# 8
# Specifikation med Logiske Udtryk.

Specifikation kontra program.

Specifikation af funktioner.

Specifikation af funktions-orienterede ADT-er.

Integreret specifikation og program i Eiffel.

Korrekthed af Eiffel klasse i forhold til dens specifikation.

Overgangen til fejlhåndtering.

**PS1 -- Specifikation med logiske udtryk**



## Noter

Notationen til specifikation af funktioner samt en del af eksemplerne i denne forelæsning er taget fra bogen *Cliff B. Jones* "Software Development -- A Rigorous Approach", Prentice Hall, 1980.

## Specifikation i forhold til program.

**Hvad og ikke hvordan**
En specifikation af en abstrakt datatype udtrykker, *hvad* operationerne gør, snarere end *hvordan* den gør det.

**Bevis af korrekthed**
En specifikation kan benyttes til at bevise, at en implementation af en abstrakt datatype er korrekt i forhold specifikationen.

**Check af korrekthed**
En specifikation kan benyttes til at checke en implementation af en abstrakt datatype under programudførelsen.





# Noter

Vi har i en tidligere forelæsning stiftet bekendtskab med den algebraiske specifikations-teknik. I denne forelæsning vil vi introducere en anden form for specifikationer, baseret på logiske udsagn tilknyttet en abstrakte datatype og dens operationer.

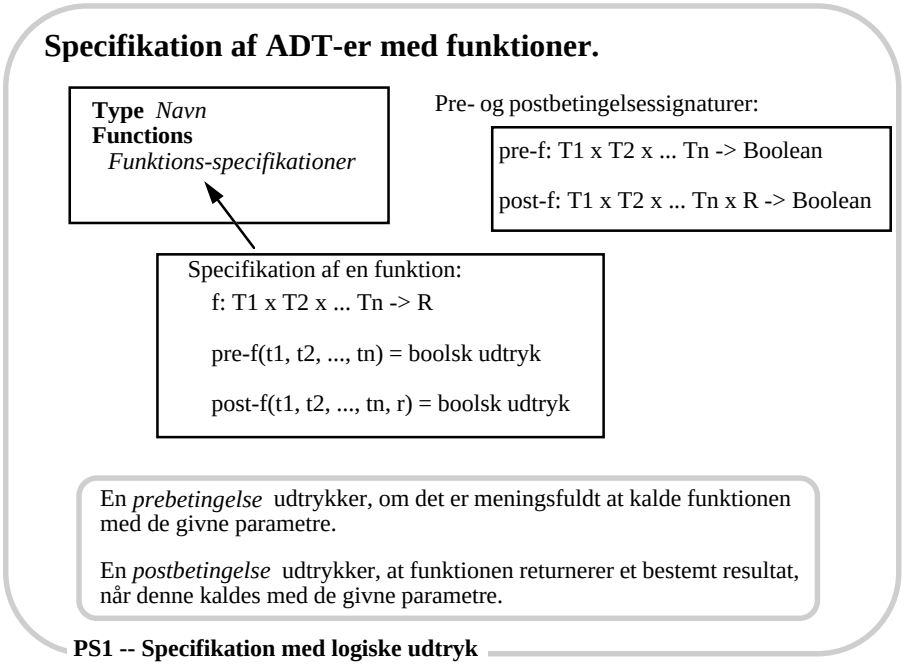
Denne slide minder læseren om, at specifikationerne, som vil blive behandlet i denne forelæsning, har samme grundlæggende egenskaber og formål som algebraiske specifikationer.

Hvorfor indfører vi en ny specifikationsform, når vi allerede har brugt tid og kræfter på at forstå een sådan formalisme (algebraiske specifikationer)?

1. I forbindelse med praktisk, objekt-orienteret programmering ønsker vi at kunne tilknytte specifikationen tæt til selve programmet. Der er ofte for stor afstand mellem elementerne i en algebraisk specifikation af en ADT og implementationen af en klasse, der realiserer typen. Én årsag er, at operationer i en algebraisk specifikation er rene funktioner, mens operationer i klasser ofte har sideeffekter på den af klassen beskrevne tilstand. Specifikation af abstrakte datatyper med logiske udtryk kan således bringes tættere til elementerne i implementationen af typen.

2. Der findes nogle datatyper, som mest hensigtsmæssigt specificeres algebraisk; og omvendt. Det betyder, at hvis vi har to forskellige specifikationsformer til vores rådighed, vil vi kunne vælge den mest hensigtsmæssige i en given situation.

## Specifikation af ADT-er med funktioner.




**PS1 -- Specifikation med logiske udtryk**



# Noter

Vi starter med at introducere specifikation af abstrakte datatyper, hvor alle operationer er funktioner, på samme måde som alle operationer i de algebraiske specifikationer, vi tidligere har set på, er funktioner.

Dette betyder, at operationer på datatypen ikke ændrer i objekters tilstand. Hvis vi er igang med at specificere typen T, kan vi have en operation f: T -> T, hvor f(t) er at betragte som en modificeret kopi af det objekt, f blev anvendt på.

I en praktisk objekt-orienteret programmeringssammenhæng vil vi i stedet for udtrykke os med t.f; operationen f kaldes på objektet t, som er af typen T; f kan have en eller anden effekt på tilstanden af det objekt, t refererer til.

Vi vil i denne forelæsning gradvis nærme os en situation, hvor vi kan bruge logiske udsagn til at specificere ADT-er i Eiffel. Dette omfatter også klasser, hvis instanser har tilstand, der muteres.

## Eksempler på specifikation af funktioner.

Subtraktion:

sub: Int x Int -> Int
pre-sub(x, y) = **true**
post-sub(x, y, r) = ( x = y + r)

Kvadratrod:

| **Eksplicit specifikation** | **Implicit specifikation** |
|---|---|
| sqrt: Real -> Real | sqrt: Real -> Real |
| pre-sqrt(r) = (r • 0) | pre-sqrt(r) = (r • 0) |
| post-sqrt(r, s) = (s = ¦r) | post-sqrt(r, s) = (s * s = r) |

Største fælles divisor:

gcd: Nat x Nat -> Nat
pre-gcd(x, y) = **true**
post-gcd(x, y, r) =
  is-common-divisor (x, y, r) **and**
  ¬ (∃ e **in** Nat: ((e > r) **and** is-common-divisor(x, y, e)) )

**PS1 -- Specifikation med logiske udtryk**



# Noter

I den eksplicitte specifikation af kvadratrodsfunktionen sqrt antager vi, at den matematiske kvadratrodsfunktion er kendt og veldefineret. I den implicitte specifikation antager vi "kun", at vi kender mulitiplikationsoperationen

Kvantorer i logiske udtryk tilføjer ekstra udtryksfuldhed i specifikationen.

**Opgave.** I specifikationen af gcd er der benyttet en anden funktion is-common-divisor: Nat x Nat x Nat -> Boolean. Intuitivt ønsker vi, at is-common-divisor(x, y, r) udtrykker om r går op i både x og y. Specificer denne funktion. I specifikationen af is-common-divisor benyttes måske en ny funktion, som bør specificeres. Diskuter hvor langt man skal gå med specifikationen af sådanne funktioner.

## Korrekthedsbevis af funktionsimplementation på basis af specifikation

Specifikation af en funktion:

f: T1 x T2 x ... Tn -> R

pref-f(t1, t2, ..., tn) = ...

post-f(t1, t2, ..., tn, r) = ...

Implementation af funktionen

```
f(p1: T1; p2: T2; ... pn: Tn): R is
--declarations
do
  --body
end
```

Relativ til implementation af f skal det vises at:

∀ t1 **in** T1, ∀ t2 **in** T2, ... ∀ tn **in** Tn:
pre-f(t1, t2, ... , tn) => post(t1, t2, ...., tn, f(t1, t2, ... tn))

For at kunne bevise dette formelt kræves, at der er formuleret bevisregler for alle de typer af konstruktioner, der benyttes i kroppen af f.

**PS1 -- Specifikation med logiske udtryk**



# Noter

Formålet med denne slide er at gøre klart, hvad det vil sige at bevise korrektheden af implementationen af en funktion ud fra en specifikation af funktionen.

Implikationen, som er kriteriet for, at funktionen er korrekt, udtrykker, at **hvis** prebetingelsen er sand **så** skal postbetingelsen være opfyldt på parametrene og det resultat, som funktions-implementationen returnerer.

For at gennemføre et sådant bevis minutiøst og overbevisende kræves, at man kender den præcise betydning af de kommandoer og udtryk, der indgår i funktionsdefinitionen.

På nuværende tidspunkt har vi ikke defineret semantikken af de enkelte Eiffel kommander på en formel matematisk måde. Derfor kan vi ikke for nærværende udføre beviser af korrektheden af en funktion. Vi vender imidlertid tilbage til bevisregler for udvalgte kommandoer senere på kurset.

Næste slide konkretiserer indholdet af denne slide i den situation, at vi vil vise, at en bestemt implementation af Fibonacci funktionen myfib er korrekt i forhold til specifikationen.

## Eksempel på specifikation, implementation og krav til korrekthedsbevis for en funktion.

**Givet funktionen fib:**

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n • 2

**Specifikation af myfib:**

myfib: Nat0 -> Nat0
pre-myfib(n) = true
post-myfib(n, r) = (r = fib(n))

**Implementation af myfib**

```
myfib(n: integer): integer is
local a: integer;
      b: integer;
      count: integer;
      br: integer
do
 from a := 1; b := 0; count := n;
 until count = 0
 loop
  br := b; b:= a;
  a := a + br; count := count - 1
 end;
 result := b
end; -- myfib
```

For at bevise, at implementationen af myfib er korrekt, skal man vise, at result = fib(n), når myfib terminerer.





# Noter

Specifikationen af myfib, og det vil i et og alt sige den rekursive defintion af fib, udgør grundlaget for et bevis for, at Eiffel implementationen af myfib er korrekt.

Som omtalt har vi endnu ikke introduceret bevisreglerne for assignment og loop kommandoerne i Eiffel, så vi kan endnu ikke bevise korrektheden af funktionen.

I en senere forelæsning vil vi beskrive semantikken af udvalgte Eiffel kommandoer. Vi vil vende tilbage til netop dette eksempel og gennemføre beviset for korrektheden af myfib i forhold til dens specifikation.

**Opgave:** Find en variant og en invariant af løkken i funktionen myfib. Løsningen findes i noterne til sidste slide i denne forelæsning. (Det anbefales, at man prøver på at formulere varianten og invarianten, inden man kigger).

## Assertions i Eiffelprogrammer.

Et Eiffel program kan tilknytte en række logiske udtryk, som vi vil benævne ‘assertions’.

Assertions udgør en programspecifikation. En specifikation udtrykker betingelserne for, at programmet er *korrekt.*

Assertions udgør også

- højniveau dokumentation af klasse i Eiffel
- understøttelse af lokalisering af årsagen til fejl (debugging)
- grundlaget for håndtering af undtagelsessituationer (exceptions).
- et redskab for definition af ansvarsfordeling mellem klasser.

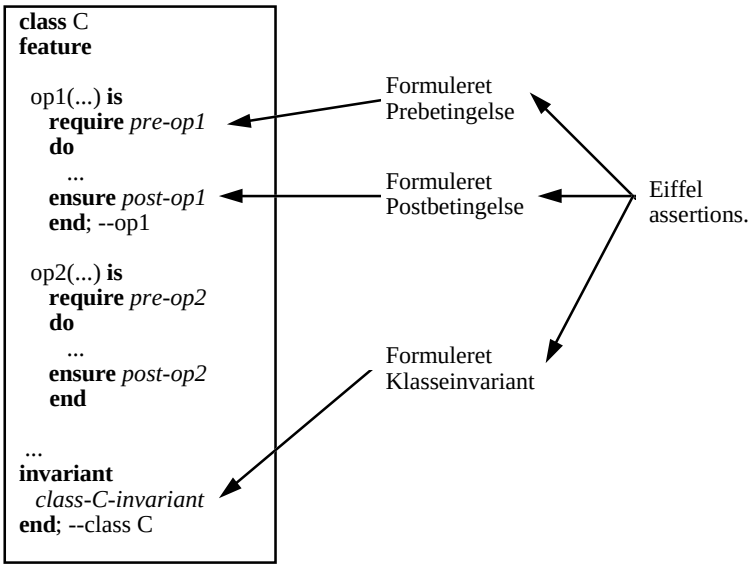




# Noter

Denne slide udtrykker nogle overordnede kvaliteter ved anvendelsen af assertions i et Eiffel program.

**Pre- og postbetingelser og klasseinvarianter i Eiffel.**




**PS1 -- Specifikation med logiske udtryk**



## Noter

Vi går nu over til at studere specifikation af Eiffel programmer.

Prebetingelserne ovenfor, såsom *pre-op1*, kaldes de **formulerede prebetingelser**. Den egentlige prebetingelse, som skal gælde for *op1* dannes ud fra den formulerede prebetingelse og invarianten. Vi vender tilbage til dette senere i kapitlet.

Ganske tilsvarende forhold gælder for postbetingelserne. Vi taler altså også om den **formulerede postbetingelse.**

Vi kalder invarianten, som angives nederst i en klasse-definition, for den **formulerede klasseinvariant.**

## Klasseinvarianter i Eiffel.

En klasseinvariant er et assertion, som udtrykker nogle generelle betingelser som alle objekter af klassen skal opfylde på bestemte tidspunkter i objeternes levetid.

Hvornår skal invarianten være opfyldt:

- Efter instantiering og initialisering (efter udførelsen af Create).
- Efter (og før) udførelse af en eksporteret operation, som overholder sin pre-betingelse.

Typisk brug af klasseinvarianter:

- En klasseinvariant udtrykker typisk betingelser for konsistens mellem attributværdier og funktionsværdier i en instans af klassen.





# Noter

Invarianten skal altså være opfyldt efter instantiering/initialisering samt efter hver ekstern operation på typen.

Man kan spørge om dette implicerer, at invarianten er opfyldt før hver operation på typen. Dette er tilfældet i simple situationer. Der kan dog laves invarianter, som kan ændres udefra, uden at der bliver kaldt en operation på typen. Det er tilfældet, hvis invarianten involverer udtryk, som kan påvirkes fra andre klasser. 7.11.3 i OOSC handler om dette.

Vi vil derfor også kræve, at invarianten er opfyldt, inden en operation udføres.

Invarianter kendes også fra vores ikke-tekniske hverdag. Det er f.eks. en regel, at kanden på afdelingens kaffemaskiner altid skal indeholde mindst een krus kaffe. Alle transaktioner på kaffemaskinen skal overholde denne invariant. Bemærk at handlingerne, der består i at tømme kaffekanden samt at lave ny kaffe er at betragte som én transaktion i denne model. På et tidspunkt er kanden tom, men da dette tidspunkt ikke forekommer mellem to (eksterne) transaktioner på maskinen, bryder vi ikke invarianten.

Ved forelæsningen vil vi se på en klasse 'Kunde', som indeholder navne og adresseattributter, og vi vil diskuterer forskellige konsistensforhold i objekter af denne klasse.

## Eksempel på klasseinvariant.

```
class BANK_ACCOUNT creation
  create
feature
  balance, interest_rate : REAL;
  interest_rate: real;

  create(initial_amount: REAL)  is
  do
    transactions.create(100);
     balance := initial_amount;
    transactions.put(initial_amount)
  end;

  deposit(amount: real) is do ... end;

   withdraw(amount: real) is do ... end;

  add_interest(days: integer) is do ... end;
```

```
feature {NONE}
  transactions: TRANSACTION_HISTORY;

  add(amount: real) is
   do
    transactions.put(amount);
     balance := balance + amount
   end

invariant

   balance = transactions.add_together;

end; -- class BANK_ACCOUNT
```





# Noter

Eksemplet viser den velkendte klasse BANK_ACCOUNT med en transaktionshistoie. Invarianten udtrykker en vigtig konsistensbetingelse mellem attributten 'balance' og den lagrede transaktionshistorie.

## Eksempel på klasseinvariant (fortsat).

```
Class TRANSACTION_HISTORY
creation create
feature{NONE}
  history: ARRAY[TRANSACTION];
  next_free, high_index: INTEGER;

feature
  create(max: INTGER) is
  do
    high_index := max;
    history.create(1,max);  next_free := 1;
  end;

  put(amount: REAL) is
  require not full
  local tr: TRANSACTION
  do
    tr.create(amount);
    history.put(tr, next_free);
    next_free := next_free + 1
  end;

  get(i: integer): TRANSACTION is
  require i < next_free;  i >= 1
  do
    result := history.item(i)
  end;

  full: BOOLEAN is
  do
    result := next_free > high_index
  end;

  add_together: REAL is
  do ... end;

end -- class TRANSACTION_HISTORY
```

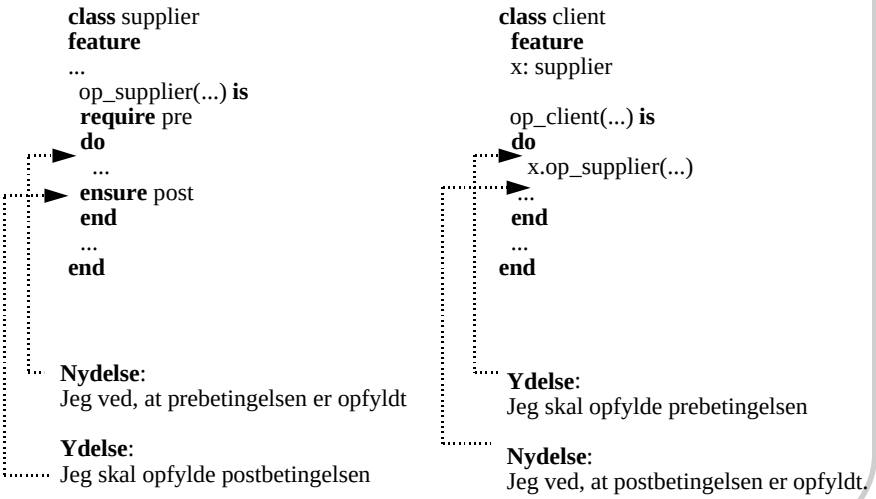
**PS1 -- Specifikation med logiske udtryk**



## Noter

Denne slide viser en skitse af klassen TRANSACTION_HISTORY, som blev anvendt i eksemplet på forrige side.

## Kontrakt mellem en klasse og dens klienter.




**PS1 -- Specifikation med logiske udtryk**



# Noter

På denne slide (og nedenfor) betegner prebetingelsen den effektive prebetingelse. Ditto med postbetingelsen.

De effektive pre- og postbetingelser udgør en *kontrakt* mellem en klasse og dens klienter.

En operation i klassen lover at opfylde post-betingelsen såfremt klienten kalder operationen med opfyldt prebetingelse.

Kontrakten er udtryk for en klar *ansvarsfordeling* mellem klient og leverandøren. Det er klientens ansvar at undersøge, om prebetingelsen af supplier-operationen er opfyldt. Hvis dette ikke er tilfældet, bliver supplier-operationen ikke kaldt, og der opstår en fejlsituation i klienten.

Det er således ikke nødvendig i kroppen af en operation i leverandør-klassen at teste, om operationens pre-betingelse er opfyldt. *Den skal være opfyldt.*

Hvis supplier-operationen ikke opfylder sin post-betingelse opstår der en fejlsituation i supplier klassen.

Denne klare ansvarsfordeling, og deraf følgende minimal, eksplicit programmet test af betingelser forud for udførelsen af en operation, står i skarp kontrast til en programmeringsteknik, som kaldes *defensiv programmering.* I defensiv programmering går man både med livrem og seler i den forstand, at man gerne udfører samme test (f.eks. af prebetingelsen) to forskellige steder i programmet, for at forhindre misforståelser om ansvarsfordelingen.

I og med at opfyldelse af prebetingelsen er klientens ansvar, skal alle dele af en prebetingelse være tilgængelig for klienterne. Dette betyder, at der ikke må indgår hemmelige (private/usynlige) deltryk i en prebetingelse. Hvis der var dele af en prebetingelse, der var hemmelige, ville man kun vanskeligt kunne laste klienten, at en prebetingelse ikke kunne opfyldes (Klienten kunne sige: “Det er meget godt, men jeg har ikke en chance for at kende til de detaljer, som prebetingelsen udtaler sig om”).

I en postbetingelse må der derimod godt indgår private aspekter fra en klasse. Når postbetingelsen annonceres til klienter i dokumentationsøjemed (af værktøjet short), ser man bort fra de dele af en postbetingelse, der indeholder private elementer.

## Formen af assertions i Eiffel.

Assertion: en konjunktion ('and then'-ing) af enkelt-assertions.
Enkelt-assertion: navngiven eller
unavngiven assertion-klausul
Assertion-klausul: Boolsk udtryk eller
Uformel assertion.

**Eksempel:**

```
min_betingelse: i • 0;
max_betingelse: i < max;
--  for alle i mellem 0 og max gælder p(i)
```

**Post-betingelser:**

Assertion som beskrevet ovenfor med følgende udvidelser

**old** exp ← exp beregnet med hensyn til objektets tilstand før kald af operationen.

equal(**strip** (), old **strip** ()) ← Sand hvis og kun hvis objektets tilstand er uforandret ift. før kaldet.

**PS1 -- Specifikation med logiske udtryk**



# Noter

I eksemplet midt på denne slide vises der en konjunktion af tre enkelt-assertions, hvoraf de to første er navngivne, og det sidste er uformelt angivet (som en kommentar: -- ...).

Navnene på del-assertions er uden betydning for det samlede assertion, og de har ingen betydning for semantikken af programmet. Navnene tjener til fornuftig kommunikation mellem køretidssystemet (når det opdager en fejl under programudførelse) og testkøreren.

Assertions der består af kommentarer spiller udelukkende en rolle som dokumentation.

Bemærk at

**old** expr

kun må forekomme i en postbetingelse.

Det noget mystiske udtryk

equal(strip(), **old** strip())

anvender et primitiv

strip(a, b, ...)

som vi endnu ikke har set på. Kort fortalt returnerer det et array af alle felter i det nuværende objekt, pånær felterne med navnene a, b, etc. Specielt returnerer

strip()

et array af alle felter (i en eller anden bestemt rækkefølge) af det nuværende objekt. Ovenstående udtryk siger dermed, at objektets tilstand er uforandret af den operation, hvoraf vi nu studerer postbetingelsen.

Strip er et generelt tilgængelig primitv, som dog klart snævert er designet mod ovenstående anvendelse.

Strip er beskrevet i afsnit 23.21 og 9.10 af "Eiffel the Language".

## Specifikation af klasser i Eiffel.

Sammendrag

**Prebetingeler, postbetingelser og invarianter.**
Eiffel understøtter pre- og postbetingelser på operationer samt invarianter i klasser.

**Mutation af programtilstand.**
Nogle operationer i en ADT i Eiffel muterer (ændrer) tilstanden af programmet. Ikke alle operationer er funktioner.

**Check af assertions.**
Assertions i Eiffel skal kunne verificeres under programudførelsen, uden massiv påvirkning af udførelsestiden af programmet. Dette udelukker udsagn med eksistens- og alkvantorer.

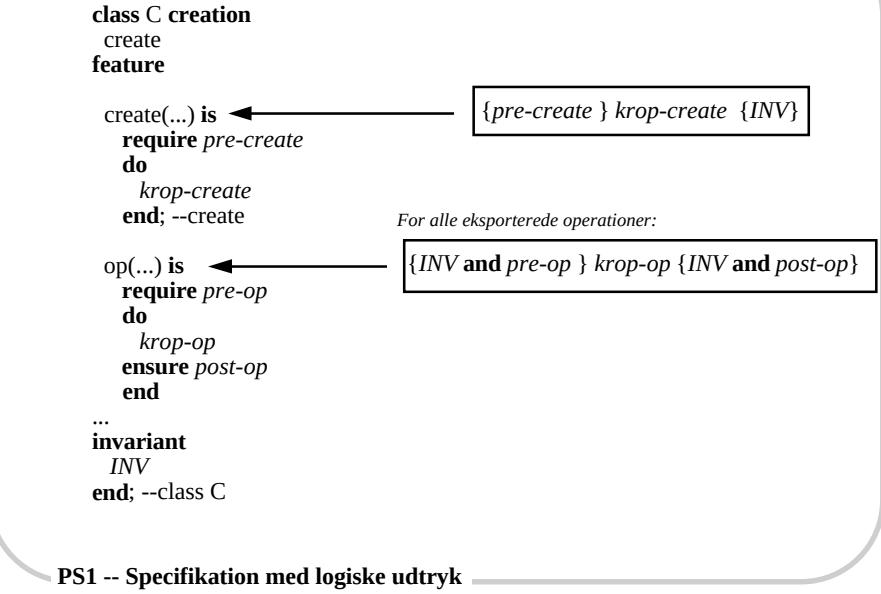
**Uformelle assertions.**
Pre- og postbetingelser er ikke nødvendigvis formelle. Man kan skrive uformelle pre- og postbetingelser, som naturligvis ikke kan verificeres under programudførelse.





Noter

## Korrekthedskriterium for Eiffel-klasser.




**PS1 -- Specifikation med logiske udtryk**



# Noter

Korrekthedskriteriet udtrykker betingelserne for, at implementationen af klassen opfylder den givne specifikation.

Notationen

{assertion1} *kommandoer* {assertion2}

er en  såkaldt Hoare-sætning, som udtrykker:

Hvis assertion1 er opfyldt før udførelsen af *kommandoer*, og hvis *kommandoer* terminerer, da skal assertion2 være opfyldt efter udførelsen af *kommandoer*.

I forbindelse med programbeviser i del 2 af noterne vil vi stifte nærmere bekendtskab med denne notation.

Korrekthedskriteriet for Create læses:  Hvis Create's prebetingelse er opfyldt, så skal klasseinvarianten INV være opfyldt, efter at kroppen af Create er udført. Man kunne godt vælge at styrk prebetingelsen med et udsagn der udtrykker default initialiseringen af felter, som foretages af Eiffel. Man kunne også godt vælge at medtage en eksplict postbetingele af creation proceduren, som dermed ville kunne styrke INV.

Tilsvarende for korrekthedskriteriet for andre operationer end Create: Hvis invarianten og prebetingelsen er opfyldt inden udførelsen af operationen,  da skal postbetingelsen og invarianten være opfyldt efter udførelse af operationen.

Konjunktionerne af de formulerede pre/postbetingelser og klasseinvarianten udgør det vi kalder de *effektive pre/postbetingelser*.

(Noterne fortsættes på flg. sides noteark).

## Hvis assertions ikke overholdes...

Hvis en assertion brydes:

- Programmet kommer i en **undtagelsestilstand**.
- Programmet får chancen for at håndtere tilstanden:
  - Undtagelsestilstanden afblæses, og programmet fortsætter *eller*
  - Programmet terminerer, og undtagelsessituationen beskrives.





# Noter

Det er relevant at spørge, om invarianten skal checkes i følgende situation:

```
class C
feature
  op1(...) is
  do ...; op2(...); ...
  end;

  op2(...) is
  do ...
  end
invariant
  INV
end; --class C
```

Vi antager at op1 kaldes fra en klient af klassen C. Spørgsmålet er hvorvidt invarianten skal checkes når op1 kalder op2 på current object. Det fremgår, at når op2 kaldes fra op1 kan op2 opfattes som en intern operation, som sammen med de øvrige handlinger i op1 skal tilfredsstille invarianten; det er ikke nødvendigt at op2 i denne situation selv tilfredsstiller invarianten.

Det er ikke let at finde nogen dokumentation af denne situation i Eiffel manualerne, og man kan blive i tvivl om præcist hvornår invarianten skal checkes. En testkørsel afslører, at vores installation ikke checker invarianten, når op2 kaldes fra op1 (Eiffel 2.3 resultat).

**Løsning til opgave**

Løsning til opgave på slide: "Eksempel på specifikation, implementation og krav til korrekthedsbevis for en funktion":

Varianten kan være count, idet denne variabel tælles ned for hvert gennemløb, og den bliver aldrig negativ.

Invarianten kan være

a = fib(n - count + 1); b = fib(n - count);

count • 0; count Š n

Bemærk, at invarianten er opfyldt før udførelsen af løkken, idet count her har værdien n, a er 1, b er 0, samt fib(1) = 1 og fib(0) = 0, jf. definitionen af fib.

Bemærk også, at ved termineringen af løkken er count = 0, hvilket vil sige, at b = fib(n). Sidstnævnte er i alt væsentlighed postbetingelsen af myfib.