Assignment 2 - Literate programming

Kenneth Skovhede Morten N. Larsen

1. Literate Programming

I mange år er programmer blevet betragtet som konstruktioner på linie med ingeniør konstruktioner. At konstruere programmer på denne måde, giver programudviklingsfasen de egenskaber som ingeniør værket. De ønskede egenskaber er projektion af tid og ressourcer, samt modularitet. Desværre er programmer anderledes komplekse, og det sker tit at budgettet overskrides og at modulerne ikke er reagerer sammen som ønsket.

Dette problem opstår fordi det en særdeles kompliceret affære at konstruere et matematisk bevis for et moduls korrekthed. Dette er i kontrast til ingeniørens verden, hvor der er klare definitioner og regler for hvor meget belastning en skrue kan håndtere. For at løse dette problem bliver man nødt til at anskue programkode anderledes. *Literate programming* er et forsøg på at anskue programkode som et stykke litterær tekst. I Knuts oprindelige tekst[1], beskrives *Literate Programming* som en ændring i måden programkode udvikles på. I stedet for at fokusere på at skrive computer instruktioner, beskrives meningen med funktionen i et menneskeligt sprog. Nedenstående viser et eksempel på hvordan man kunne ønske at programkoden kunne se ud. Eksemplet udregner arealet af en cirkel givet radius'en. Kildekode fundet på *"http://en.wikipedia.org/wiki/Literate_programming"*.

```
- print text Let's work out the area of a circle.
- newline
- print text Please enter the radius of the circle in furlongs:
- store input
- multiplyby value
- multiplyby 3.14159
- print Thank you
- newline
- print text The area of the circle is
- print value
- print text square furlongs.
- newline
- newline
```

1.1 Literate programming programmer

Da det endnu ikke er muligt at programmere en computer med naturligt sprog, er Knuts løsning et system der hedder *CWEB*[2]. *CWEB* er et system hvor det er muligt at kombinere *latex* kode med *C* kode. Iden er at det er muligt at skrive et litterært værk med indlejret kode. Der bruges forskellige tags i dokumentet til at vise om det er latex text, overskrifter m.m. eller c kode. Der findes så værktøjer som kan udtrække *latex* kode og

Written 3. maj 2007.

C kode til separate filer. Disse to filer kan så compiles til et læseligt dokument f.eks. dvi og c koden til en .exe. Umiddelbart virker det ikke som den store ændring i forhold til andre dokumentationsværktøjer. Ændringen er mere i filosofien bag, end i de konkrete værktøjer. I CWEB er strukturen således at man opfordres til at til at beskrive programkoden, meget mere end bare ved at beskrive alle argumenterne. For at følge literate programming filosofien, er det meningen at denne integrerede beskrivelse, skal gøre det lettere for programmøren selv, og andre at forstå og benytte koden. Eksempler på kode skrevet i CWEB kan bl.a. findes på *http://www-cs-faculty.stanford.edu/ knuth/programs.html*. Et eksempel på hvordan et CWEB dokument kan se ud ses nedenfor. Dette er et lille uddrag af koden "sham.w" *"http://www-cs-faculty.stanford.edu/ knuth/programs/sham.w"*.

```
\datethis
@i gb_types.w

@*Symmetric Hamiltonian cycles. This program finds all Hamiltonian cycles of
an undirected graph in which the mapping $v\mapsto N-1-v$ is an
automorphism, such that the same automorphism also applies to the cycle.

We use a utility field to record the vertex degrees.

@d deg u.I
@d mm 8 /* should be even */
@d nn 9

@c
#include "gb_graph.h" /* the GraphBase data structures */
#include "gb_basic.h" /* standard graphs */

main()
{
    Graph *g=board(mm,nn,0,0,5,0,0); /* knight moves on rectangular chessboard */
    Vertex *x, *z, *tmax;
    register Vertex *t,*u,*v;
    register Arc *a,*aa, *yy;
    register int d;
    Arc *b,*bb;
    int count=0,dcount=0;
    int dmin;
    @<Reduce |g| to half size@>;
    @<Prepare |g| for backtracking, and find a vertex |x| of minimum degree@>;
    for (v=g->vertices;v<g->vertices+g->n;v++) printf(" %d",v->deg);
    printf("\n"); /* TEMPORARY CHECK */
    if (x->deg<2) {
        printf("The minimum degree is %d (vertex %s)!\n",x->deg,x->name);
    return -1;
}
for (b=x->arcs;b->next;b=b->next) for (bb=b->next;bb;bb=bb->next) {
    a=b;
    z=bb->tip;
    @<Find all simple paths of length |g->n-2| from |a->tip| to |z|, avoiding |x|@>;
}
printf("Altogether %d solutions and %d wannabees.\n",count,dcount);
for (v=g->vertices;v<g->vertices+g->n;v++) printf(" %d",v->deg);
printf("\n"); /* TEMPORARY CHECK, SHOULD AGREE WITH FORMER VALUES */
}
```

Andre eksempler på literate programming er f.eks. JavaDoc. JavaDoc er meget brugbart som dokumentation i objekt-orienterede programmer. Dokumentationen skrives i klasserne og der benyttes en særligt tegnsæt "/**...*/". Dette er en nem og overskuelig måde at dokumentere koden på samtidig med at man koder. Netop det at man dokumentere direkte i koden medfører at sandsynligheden for at dokumentationen rettes under afprøvningen stiger eftersom man ikke skal rette kode i et dokument og dokumentation i et andet. I CWEB bor programmet i dokumentationen og derved bliver integrationen endnu bedre, dog kan dette også betyde at man nemt mister lidt overblik

eftersom dokumentation og kode er så tæt integreret.

2. Konklusion

Dokumentation af programmer er ofte krævende og meget omfattende tidsmæssigt. Ofte vil man opleve at dokumentationen ikke bliver opdateret ved små rettelser i koden. Dette ville f.eks. JavaDoc eller CWEB kunne rette lidt op på, da det er nemt at rette i dokumentation når man rette i koden. Tilgengæld synes vi at koden / dokumentationen i en fil nemt bliver uoverskuelig, selvfølgelig kunne man læse dem hver for sig, men det ødelægger noget af overblikket. Og efter som det ikke er særlig brugt i den "virkelige" verden er det måske også tvivlsom hvor anvendeligt det er.

Litteratur

[1] D. Knuth, *Literate Programming (1984)*, CSLI (1992).
[2] D. Knuth, *The CWEB System of Structure Documentation*, Addison-Wesley (1994).