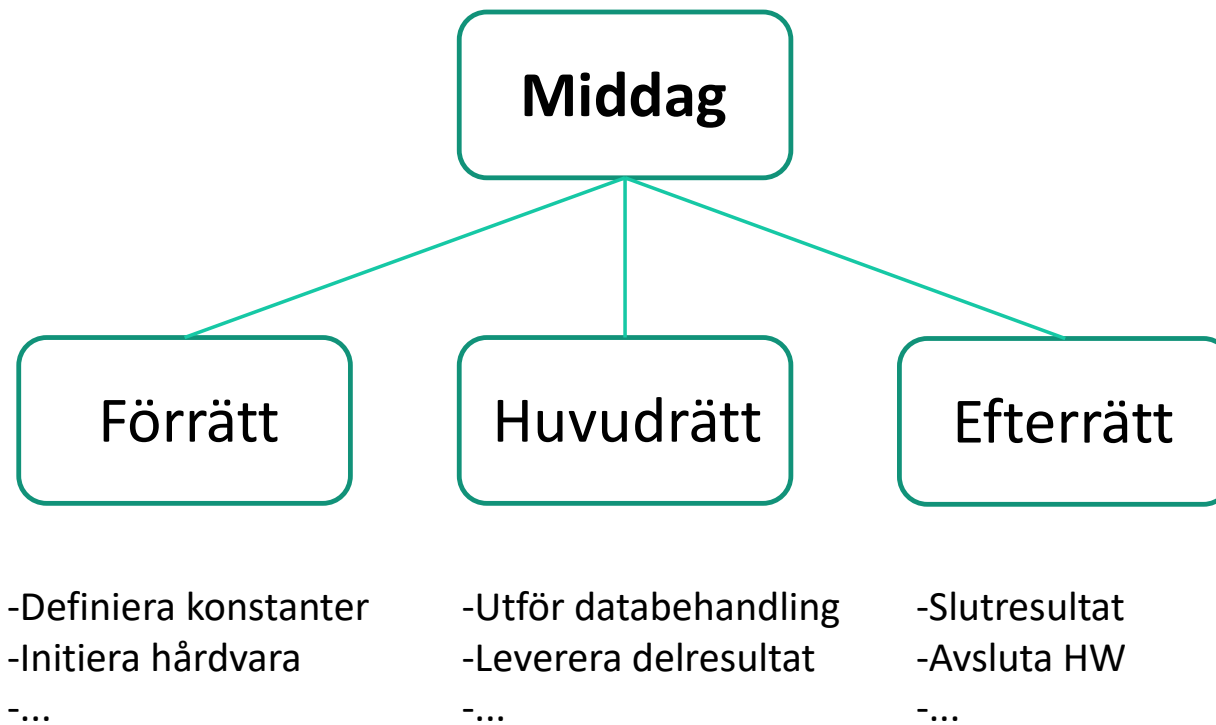
# JSP : Strukturdiagram Byggblock

Sekvens

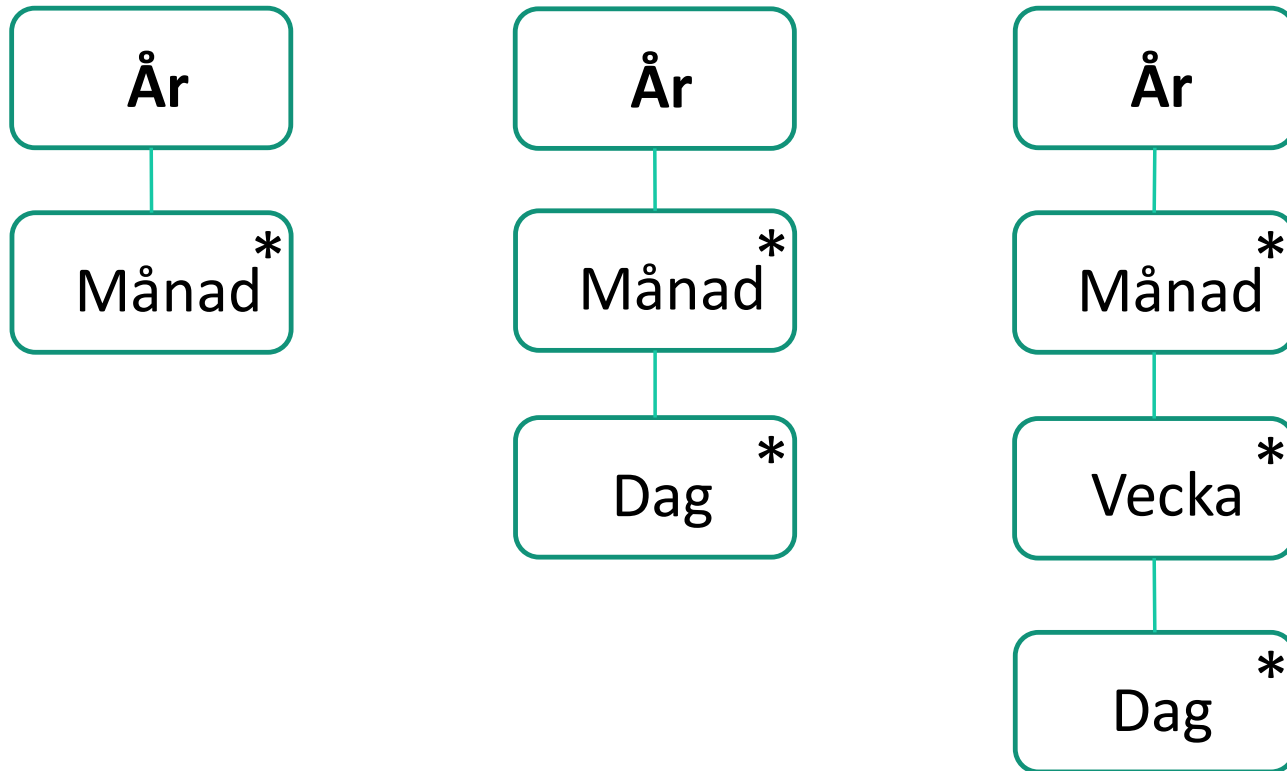
Iteration \*

Selektion ○

## JSP : Sekvens : först det ena, sen det andra och sen ...

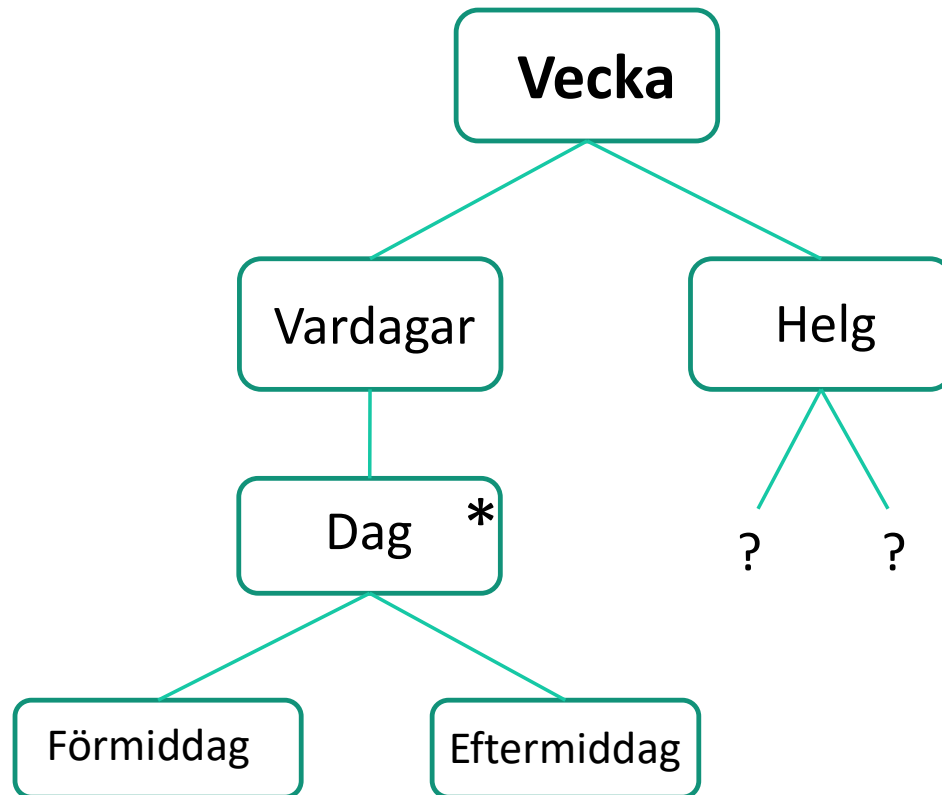


# JSP : Iteration : om och om igen och igen ...



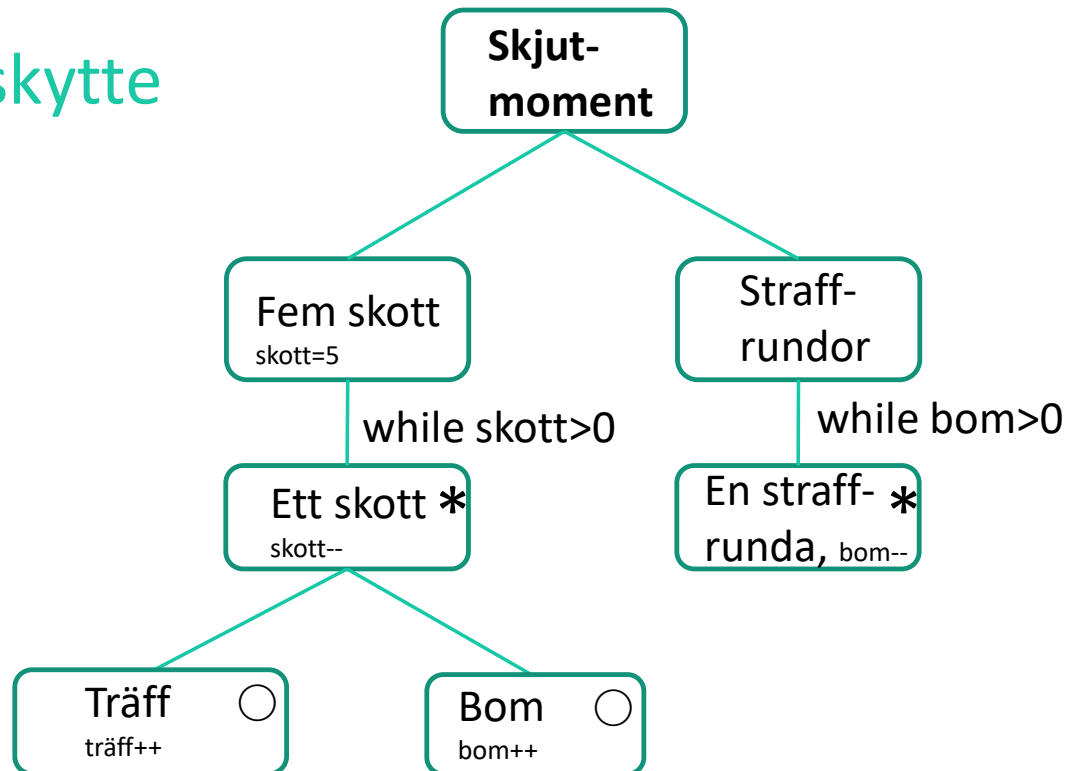
Iterationsdelen kan  
Utföras **NOLL** gånger!

# JSP : Kombinerad sekvens och iteration



# JSP : Selektion : det ena eller det andra eller ...

## Skidskytte



Iterationsdelen kan  
Utföras **NOLL** gånger!



# JSP : Regler

En komponent får endast ha delar av samma typ:

- En **sekvens** ska endast bestå av **sekvensdelar**
- En **selektion** ska endast bestå av **selektionsdelar**
- En **iteration** ska endast bestå av en (1) **iterationsdel**

Delarna kan dock utgöras av komponenter av annan typ.

En iteration kan ske noll gånger:

- Tänk **while()** inte **if()**

En selektion måste ha minst två delar:

- Ett val måste ha mer än ett alternativ
- Ofta ett defaultalternativ också

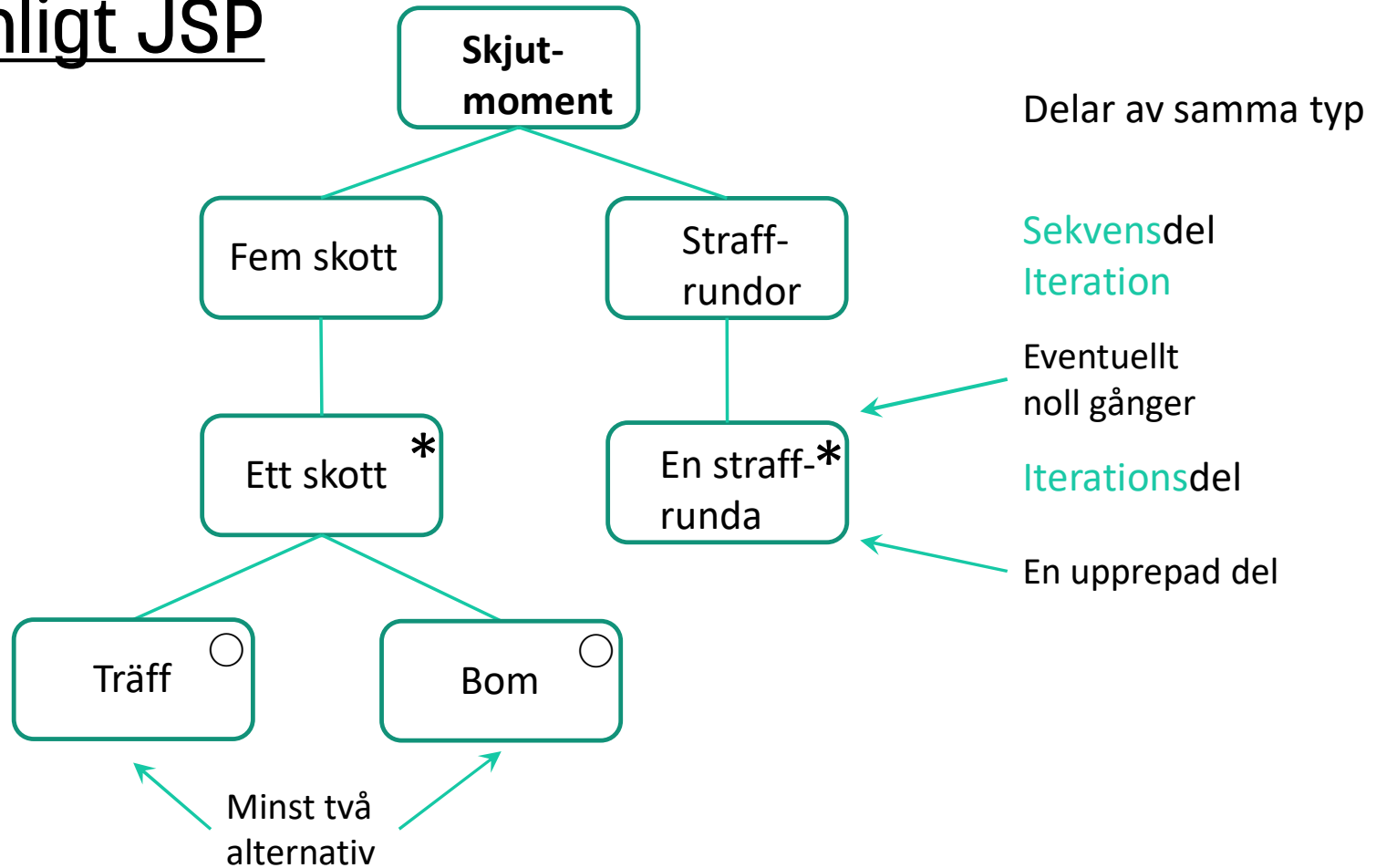
# Korrekt enligt JSP

Sekvens

Sekvensdel  
Iteration

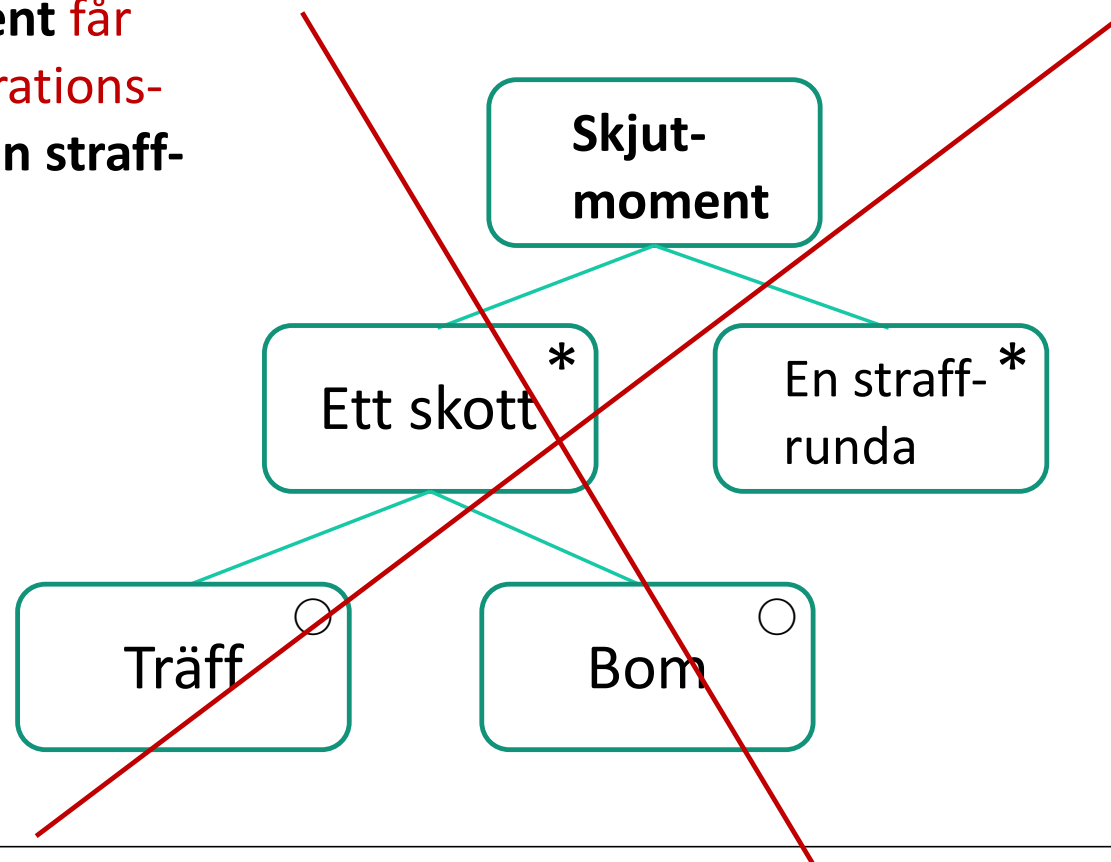
Iterationsdel  
Selektion

Selektionsdelar



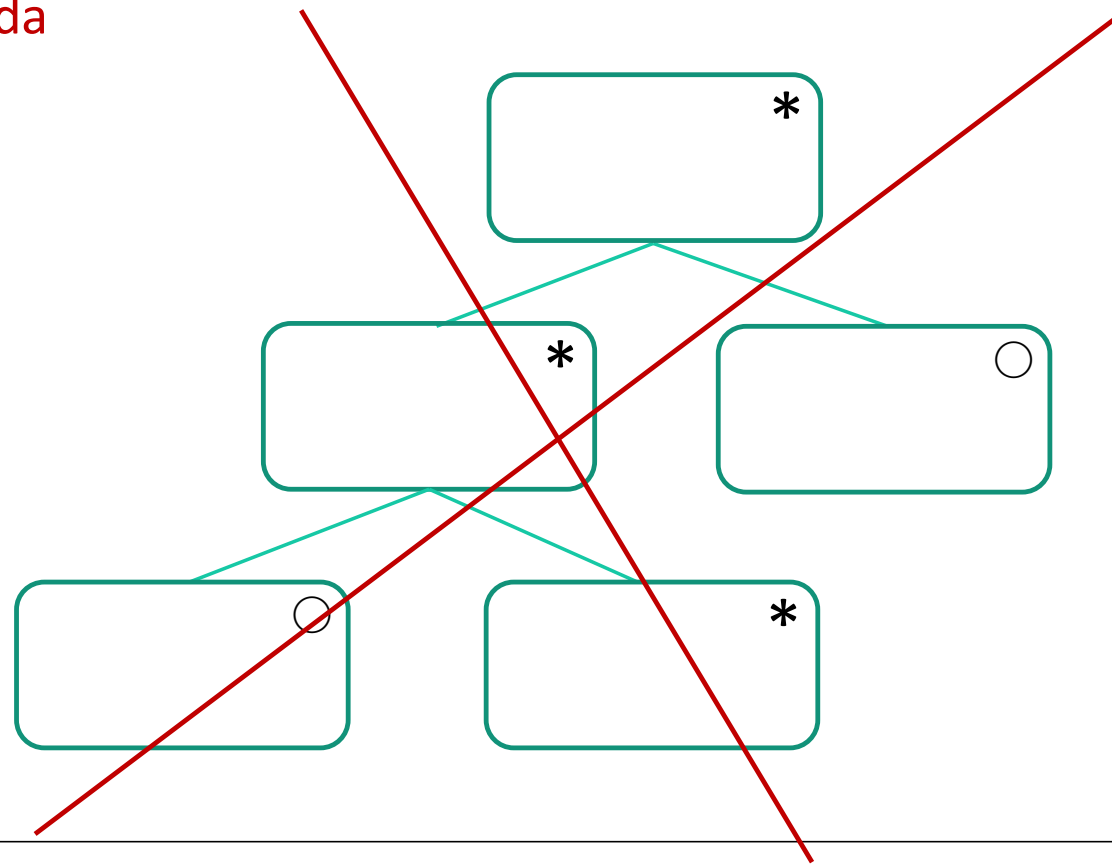
# Inte korrekt enligt JSP

Sekvensen **Skjutmoment** får  
t ex inte utgöras av iterations-  
delarna **Ett skott** och **En straff-  
runda!!**

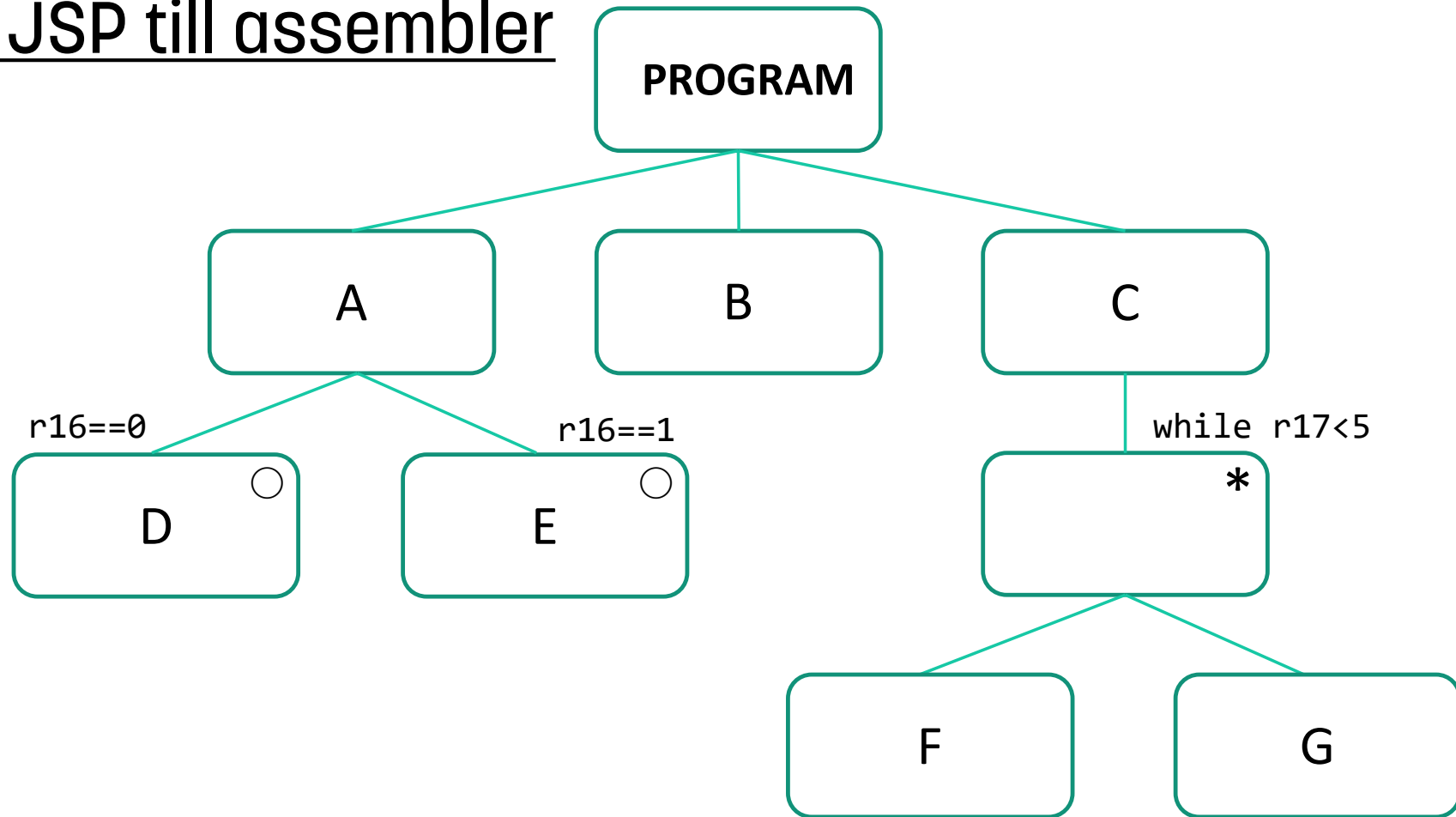


# Inte korrekt enligt JSP

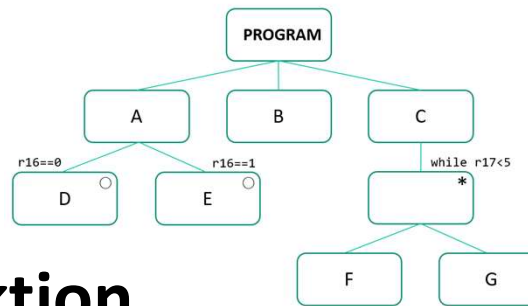
Man kan ej heller blanda  
delar av olika typ!!



# Från JSP till assembler



# Från JSP till assembler



## Sekvens

```

PROGRAM:
    call A
    call B
    call C
LOOP:
    ...
    jmp LOOP

B:
    ...
    ret
  
```

## Selektion

```

A:
    cpi r16,0
    brne A_1
    call D
    jmp A_EXIT
A_1:
    cpi r16,1
    brne A_DEFAULT
    call E
    jmp A_EXIT
A_DEFAULT:
    ...
A_EXIT:
    ret
  
```

## Iteration

```

C:
    clr r17
C_LOOP:
    cpi r17,5
    breq C_EXIT
    call F
    call G
    inc r17
    jmp C_LOOP
C_EXIT:
    ret
  
```

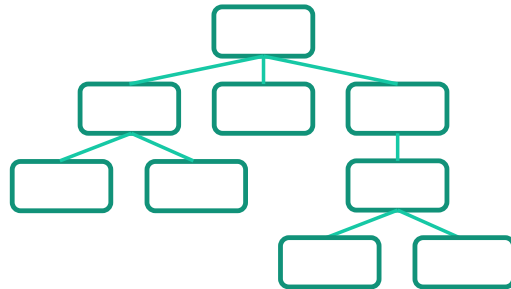
# Kodstil

# Från idé till program

Idé



Strukturerad lösning  
t ex via JSP



Programkod  
.asm

```
.org 0
jmp MAIN

.dseg $0100
ARR:
.byte $0C

.cseg
MAIN:
ldi r16,HIGH
out SPH,r16
ldi r16,LOW
out SPL,r16
...
```

Maskinkod  
.hex

```
:1001000049726F6E
:100110006D616964
:10012000656E7275
:100130006C657A20
:
:
:
:
:
:
:
:
```



# Övergripande princip

*Definiera först, använd sen*

**Vektortabell** : I huvudsak, hopp till huvudprogram men avbrottsvektorer

**Minnesanvändning** : I huvudsak, deklaration av var minnesanvändningen börjar, men även deklaration av variabler i minnet.

**Subrutiner** : Definition av subrutiner, gärna grupperade i samhörande moduler. Även definition av avbrottsrutiner.

**Huvudprogram** : Initiering av variabler/data, följt av huvudloop med anrop av subrutiner och villkorsstyrda förgreningar.

**Vektortabell**

Definition av  
**minnesanvändning**

Definition av  
**subrutiner/**  
underfunktioner

**Huvudprogram**  
med huvudloop

```

.org 0
jmp MAIN

.dseg $0100
ARR:
.byte $0C

.cseg
SUB1:
push r16
...
pop r16
ret

...

MAIN:
ldi r16,HIGH
out SPH,r16
ldi r16,LOW
out SPL,r16
MAIN_LOOP:
...
call SUB1
...
jmp MAIN_LOOP

```

# Vektortabell

I programminnets början finns en vektortabell dit exekveringen automatiskt hoppar vid reset/start av processorn (adress `0x000`), eller vid avbrott (någon annan adress beroende på vilket avbrott som inträffat).

I vektortabellen ska det bara finnas hoppinstruktioner som hoppar vidare till rätt plats i programmet beroende på vad som hänt.

```
.org 0x000
jmp  MAIN
.org 0x001
jmp  INT0_vect
.org 0x015
jmp  ADC_vect
...
```

# Minnesanvändning

Deklaration av var minnesanvändningen börjar inleds med `.dseg adress` vilket talar om var datasegmentet börjar.

Inne i datasegmentet kan man deklarera namn på variabler och hur mycket minne respektive variabel ska ha

```
variabel1:      .byte    minnesmängd
variabel2:      .byte    minnesmängd
```

Avslutningsvis talar man om var datasegmentet slutar och kodsegmentet börjar med `.cseg`

```
      .dseg 0x0100
VAR1:
      .byte 0x0C
VAR2:
      .byte 0x02
      .cseg
```

```
VAR1: [$0100..$010B]
```

```
VAR2: [$010C..$010D]
```

# Subrutiner

Subrutiner är ”underprogram” som anropas från andra delar av programmet, och där man återvänder (gör retur) till efter anropet när subrutinen är färdig.

Inne i en subrutin förekommer ofta behovet av en lokal användning av register, som kanske används på andra platser i programmet. Innehållet i de register som används lokalt sparas lämpligen undan (push) på processorns stack i början på subrutinen, och återhämtas (pop) i slutet. Detta kallas för att spara och återhämta kontext.

Alla symboliska adresser (lablar) i en subrutin ges lämpligen samma stam-namn (med något tillägg) som subrutinens namn.

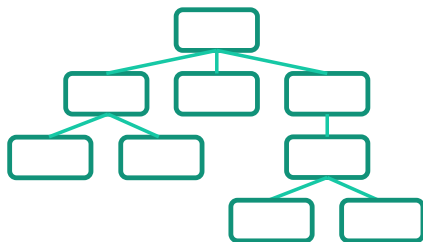
```
; SUB ADC_INIT
; Initialize A/D
ADC_INIT:
    push    r16
    ...
    ldi     r16,4
    out     ADMUX,r16
    ...
ADC_INIT_EXIT:
    pop     r16
    ret
```

# Huvudprogram

Huvudprogrammet ligger lämpligen sist i programkoden, då det typiskt kommer att använda och anropa de tidigare definitionerna av subrutiner. *Definiera först, använd sen.*

Huvudprogrammet inleds typiskt med någon form av initering följt av en huvudloop där programmet sedan för evigt kör.

Det är egentligen i huvudprogrammet som själva strukturen av det tidigare JSP-diagrammet bor. Huvudprogrammet byggs upp av villkorsstyrda programförgreningar och anrop av subrutiner.



```
; ### Main program
MAIN:
    ; set stack
    ldi    r16,HIGH
    out    SPH,r16
    ...
    call   ADC_INIT
    ...
MAIN_LOOP:
    ...
MAIN_GET_KEY:
    call   READ_KBD
    cpi    r16,4
    brne   MAIN_GET_KEY
    ...
    jmp    MAIN_LOOP
```

# Prioritetsordning

- 1. Funktion:** Programmets funktion är överordnat, men nästan lika viktigt är nästa punkt, struktur.
- 2. Struktur:** För att ett program ska fungera bra, vara smidigt att utveckla och vidareutveckla, vara läsbart och senare kunna optimeras är dess struktur avgörande.
- 3. Optimering:** När funktion och struktur finns på plats blir optimering ofta enkelt. Det handlar då typiskt om att stegvis reducera "randvillkor" mellan olika subrutiner eller moduler, om nu detta är nödvändigt. En skicklig och erfaren programmerare kan säkert uppnå extremt optimerad kod genom gå direkt på optimeringssteget och hoppa över struktureringen, men sådan kod blir ofta obskyr och mer eller mindre omöjlig att vidareutveckla.

# Labb 1

...

Tid för Frågor



Anders Nilsson

[www.liu.se](http://www.liu.se)