TDP002 - Problemlösning

Pontus Haglund

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Att lösa ett problem
  Stegvis förfining
2 Kontrollera komplexitet
3 regler
4 Rekursion

LINKÖPING UNIVERSITY

1 Att lösa ett problem
Stegvis förfining
2 Kontrollera komplexitet
3 regler
4 Rekursion

LINKÖPING UNIVERSITY

Att konstruera ett program

- Vad är uppgiften?
- Förstå problemet
  1. Beskriv problemet
  2. Gör en lösning (pseudokod)
  3. Implementera (programkod)
  4. Testa/utvärdera/verifiera)
  5. Börja om vid 1
- Skeppa produkten

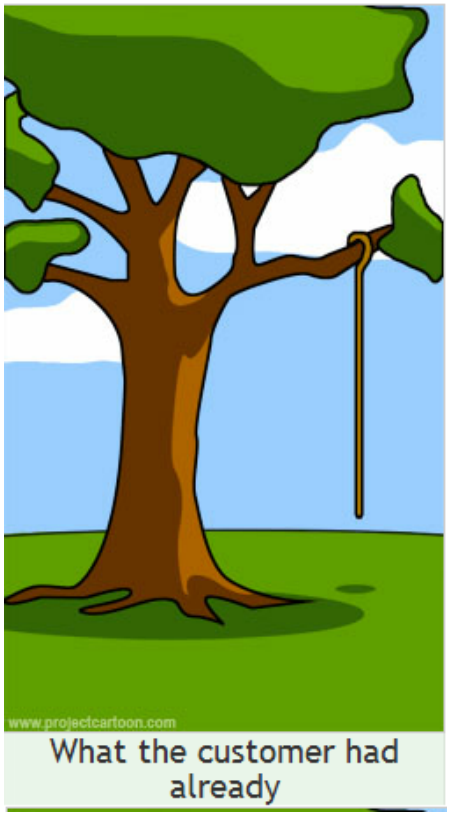
Vad är uppgiften?

- Möten
- Specifikationer
- labbuppgift

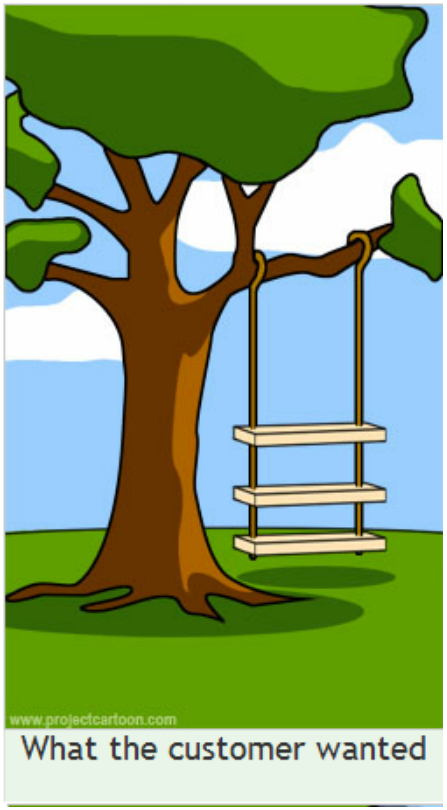
Förstå problemet

- Hur har jag förstått problemet
- Rubberduck
- Kund
- Partner

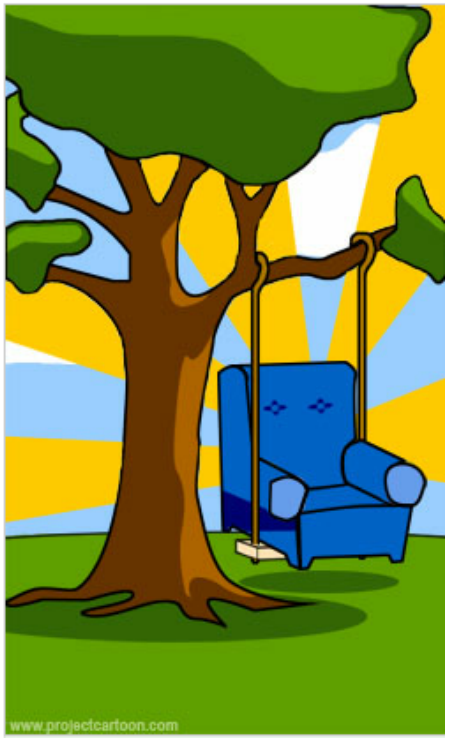
Systemutveckling



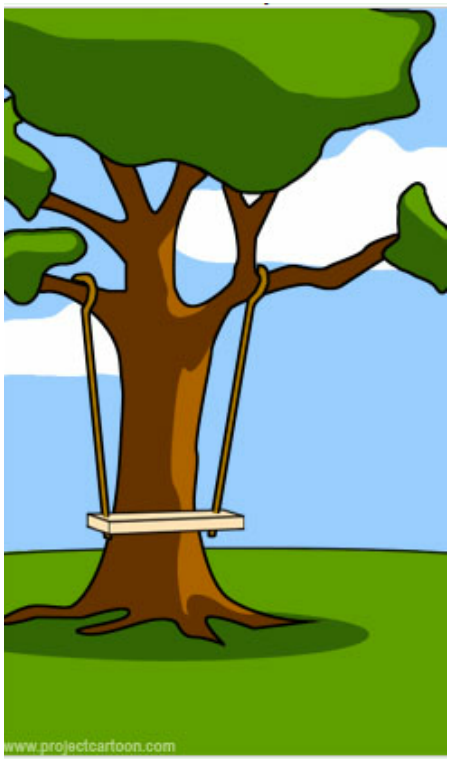
Systemutveckling



Systemutveckling



Systemutveckling



Systemutveckling

Systemutveckling

Kommunicera behov/uppgift

- Vi bryr oss inte om kunder i denna kurs
- Jag är kunden
- Hur du uppfattar uppgiften spelar roll
  - Har din partner uppfattat samma sak?
  - Är det vad jag menar?

Vattenfallsmodellen (waterfall)

Agil utveckling (agile)

Stegvis förfining

- Metod för att skapa ett program från ett problem
  - en kreativ process
- Formulera på informell nivå (vanlig svenska)
- Förfina olika steg till en mekanisk nivå
  - Gå mot kållkodslikt språk
- Kallas även top-down-design

verktyg för förfining

- Pseudokod
- Flödesdiagram

Exempel: skriv ut x-kvadrat i tabell

- Uppgift:
  - Skapa en tabell med uträkning av x-kvadrat för varje x mellan 1 och 20.

Körexempel

```
x: x^2:
========
1  1
2  4
3  9
4  16
.
.
========
```

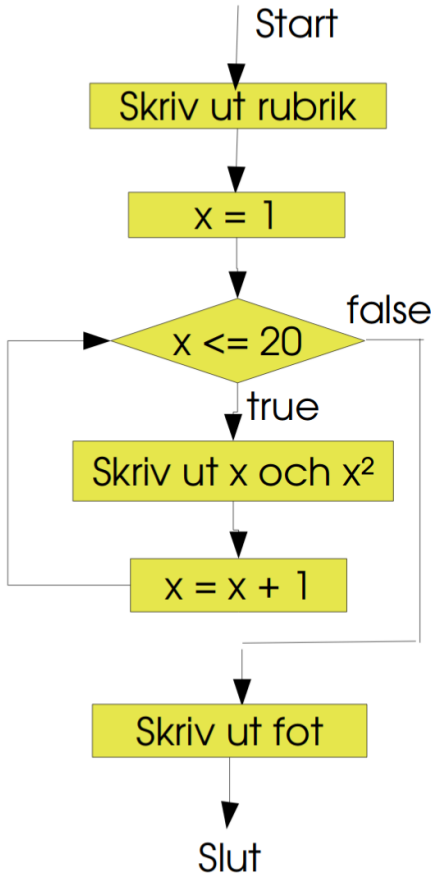
Pseudokod

- Pseudokod:
  1. Skriv ut tabellrubrik för 2 kolumner
  2. För varje tal x mellan 1 och 20:
     1. skriv ut värdet på x i kolumn
     2. skriv ut värdet på x**2 i kolumn
  3. Skriv ut tabellfot

Körexempel

```
x: x^2:
========
1  1
2  4
3  9
4  16
.
.
========
```

Flödesschema



Implementation

```
print('x: x^2:\n========')

for x in range(1, 20):
    print(x, x*x, sep='\t')
print('========')
```

```
x: x^2:
========
1  1
2  4
3  9
4  16
.
.
========
```

Tillbaka till början

1. Beskriv problemet
2. Gör en lösning (pseudokod)
3. Implementera (programkod)
4. Testa/utvärdera/verifiera)
5. Börja om vid 1

1 Att lösa ett problem
Stegvis förfining
2 Kontrollera komplexitet
3 regler
4 Rekursion

LINKÖPING UNIVERSITY

Programmeringens hantverk

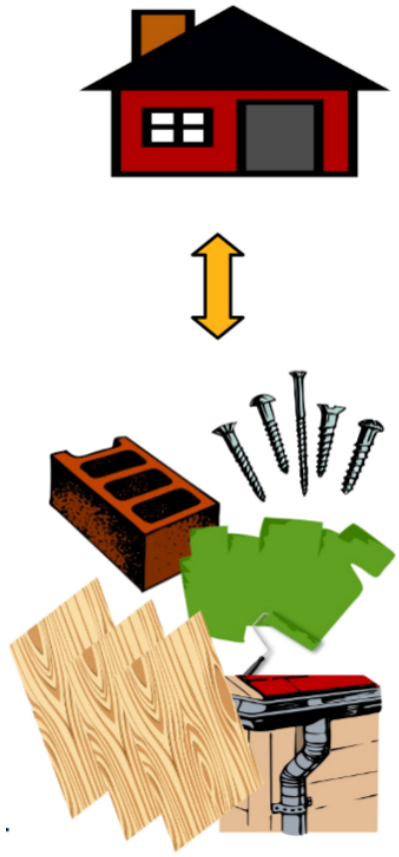
Controlling complexity is the essence of computer programming

-Brian Kernighan

Abstraktion

Dekomposition

- Uppdelning i delar
- Varje delning blir enklare
- Sträva mot att varje dela har ETT tydligt och enkelt syfte
- Summan av delarna ger heltheten



Variabler

Utan

```
print(int(input('Mata in tal: '))**2)
```

Med

```
user_input = int(input('Mata in tal: '))
squared = user_input ** 2
print(squared)
```

Dekomposition

- Variabler lagrar delresultat
- Varje sats löser EN avgränsad uppgift
- Funktioner separerar ut delar av programmet under nya namn

Satsnivå

Uppgift:

- Vi vill göra en uträkning för varje tal 0..9, samt en annan uträkning för varje jämnt tal.

Komplex lösning

```
results_1 = []
result_2 = []
for x in range(10):
    results_1.append(x * 3 / 5.0)
    if (x % 2 == 0):
        results_2.append(x * 4 / 3.0)
```

Satsnivå

```
results_1 = []
for x in range(10):
    result = x * 3 / 5.0
    results_1.append(result)

results_2 = []
for x in range(0,10, 2):
    result = x * 4 / 3.0
    results_2.append(result)
```

- mindre delar
- tydligare delar
- mindre effektivt?
- …

Procedurell dekomposition

```
def calculate_results_1():
    results_1 = []
    for x in range(10):
        result = x * 3 / 5.0
        results_1.append(result)
    return results_1

def calculate_results_2():
    results_2 = []
    for x in range(0,10, 2):
        result = x * 4 / 3.0
        results_2.append(result)
    return results_2

results_1 = calculate_results_1()
results_2 = calculate_results_2()
present_results(results_1)
present_results(results_2)
```

- Ortogonalitet: en funktion ett syfte
- Håll isär presentation och innehåll
  - (ha inte i/o och beräkningar tillsammans)

Funktioner

Utan

```
"""
Skriv ut primtal
"""
for i in range(1000):
    is_prime = True
    for j in range(i):
        if i%j == 0:
            is_prime = False
            break
    if is_prime:
        print(i)
```

Med

```
def prime(i):
    is_prime = True
    for j in range(i):
        if i%j == 0:
            is_prime = False
            break
    return is_prime

#Huvudprogram
for i in range(1000):
    if prime(i):
        print(i)
```

1 Att lösa ett problem
Stegvis förfining
2 Kontrollera komplexitet
3 regler
4 Rekursion

LINKÖPING UNIVERSITY

KISS

Keep It Simple, Stupid!

- En komplex uppgift är en kombination av flera enkla deluppgifter

Genom att dela upp programmets totala uppgift i delar håller vi varje deluppgift avgränsat och enkelt. Vi löser större uppgifter genom att kombinera lösningar på flera enklare uppgifter.

- Skriv aldrig komplex källkod för att lösa ett komplext problem (eller ett enkelt problem)

Exempel

Lösa sokoban på några minuter...

DRY

Don’t Repeat Yourself

- Varje bit av information ska finnas representerad endast en gång i ett program.

Exempelvis: namn på användare, färger i gränssnittet, storlek på fonter, algortimer för uträkningar...

- Duplicering gör det jättesvårt att underhålla och felsöka program.

1 Att lösa ett problem
Stegvis förfining
2 Kontrollera komplexitet
3 regler
4 Rekursion

LINKÖPING UNIVERSITY

Rekursion

Ett annat sätt att göra repetition

Räkna

Räkna från i upp till 10

Iterativ

```
def count_print(i):
    for j in range(i+1):
        print(i)
```

Rekursiv

```
def count_print(i):
    print(i)
    if i < 10:
        count_print(i+1)
```

Factorial

Iterativ

```
def factorial(n):
    summan = 1
    for i in range(1, n+1):
        summan *= i
    return summan
```

Rekursiv

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Summera listor

Iterativ

```
def sum_list(l):
    summa = 0
    for e in l:
        summa += e
    return summa
```

Rekursiv

```
def sum_list(l):
    if not l:
        return 0
    else:
        return l[0] + sum_list(l[1:])
```

Summera listor i listor...

Iterativ - är detta lika bra?

```
def sum_arb_list(l):
    tot = 0
    for x in l:
        if isinstance(x, list):
            for y in x:
                tot += y
        else:
            tot += x
    return tot
```

Rekursiv

```
def sum_arb_list(l):
    tot = 0
    for x in l:
        if not isinstance(x, list):
            tot += x
        else:
            tot += sum_arb_list(x)
    return tot
```

Towers of hanoi

```
A = [4, 3, 2, 1]
B = []
C = []

def move(n, source, target, auxiliary):
    if n > 0:
        # move n - 1 disks from source to aux, so they are out of the way
        move(n - 1, source, auxiliary, target)

        # move the nth disk from source to target
        target.append(source.pop())

        # Display our progress
        print(A, B, C, '###############', sep = '\n')

        # move the n - 1 disks that we left on auxiliary onto target
        move(n - 1, auxiliary, target, source)

# initiate call from source A to target C with auxiliary B
move(len(A), A, C, B)
```

www.liu.se

LINKÖPING UNIVERSITY