Datalogi OB, Efterår 2002 OH'er, forelæsning 3/9-2002

Datastrukturer og algoritmer

Henning Christiansen
henning@ruc.dk
http://www.ruc.dk/~henning

Formål: at kunne

- forstå datastrukturer og algoritmer (teoretisk forståelse og intuition)
- vurdere datastrukturer og algoritmer } relevans,
- vælge datastrukturer og algoritmer } objektive kriterier
- designe og implementere datastrukturer og algoritmer

Kursusindhold:

- at bruge ((Javas) OOP-) prog.sprogs-faciliteter til at strukturere
- klassiske datastrukturere og algoritmer
- metoder til at vurdere effektivitet...
- designe og implementere solide datastrukturer og algoritmer

Hvad vi ikke kommer ind på:

- tråde og parallelle algoritmer
- formel semantik af programmeringssprog og verifikation
- systematisk aftestning
- styring af store projekter (mange personer/grupper roder med kode og struktur samtidigt)

Lærebog: M.A.Weiss: Data Structures & Problem Solving using JAVA, Addison-Wesley 2002

Idag: Introduktion, motivation, grundlæggende begreber
læs kap. 1–3 (ca.)
Næste gang: detaljeret om nedarvning og hvad vi kan bruge det til
læs kap. 4 (ca.)
NB: Alt på www snarest!

NB, NB: Løs opgaverne, aftest dem, kom til øvelserne!

At udvikle og kunne vedligeholde systemer:

Finde ud af, hvad systemet skal lave -> skrive ned i et specifikation.

At give overordnet programdesign

- genbruge så meget eksisterende som muligt
- pile og kasser (f.eks. klasser og nedarvningshierarki eller procedurelt, hvem kalder hvem)
    - detailspecifikationer

"Kasserne" ikke for store og ikke for små:

- begrebsmæssigt forståelige enheder
- overkommelig teknisk indmad
    - så længe "kasse" for kompliceret til at implementere i ét huk,
        - del op internt /hjælpeklasser/pakker

    Top-down design

Bottom-up design:

"dette her ligner noget jeg har set før"

Se om det kan betale sig at generalisere?

Eksempel: Flasker (købe, fylde, tømme, lukke, åbne)

Dåser (købe, åbne, tømme, krølle)

- ⇨ Beholder (købe, åbne, tømme)
- ⇨ Flaske er beholder som ...
- ⇨ Dåse er beholder som ...

Mere bottom-up i design:

skrive program => forstå anvendelse + indre logik bedre => tilføje ny funktionalitet, ændre funktionalitet

Erfaren programmør: Forudser de fleste generaliseringer

At vedligeholde/at skrive ny systemer - grænsen er udflydende

- identificere og rette fejl
  \ symptomer i forhold til overordnet spec.
  \ lokalisere - Fejl i detail.spec? Fejl i implementation?
  - grundlæggende fejl i kasse-pile struktut? ØV
- tilføj ny funktionalitet/tilpas funktionalitet
  - ændringer i verden
  - ændringer ud fra erfaringer med system
  - programmørens forslag
- optimere: "det kører li'godt for langsomt"

At begrænse vedligeholdelse/gøre det nemt:

- skrive fejlfri, effektive og godt designede programmer fra starten
- undgå at det "samme" kode er gentaget flere steder
- veldefineret interface, indpakning af detaljer

Objektorienteret tankegang: Måde at strukturere begreber på
"noget med kasser og pile"

Objektorienteret programmeringssprog:

Programmeringssprog afspejler objektorienteret tankegang

⇨ sprogets konstruktioner svarer til “kasser og pile”

... se på, hvad Java tilbyder

Compiler for “monolitiske” programmeringssprog:

oversæt højniveau til noget udførbart

Compiler for OO sprog:

som overfor

+ generel administrator

Eks: nedarvning sparer meget copy-paste og små huskesedler

Filsystem, compiler og værktøj udgør en fælles omgivelse!!!

Krav til programmer/programdele

- veldefineret formål og korrekthed
- robusthed og sikkerhed
- effektivitet, store datamængder
- nemt at vedligeholde
- kan genbruges

Opnås ved lige dele metodik og sprog/værktøj

Programkvaliteter, som ikke understøttes særligt godt af OOP og OOP-sprog:

- hurtig omsætning af idé til kørende prototype eller system
- korte og koncise programmer

  "man skal skrive 10^10 kodelinjer før man kan få gjort noget,
  og så har man alligevel glemt hvad man ville"
- eksperimentel systemudvikling (= bottom-up når det er smukkest)
- glidende overgang fra prototype til systemaflevering

Sammenstilling at to programmeringspadigmer

	Java	**Prolog**
paradigme	imperativt/ procedurelt/OOP	logikprogrammering/ regelbaseret
strengt typet	+	–
variabelerklæringer	+	–
indkapsling	+	(–)
semantik	mareridt	enkle matematisk begreber (hmm)
udtrykskraft pr. linje	0,01	1000
effektivitet	ja, når brugt rigtigt	ja, når brugt til rette ting
robusthed/sikkerhed	ja, når brugt rigtigt	ikke en dyt!
genbrug/generisk	ja, men tager tid	skriv forfra

Et programmeringssprog består af *små* og *stort*

Småt: simple typer, simple variable
assigment, sekvens

Halvstort: arrays, referencer
løkker a la for og while...

Stort: Abstraktionsmekanismer

Abstraktionsmekanisme:

1) indkapsel og navngive
   procedure p; begin yksi-kaksi-kolme end
2) benytte ved navns nævnelse
   x= 17; p; p; ...
3) parameterisere
   procedure p(x: integer); begin .... p .... end
4) specialisering (nedavning)
   "implicitte parametre"

Abstraktionsmekanismer er metasprog i sproget selv

Abstraktioner er sprogudvidelser

Spørgsmål:

- abstraktion over hvilke syntaktiske kategorier?
  sætninger, konstanter, udtryk, typer, erklæringer, abstraktionsmekanismer??
- hvilke slags parametre?
  simple værdier, sætninger ("definer din egen løkke"),
  klasser (generisk ...)
- parameteroverførsel
  call-by-value (JAVA)
  call-by-reference
  call-by-name
- scoperegler (indmad, det navngivne), lokal/global?, overloading, ...
- hvordan foregår specialisering/nedarvning

Klasser: Dataabstraktion

Beslægtet med abstrakt datatype: defineret ene og alene ved operationerne

Klasser har interface + indmad.

- interface: det som "bruger" kan se, public felter og metoder
  ideelt set: specifikation af, hvad der sker
- indmad: kode som implementerer; "bruger" uvedkommende; kun relevant for "implementør"
  kan meget vel være samme person!

Eksempel:

```
class Ticket
{
   public Ticket( )  {        System.out.println( "Calling constructor" );
                              serialNumber = ++ticketCount; }

   public int getSerial( ) { return serialNumber; }

   public String toString( ) {return "Ticket #" + getSerial( ); }

   public static int getTicketCount( ) { return ticketCount;}

   private int serialNumber;
   private static int ticketCount = 0;  }

class TestTicket
{
   public static void main( String [ ] args )
   {    Ticket t1; Ticket t2;
        System.out.println( "Ticket count is " + Ticket.getTicketCount( ) );
        t1 = new Ticket( );
        t2 = new Ticket( );
        System.out.println( "Ticket count is " + Ticket.getTicketCount( ) );
        System.out.println( t1.getSerial( ) );
        System.out.println( t2.getSerial( ) );     }}
```

Interface:

```
class Ticket
// forklaring i tekst på hvad det her er!
  {public Ticket( )  {-}

    public int getSerial( ) { - }

    public String toString( ) {-}

    public static int getTicketCount( ) {-}  }

class TestTicket
{ public static void main( String [ ] args ) {-}}
```

Særlige metoder:

- main
- konstruktør
- toString
- equals

Problem ved sprog: Skelnes ikke klart mellem interface og indmad

Kan kompenseres ved smart editor integeret med compiler
... værktøj som "Javadoc" (s. 65 i bog) er bøjet som som giver lidt af det...

Packages: integrere klasser med filsystem
konventioner for at opdele programmer på filer; styre synlighed af klasser
!! gør Java til ét stort global program (filsystem = Internet)
[læs selv]

Afslut: Undtagelser og brugerdefinerede over undtagelser

Formål:

- gøre det lidt nemmere at lave sikre programmer uden at teste alle steder
- lave "lange udhop" i sære tilfælde (i stedet for at teste hele vejen op)

Uden eksplicitte undtagelser: pseudokode:

type int_med_check = int U "fejl" ; // nedarvning eller klasse med statusfelt

a/b --> if(b=0 || a="fejl" || b="fejl", "fejl", a/b)

a+b --> if(a="fejl" || b="fejl", "fejl", a+b)

... kan måske afhjælpes ved overloading (?); men ineffektivt og er man nu sikker på at ha' fået det hele med?

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class DivideByTwo
{
   public static void main( String [ ] args )
   {
      BufferedReader in = new BufferedReader( new
                        InputStreamReader( System.in ) );
      int x;
      String oneLine;

      System.out.println( "Enter an integer: " );
      try
      {
         oneLine = in.readLine( );
         x = Integer.parseInt( oneLine );
         System.out.println( "Half of x is " + ( x / 2 ) );
      }
      catch( IOException e )
        { System.out.println( e ); }
      catch( NumberFormatException e )
        { System.out.println( e ); }
   }}
```