Kontraktbaseret Design

Anker Mørk Thomsen

17. maj 2014

Kontraktbaseret Design

1. udgave

ISBN: 9788740491500

Forord

Bogen er blevet til gennem undervisning i faget Kontraktbaseret Udvikling på bacheloruddannelsen i Softwareudvikling. Emnet har jeg opdelt i to dele. Den første del omfatter konstruktion og udvikling af algoritmer ved anvendelse af formelle metoder. Denne første del indeholder også den diskrete matematik, som danner grundlag for at kunne arbejde med de formelle metoder. "Kontraktbaseret Design" omhandler design af programmodulerne i et objektorienteret sprog og er inspireret af "Design by Contract by example", se Mitchell and McKim [2002].

Januar 2014

Anker Mørk Thomsen

Indhold

Introduktion

Bogen er beregnet til brug som lærebog på kurset Kontraktbaseret Udvikling på erhvervsakademiernes bacheloruddannelse i Software Udvikling. Bogen indeholder pensum til den del af kurset, som omhandler design af programmoduler baseret på specifikationer. Indholdet er inspireret af Michell og McKims "Design by Contract by Example"Mitchell and McKim [2002]. "Design by Contract®" er et varemærke, som ejes af Bertrand Meyer, som konstruerede programmeringssproget Eiffel. I "Design by Contract by Example" anvender Mitchell og McKim programmeringssproget Eiffel, hvor man kan skrive specifikationer direkte i koden. Eiffel udmærker sig ved, at specifikationssproget er simpelt og let forståeligt.

I "Kontraktbaseret Programmering" Thomsen [2014] anvendte jeg en programnotation, som blev introduceret af E. W. Dijkstra blandt andet i bogen "A Dicipline of Programming" Dijkstra [1976] Dijkstra kaldte notationen Q. I denne notation anvendes operatoren `:=` som tilordningsoperator og ikke `=` som i *Java*, C#, C og C++. Anvendelse af `=` som tilordningsoperator giver det indtryk, at der er tale om lighed og ikke om en aktivitet. Sætningen `int k=7` i *Java* skal ikke læses "k er lig med 7" men "k sættes til 7". Brugen af = som tilordningsoperator har konsekvensen, at logisk ækvivalens i boolske udtryk kræver en operator, der ikke kan forveksles med =. Sprogene *Java*, C#, C og C++ anvender `==` til sammenligning af simple værdier og referencer. I notationen Q anvendes `:=` som tilordningsoperator og = til sammenligninger. I Java betyder sætningen

```
boolean p = (x==y);
```

at variablen p får tildelt værdien af $x = y$. I Q skrives dette

```
boolean p:= (x=y)
```

Notationen i Q er bedre end notationen i Java og de andre ovennævnte sprog. Jeg har derfor overvejet at anvende Q, men indtil videre vil jeg bruge *Java*. Derved er det muligt at afprøve bogens eksempler uden at skulle oversætte dem fra Q til *Java*.

Specifikationerne (kontrakterne) skrives i specifikationssproget JML. Referencemanualen til JML Leavens and et. al. [2002] er på 196 sider. Et skræmmende omfangsrigt og kompliceret dokument til beskrivelse af specifikationssproget. Det er vigtig, at et specifikationssprog er simpelt, og at beskrivelsen af syntaks og semantik kun fylder få sider. JML's kompleksitet afspejler også, at det er lykkedes at udvide Java, så også Java er blevet en programteknisk "Jumbo Jet".

De fleste af bogens eksempler anvender generiske typer. Anvendelse af exception er undgået. Specifikationerne skrives i en simpel delmængde af JML, hvor kun følgende nøglebegreber anvendes:

- `pre Q;`
- `post R;`
- `assignable x;`
- `invariant C;`

JML indholder muligheden for at skrive specifikationer på flere niveauer, kaldet letvægts- og sværvægtsspecifikationer. Desuden kan man i JML specificere "normal behaviour" og "exceptional behaviour". Vi anvender kun letvægtsspecifikationer, idet de er fuldt dækkende for en komplet specifikation. Bogens eksempler benytter ikke kald af exceptions undtagen i de tilfælde, hvor der er tale om en programmeringsfejl, eller hvor Java tvinger programmøren til at bruge exceptions (try-catch), f. eks. ved input/output. Anvendelse af exceptions anser jeg for at være en form for goto-programmering og kan kun medvirke til skrivning af programmer, som er vanskelige at gennemskue.

Første kapitel indeholder en præsentation af, hvorledes man udformer og udvikler specifikationer for et spil-program samt at par almindeligt anvendte typer af moduler, der repræsenterer registre (objektsamlinger). Kapitlerne 2 til og med 6 behandler kontrakter for en række abstrakte datatyper. Derefter følger et kapitel, hvor bogen behandler kontrakter for sybtyper. I kapitlet om subtyper gennemgås Liskovs subsitutionsprincip. Bogens sidste kapitel behandler kontrakter for et rammesystem, der realiserer designmønstret Observer.

Bogen er udformet, så den kan læses uafhængigt af bogen "Kontraktbaseret Programmering".

Kapitel 1

Kontraktbaseret Design

På engelsk kaldes emnet for *Design by Contract*. Begrebet er introduceret af Bertrand Meyer allerede i 1986 i programmeringssproget ™Eiffel. Sproget Eiffel blev beskrevet i en række artikler og senere i bogen *Object Oriented Software Construction*. I 2009 udgav Springer Verlag Bertrand Meyers nye bog *Touch of Class* med undertitlen *Learning to Program Well with Object and Contracts*. Her finder man en grundig indføring i objektorienteret programmering i Eiffel med brug af specifikationer og løkkeinvarianter, se Thomsen [2014].

Kontraktbaseret Design (KD) er idag et omfattende forskningsområde, og teknikken er begyndt at finde anvendelse i erhvervslivet især på områder, hvor det er af stor betydning at kunne skrive korrekte programmer. Et korrekt program overholder de krav, der stilles til programmets udførelse [1]. Disse krav opdeles ofte i flere typer af krav. Det kan være krav til udførelsestid, krav til pladsforbrug under udførelsen og ikke mindst krav til funktionalitet. I KD koncentrerer vi os om de funktionelle krav. Vi beskæftiger os ikke generelt med krav til programmernes effektivitet, dvs. programmernes tids- og pladsforbrug. Alligevel vil vi undertiden skele til effektiviteten, dvs. tids- og pladskompleksiteten.

De funktionelle krav vil i den udstrækning, det er muligt og hensigtsmæssigt, blive formuleret som formelle specifikationer. Der findes to måder, hvorpå man kan kontrollere, om et program overholder specifikationerne. Man kan udføre programmet med forskellige input og så kontrollere, at programmets resultater svarer til kravene i specifikationerne. Dette svarer til at udføre test. Hvis specifikationerne er formulerede i et formelt sprog, kan run-time-systemet foretage en automatisk kontrol af programudførelsens resultater. Dette kaldes dynamisk prøvning eller test af programmet. Den anden måde består i at analysere programteksten (kildekoden), og gennemføre et bevis for, at programmet overholder specifikationerne. Denne kontrol kaldes en statisk prøvning, og den kræver ikke, at programmet udføres. En statisk prøvning kan afgøre, om et program er korrekt, dvs. at programmet giver korrekte resultater uanset input. Den dynamiske prøvning kan derimod ikke afgøre korrektheden af et program men blot sandsynliggøre, at programmet ikke indeholder fejl. En test af et program kan kun vise tilstedeværelsen af fejl. En test kan ikke vise fravær af fejl. I bogen Thomsen [2014] gennemgås nogle fremgangsmåder til at konstruere korrekte algoritmer.

[1] Det har god mening at definere, hvad der forstås ved et korrekt program. Næsten alle større programmer indeholder fejl, som bevirker, at programmerne ikke er korrekte. De udfører ikke i alle tilfælde den ønskede beregning.

Vi er interesseret i at kunne skrive kontrakter for programsystemer. Hvordan skal sådanne kontrakter udformes? Kan vi skrive en kontrakt, der omfatter et helt program endsige et system af programmer? Svaret er både ja og nej. Det vil være en uoverkommelig og meget kompliceret opgave at formalisere alle funktionelle specifikationer for et system af programmer. Vi kan dog indfange mange væsentlige dele af programmernes funktion gennem formelle specifikationer.

1.1 Specifikationer

En specifikation for en algoritme består af tre prædikater (betingelser): en præbetingelse, en postbetingelse og en angivelse af, hvilke variabler, der må ændres af algoritmen (`assignable`). En specifikation for en procedure har følgende opbygning

```
/*@ pre Q;
  @ assignable x;
  @ post R;
  @*/
void <navn>(<param>);
```

Nøgleordet `void` angiver i *Java*, at metoden er en procedure. `<navn>` er procedurens navn og `<param>` er listen af parametre. Procedurens krop indeholder den algoritme, som specifikationen vedrører. Metodesignaturen er også en del af specikationen, idet den definerer input til metoden. I *java* er alle metoder medlemmer i en klasse, og metoderne kan tilgå klassens variabler og i mange tilfælde også variabler i andre klasser. Metoden kan derfor operere på globale variabler. Selv om det er tilladt i *Java*, er det dårlig programmeringsskik at anvende globale variabler uden for den klasse, hvor metoden er medlem.

Nøgleordet `pre` efterfølges af præbetingelsen i form af prædikatet Q. Præbetingelsen beskriver de programtilstande, i hvilke det er tilladt at kalde proceduren. Programmets variabler skal have værdier, der opfylder Q, hvis man vil kalde proceduren `<navn>`. Er Q ikke opfyldt, kan kalderen ikke stille krav til funktionens resultat. Procedurens parametre er input til proceduren.

Nøgleordet `assignable` efterfølges af en liste af de variabler, som må ændres af proceduren. Det er underforstået, at ingen andre variabler må ændres af proceduren. I Java anvendes call-by-value, hvilket indebærer, at argumenterne til et procedurekald kopieres til lokale variabler i proceduren. Hvis parametrenes værdier ændres af proceduren, ændrer dette ikke på argumenternes værdier. Der er derfor ingen grund til at nævne parametrene i en assignable-sætning. En ændring af parametrene i procedurens krop har ingen virkning på procedurens omgivelser og specielt ikke på objekttilstanden. Man skal altid nævne de globale variabler, der ændres af proceduren, i assignable-sætningen.

Nøgleordet `post` efterfølges af et prædikat, der beskriver procedurens resultat. Proceduren skal, såfremt den er kaldt på legal vis, terminere i en tilstand, hvor postbetingelsen er opfyldt. Det fordres altså, at proceduren faktisk terminerer. Der må ikke være nogen uendelig løkke i procedurens krop.

En algoritme er korrekt, hvis og kun hvis følgende er opfyldt:

Hvis man begynder udførelsen af algoritmen i en tilstand, hvor præbetingelsen er sand, så skal algoritmen terminere i en tilstand, hvor postbetingelsen er sand. Algoritmen må kun ændre værdierne for de variabler, der er nævnt i assignable-listen.

Eksempel 1.1. Proceduren

```
/*@ pre x>=0;
  @ assignable t;
  @ post t == \old(t)+x;
  @*/
void inc(int x)
```

Her er `x` en global variabel, f. eks. en instansvariabel i den klasse, hvor proceduren `inc` er defineret.

En specifikation af en funktion har denne opbygning

```
/*@ pre Q;
  @ post R;
  @*/
<returtype> <navn>(<param>);
```

`<returtype>` er angiver typen af den værdi, som funktionen returnerer. Normalt vil der ikke være nogen assignable i specifikationen, da vi normalt ikke tillader, at funktioner påvirker deres omgivelser. Dette indbærer, at globale variabler kun aflæses.

Eksempel 1.2. Specifikation for en funktion:

```
/*@ pre 0<=k && k<b.length;
  @ post \result == (b[k]%x==0);
  @*/
boolean gårOpI(int[] b, int k, int x);
```

Funktionen undersøger, om `x` går op i `b[k]`. Funktionen må kun kaldes, hvis `k` er et gyldigt indeks i `b`.

1.2 Spillet NIM

Eksemplet nedenfor viser en funktionel specifikation af et spil mellem to spillere. Den ene spiller er en person. Den anden spiller er computeren.

Eksempel 1.3. Spillet NIM.

Spillet NIM er et tændstikspil med to spillere A og B. Spillet består i, at der lægges nogle rækker med tændstikker, f. eks. fire rækker med henholdsvis $(3, 6, 5, 8)$ tændstikker. Man trækker lod om, hvem der skal starte. På skift må hver af spillerne nu fjerne én eller flere tændstikker fra én række efter eget valg. Den spiller, som tager den sidste tændstik, har vundet.

En specifikation for et program, der kan spille NIM mod en person, kan udvikles skridt for skridt. I første omgang beskrives programmets startbetingelse og slutbetingelse. Startbetingelsen skal beskrive, at der er n rækker, der hver indeholder en eller flere tændstikker. Dette modellerer vi med en liste, hvor hvert element i listen angiver antallet af tændstikker i den pågældende række. Lad os kalde listen tr. Udtrykket $tr(i)$ angiver da antallet af tændstikker i række nummer i.

Der skal være mindst én række tændstikker, når spillet begynder. Dette kan vi beskrive med betingelsen $n > 0$. At alle rækker skal indeholde mindst én tændstik, kan vi beskrive med:

- For alle hele tal i, hvor $0 \leq i < n$, skal $tr(i) > 0$.

I matematisk notation skriver vi denne betingelse med prædikatet

$$(\forall\, i \,:\, 0 \leq i < n \,:\, tr(i) > 0)$$

Udtrykket $person()$ angiver, hvem der er i trækket. Hvis $person() \equiv true$ er det personens tur til at trække tændstikker. Hvis $person() \equiv false$ er det computerens (programmets) tur til at trække. Vi kan vælge at trække lod om, hvem der skal begynde spillet. I så fald er startbetingelsen for spillet *true*. Altså at det er uden betydning, hvilken værdi $person()$ har, når spillet begynder.

Læg mærke til, at vi anvender en boolsk funktion $person()$ til at markere, hvem der skal trække. Hvis vi havde brugt en variabel, så ville det kræve, at denne variabel var tilgængelig i implementeringen af spillet. Ved at anvende en funktion overlades det til programmøren, hvorledes "hvem skal trække" skal implementeres.

Spillet skal ende i en tilstand, hvor der ikke er flere tændstikker tilbage. Dette kan vi udtrykke ved at skrive, at summen af alle tallene i tr skal være 0. Specifikationen bliver i JML-notation:

```
/*@ pre  (\forall int i; 0<=i && i<n; tr(i)>0) && n>0;
  @ post  (\sum int i; 0<=i && i<n; tr(i)==0);
  @*/
{ glovar int n;
  Program
}
```

Præbetingelsen fordrer, at programmet begynder i en tilstand, hvor alle rækker indeholder tændstikker. Postbetingelsen kræver, at det korrekte program ender i en tilstand, hvor alle tændstikker er fjernet. Det er slutværdien af $person()$, der afgør, hvem der har vundet spillet. Programmet skal ændre på resultatet af funktionen $tr()$ og på resultatet af funktionen $person()$.