Begrunnelser for begrensninger i Simula

Og: Kan noen av dem lettes på?

Stein Krogdahl, Ifi, UiO, til møte 15. mars 2019

Den Simula-standarden dette notatet forholder seg til er:
«SIMULA Standard, as defined by the SIMULA Standards Group, 25th August 1986»
Denne kan hentes fra: http://simula67.at.ifi.uio.no/Standard-86/

Vi skal her se nærmere på to begrensninger i Simula-språket, nemlig

(A) Man må definere subklasser til klassen C på samme «blokknivå» som klassen C

(B) Man får ikke lov å «dotte» seg inn i klasser med lokale klasser.

Problemstillingen er om disse begrensnigene kan lettes på eller kanskje fjernes helt.

A: Alle subklasser på samme blokknivå som superklassen

Merk her at dette ikke er en så stor restriksjon som man i første omgang kanskje kunne tenke seg. Man kan jo nemlig ha:

```
begin
    class A;
    begin
        class B; begin … end B;          ! På blokknivå 2 (avh. av def. av «blokknivå») ;
        …
    end A;

    A class AA;
        B class BB; begin … end BB;      ! Også på blokknivå 2 ;
        …
    end AA;
end
```

Restriksjonen betyr altså ikke at en klasse og alle dens subklasse må ligge i samme tekstlige begin…end-blokk, og dermed har man tross alt en rimelig fleksibilitet. Konstruksjonen med klasser i klasser er spesielt viktig når man har eksterne klasser som skal brukes som blokkprefiks, slik som «Simset» og «Simulation». Anta f.eks. at vi har følgende eksterne klasse «DomainConcepts» som inneholder klasser og opersjoner som et greit grunnlag for å bygge programmer innen den aktuelle «domain»:

```
class DomainConcepts;
begin
    ! ... Diverse klasser fra «concept area»:  ConceptA, conceptB, ... ;
end;
```

Da kan vi lage et program slik:

```
DomainConcepts begin
    ! ... Subklasser av klassene ConceptA , ConceptB..., samt nye klasser m.m. ;
end
```

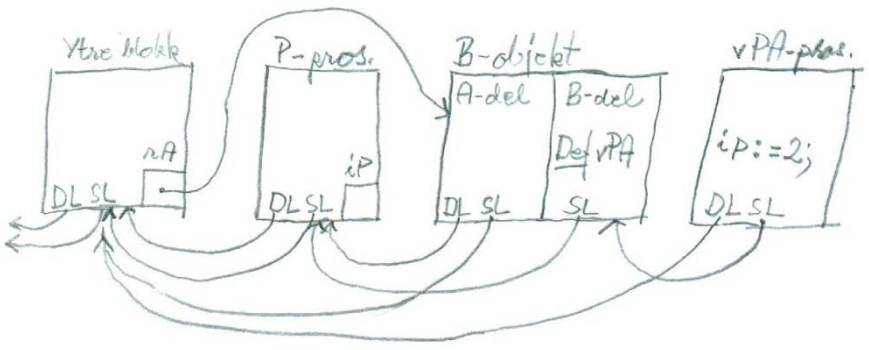
Regelen om at en klasse og alle dens subklasser skal ligge på samme blokknivå sier er altså at man ikke får lage programmer som f.eks. dette:

```
begin
    class A; virtual: procedure vPA; begin ... end A;
    ref(A) rA;

    procedure P;
    begin
        integer iP;
        A class B;
        begin
            procedure vPA; begin   iP:= 2; ...  end vPA;    !  B's versjon av den virtuelle vPA ;
            ...
        end B;
        rA:- new B;
    end P;

    P;          ! Dette kallet vil altså utføre «rA:- new B» ;
    rA.vPA;   ! Se i teksten for diskusjon av problemet her  ;
end
```

Problemet blir her tydelig ved kallet «rA.vPA;» ved at den statiske omgivelse til dette kallet innenfra og ut er som følger:

0: Selve kallet på vPA (altså, versjonen i B)
1: B-delen av B-objektet pekt på av rA
2: Kallet på P
3: Ytterste program-blokken

Men Simula (i likhet med Algol) foreskriver eksplisitt at aktivitets-blokken til en prosedyre kan/skal «kastes» når kallet terminerer. Det vil si at ikke alle elementene i den statiske omgivelsen til vPA-kallet vil være tilgjengelige mens kallet utføres, og at f.eks. aksess av den «synlige» variablen iP kan bli bare tull.



Situasjonen i Java

Spesielt blir dette ille i språk som Java, der man bruker en eksplisitt stakk til prosedyrekall (mens de i Simula legges på heapen pga. kvasiparallelliteten). I Java kan derfor kallet på P i den statiske omgivelse til vPA være overskrevet av et annet kall når vPA kalles. Java tillater imidlertid at man har subklasser på dypere blokknivåer enn superklassen enten ved klasser i klasser, eller ved (anonyme) klasser inne i prosedyrer/metoder. Men de løser dette rett og slett ved å forby aksess til variable som «iP». Altså, det er klare restriksjoner på hva man kan aksessere fra slike subklasser på dypere nivåer, og disse gjør at slike situasjoner ikke kan oppstå (håper jeg(!), men jeg er ikke ekspert i disse detaljene i Java).

Man kunne forsøke seg med liknende begrensninger i Simula, og dette var faktisk på tale den gang Simula ble definert. Den reglen vi nå har ble kalt «Rule S», mens alternativet som var på tale ble kalt «Rule R» og den var slik:

> *A reference assignment is legal only if the left hand quantity is declared in the scope of the class qualifying the right hand side and all its subclasses.*

Ut fra «Rule R» ville altså tilordningen «rA:- new B» være ulovlig siden subklassen B ikke var synlig der variablen rA ble definert. Men man valgte altså til slutt å innføre «Rule S» i Simula, siden den tross alt er enklere å forholde seg til og enklere å teste i kompilatoren, og er heller ikke voldsomt begrensende i vanlig programmering.

Betas løsning

En annen mulighet kunne være å gjøre som i Beta, der man tillater at en subklasse (eller «subpattern») defineres på et hvilket som helst nivå bare superklassen er synlig. For at dette skal bli konsistent kutter Beta ut regelen om at aktivitetsblokken til et prosedyrekall skal «kastes» etter at prosedyren teminerer. Dette faller naturlig i Beta, siden både klasser og prosedyrer dekkes av begrepet «pattern», og dermed er det ikke noen naturlig definisjon av når en instans av et pattern skal anses som en prosedyre (og dermed eventuelt skal «kastes» ved terminering), og når den skal anses som et objekt som skal bevares.

I Beta kan man til og med angi at superklassen skal være en lokal klasse inne i et annet objekt, og at superklassen derved kan angis ved «dotting» inn i dens omliggende objekt. Hva dette innebærer i forhold til transplantasjon (se senere) er meg ikke helt klart, men vi går ikke videre på dette her. Det blir mye som figuren på side 2.

Om vi forsøker å overføre Betas blokknivå-regler til Simula må vi tillate at dataene til et prosedyrekall kan leve etter at kallet er terminert, og at de da fremdeles kan ha SL og DL til seg som GC'en skal ta hensyn til, slik at kall-dataene bevares. Merk da at i et slikt regime vil hvert «subklassenivå» i et objekt ofte ha sin egen SL, og dette må tas hensyn til i all ikke-lokal aksess f.eks. av variable (men andre løsninger er også mulige). Dette er et løsning det ikke vil være håpløst å implementere også i Simula, og den ville altså gjøre at prosedyre-kall blir mer lik objekter, ved at hele kall-blokken ofte vil leve videre etter at kallet er terminert. Jeg har ikke full oversikt, men med tanke på alle invariantene som skal gjelde for kvasiparallell utførelse i Simula er jeg dog ikke sikker på at dette vil virke greit i Simula!! Mye måtte nok da tenkes gjennom på nytt, og kanskje redefineres.

Kan Simulas mekanisme for blokk-prefiks forbedres?

Her tar vi en avstikker til et annet tema som naturlig kan nevnes i denne sammenheng, og der en regelendring ville være av stor interesse. Det gjelder det som var nevnt over om at man kan la klassenavn være «prefiks» til en blokk (slik som klassen DomainConcepts nevnt lenger opp). Dette får den effekt at de lokale klassene inne i DomainConcept-klassen vil ligge klare inne i den prefiksede blokken, og dermed utgjøre et grunnsett av klasser og operasjoner som gjør det greiere å lage programmer innen den aktuelle «domain», og det er da viktig at man også kan lage nye subklasser av disse. Husk her at DomainConcept-klassen eventuelt skal ha aktuelle parametere når den brukes som prefiks til en blokk!

Men ofte er man interessert i å ta utgangspunkt i flere slike DomainConcept-klasser (la oss kalle dem hhv. DC1 og DC2), for så å kunne utvikle sitt eget program i en omgivelse der man fritt kan lage subklasser av alle deres indre klasser, og i disse greit kunne referere til alle disse nye subklassene på kryss og tvers. Dette går dessverre ikke an slik Simula er definert i dag siden det bare er lov med ett prefiks på en blokk. Om man vil bruke flere DomainConcept-klasser, f.eks. DC1 og DC2, så må man eventuelt gjøre det slik:

```
DC1 begin
   DC2 begin
        ! Her ønsker man rimeligvis å kunne bruke klassene og operasjonene som er
          definert inne i  DC1 og DC2, også til å lage nye subklasser av disse. Men
          subklasser av lokale klasser i DC1 må, på grunn av blokknivå-reglen defineres
          allerede før «DC2 begin», og de for DC2 må vente til etter dette. Det er også
          viktig at innmaten av disse subklassene  må kunne referere til alt som er synlig
          inne i «DC1 begin» og «DC2 begin»-blokkene, altså også alle de andre
          nydefinerte subklassene.  Men det går da selvfølgelig ikke.
   end DC2;
end DC1
```

Men det ønsket som er angitt i kommentaren over er altså ikke mulig å oppfylle innen dagens Simula på grunn av blokknivåreglen. Den gjør at vi ikke kan lage subklasser både av de lokale klassene i DC1 og de i DC2 i samme blokk, og dermed kan ikke subklassenes definisjoner referere fritt til hverandre. Subklasser av de lokale klassene i DC1 må eventuelt defineres før man går inn i «DC2 begin»-blokken.

Men dette bør forholdsvis lett kunne rettes på. Man kan f.eks. i første omgang rent syntaktisk tillate flere prefiks foran en blokk, altså f.eks. slik:

```
DC1, DC2 begin    ! NB: Husk at eventuelle aktuelle parametere på DC1/DC2 skal med her!  ;
   ...
end
```

Inne i denne blokken kunne man så uproblematisk tillate at det lages subklasser både av de lokale klassene i DC1 og de i DC2, slik vi ønsket over. Grunnen til at noe slikt ikke ble innført i Simula fra start var nok at man ville holde på prinsippet om ett prefiks også for blokker når dette var regelen for klasser. For klasser er dette også nokså komplisert å implementere,

men flere prefiksklasser til en blokk kan nokså greit implementeres (med hensyn til aksess langs SL etc.) ved å «anta» at det står:

```
DC1 begin
   ! Ingen nye deklarasjoner her!  ;
   DC2 begin
      ...
   end DC2;
end DC1
```

Vi vil her vite at det ikke finnes noen nye deklarasjoner i den ytterste blokken, og derved vil det hele faktisk tilfredstille «Rule R» (diskutert over), slik at det å definere subklasser av lokale klasser fra DC1 inne i DC2-blokka er helt problemfritt. I kompilatoren kan man kontrollere dette ved å gjøre forskjell på «blokknivået for å gjøre aksess ved run-time» og «blokknivået for å sjekke at subklasser ikke bryter blokknivå-regelen».

Man kunne også prøve å dra dette enda lenger f.eks. ved å gjøre en syntaktisk forskjell på «vanlige» klasser ment for objekt-generering og klasser ment for prefiksing av blokker. Vi kunne kalle de siste for «ramme-klasser», og bruke nøkkelordet «frameclass» i stedet for «class». Klassene Simset og Simulation ville da typisk være slike klasser. Fordelen med dette måtte da være at man fikk lov til å ha flere prefiks-klasser (multippel arv) for ramme-klasser, altså f.eks. slik:

```
frameclass FC1; begin ... end;
frameclass FC2; begin ... end;
FC1, FC2 frameclass FC3; begin ... end;
frameclass FC4; begin ... end;
FC3, FC4 begin ... end;
```

Uten å ha helt oversikten mener jeg denne slags multippel arv ville være lettere å implementere da man aldri vil lage vanlige objekter av disse klassene, og dermed heller ikke ha pekere til dem. En måte kan være å spandere ny kompilering (som kan gi nye relativ-adresser til attributtene i prefiksklassene) hver gang man definerer en ny rammeklasse, og dette kunne kanskje også optimaliseres en god del om man ser nærmere på det.

Men merk at man her også måtte ta inn over seg «diamond»-problemet, altså det som ville oppstå om vi f.eks. forandret definisjonen av FC4 til «FC1 fameclass FC4; begin ... end;» slik at FC1 ville komme inn i den prefiksede blokken via to veier. Det er da nærliggende å se disse som helt separate utgaver av FC1, men man får i så fall lett navnekollisjoner som programmereren måtte få midler til å løse.

Mulig kompromiss: En fornuftig mulighet kunne være å innføre flere prefiks på blokker, men ikke tillate noe i retning av «frameclass». Jeg tror allerede dette kunne gi signifikant større frihet i bruk av ferdig-programmerte moduler i Simula.

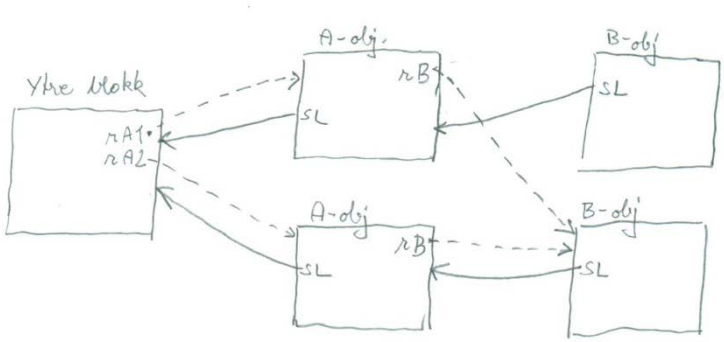
B: Om å «dotte» seg inn i klasser med lokale klasser

Simula har altså nå en regel om at det ikke er lov å «dotte» seg inn i objekter med lokale klasser. I "*SIMULA Standard*" er dette formulert ved at konstruksjonen «X.A», der X er typet med klassen C, bare er lovlig dersom: «*The object referenced by X has no class attribute declared at any prefix level equal or outer to that of C.*»

Transplantasjon

Grunnen til denne reglen er å unngå såkalt «transplantasjon», som vi får f.eks. i følgende (ulovlige!) program:

```
begin
    class A;
    begin
        class B;
        begin
            ...
        end B;
        ref(B) rB;

        rB:- new B;
    end A;
    ref(A) rA1, rA2;

    rA1:- new A;   ! Her blir et nytt B-objekt laget, av B-klassen i det ene A-objektet  ;
    rA2:- new A;   ! Og her blir ett til lage, men av B-klassen i det andre A-objektet    ;
    rA1.rB:- rA2.rB;     ! Ulovlig!  Her får vi TRANSPLANTASON. Se i teksten under ;
end
```



Simula insisterer altså på at vi i programmet over får to separate utgaver av klassen B, nemlig én som er lokal i det ene A-objektet og én som er lokal i det andre. Det å ikke holde disse klassene klart fra hverandre kalles «transplantasjon», og regnes i Simula som en feil. Typisk ville det oppstå en uklar situasjon dersom man fra A-objektet referert av rA1 nå gjorde et kall «rB.p», der p er en metode i klassen B. Dette metodekallet ville da få SL-kjede gjennom det ene A-objektet og DL-kjede gjennom det andre. Om og når dette vil slå ut i en

inkonsistent situasjon senere under kjøringen vil imidlertid avhenge av en rekke faktorer, som hva men gjør videre, samt detaljer i hvordan runtime-systemet er kodet.

Uten forbudet mot dotting inn i objekter med lokale klasser kunne vi altså fritt skive «rA1.rB» og «rA2.rB», men disse ville være typet med to forskjellige klasser og tilordningen over vil derfor være ulovlig ut fra vanlige Simula-regler. Det ubehagelige er at dersom vi tillater slik dotting, så er det i *kompilatoren* ikke mulig å finne de tilordningene som dermed er ulovlige, og det å teste for dette under *utførelsen* ville ta uforholdsmessig mye tid. For å slippe dette problemet laget man derfor (i 1967) en ganske «grov» regel, nemlig at man i det hele tatt ikke får lov til å dotte seg inn i en klasse med lokale klasser.

Kan inspect-setningen gi problemer her

Regelen omtalt over sikrer langt på vei at transplantasjon ikke forekommer. Simula har imidlertid en inspect-setning som har en liknende effekt som dot-notasjon. Om variabelen rC er typet med klassen C, rC ikke er «none» og man sier «inspect rC do begin … end», så kan man inne i «begin … end» aksessere attributtene i objektet rC direkte, som om man var inne i objektet. Og det er ingen regel mot å «inspisere» objekter med lokale klasser, og lokale klasser i objektet kan også aksesseres på linje med andre attributter. Om det ikke er andre regler kan dette også føre til transplantasjon, ved f.eks. å gjøre to inspect'er inne i hverandre, slik som i programmet under. Setningen "inspect rA do begin rB2:- rB1 end»

```
begin
    class A;
    begin
        class B;
        begin
            ...
        end B;
        ref(B) rB1;
        rB1:- new B;
    end A;

    A class AA;    ! Vil ha et navn (nemlig «rB2») i rAA som ikke skygges for av navn i rA. ;
    begin
        ref(B) rB2;
        rB2:- new B;
     end AA;

    ref(AA) rAA;
    ref(A)   rA;

    rAA:- new AA;  ! Utfører "rB2:- new B" inni rAA  (også "rB1:- new B", men uviktig her) ;
    rA:- new A;      ! Utfører "rB1:- new B" inni rA      ;
```

```
    inspect rAA do
    begin
      inspect rA do
      begin
        ! rB1 er nå typet med B i rA mens rB2 er typet med B i  rAA, altså forskjellige typer ;
        rB2:- rB1;     ! Dette gir derfor transplantasjon  ;
      end;
    end;
end
```

Simula-definisjonen sier (så vidt jeg forstår den) ikke noe om at denne og liknende konstruksjoner er ulovlige (men andre er uenige, og mener at blokk-nivå betraktninger gjør at tilordningen er ulovlig). Hvordan dette behandles i Simulakompilatorer varierer. Noen slipper det gjennom uten feilmelding, noe som altså skaper transplantasjon og fare for senere inkonsistenser, mens andre sier at interne klasser som blir synlige i hver av de to inspectene alltid skal anses som forskjellige klasser allerede i kompilatoren (selv om det tilfeldigvis *kan* være samme objektet som inspiseres under utførelsen).

Et problem her er at det, litt avhengig av designen av kompilatorens deklarasjonstabeller), ikke er greit å gjøre en rett fram test av dette. Den kompilatoren Birger og jeg laget på NR rundt 1980 gjorde det med nokså komplisert kode som inkluderte kopiering av tabellverk. Også andre gjør det, f.eks. IBM 360/370-kompilatoren, men jeg vet ikke hvordan. Jeg er nok tilhenger av at en slik test bør gjøres, men at den ikke gjøres er neppe noe problem i praksis.

Mulige lettelser i reglen som forbyr dotting inn i objekter med lokale klasser

Men la oss se bort fra de problemene som inspect-setningen lager, og gå tilbake til forbudet mot dotting inn objekter med lokale klasser. Det ville vært flott å kunne løsne i hvert fall litt opp i dette forbudet, og jeg ser to veier som kanskje kunne føre til noe:

- Man undersøker om det holder å begrense forbudet mot dotting til bare å gjelde aksess av attributter som har noe med de lokale klassene å gjøre. Ellers er det fritt fram.

- Man undersøker nærmere om transplantasjon egentlig er så ille. Kan man kanskje i programmene over betrakte klassen B som den samme, selv om den er lokal i forskjellige objekter av klassen A? Finnes det programmer der transplantasjon virkelig fører til inkonsistente situasjoner? Merk at B-klassene i alle A-objekter i en implementasjon normalt er representert av samme deskriptor (prototyp), og at noe annet ville bli veldig tungvint!

Kan vi begrense forbudet mot å dotte seg inn i lokale klasser?

Dette punktet har en liten forhistorie som jeg faktisk selv er deltaker i, og som jeg ikke kan dy meg for å skissere her. Det var i begynnelsen av 80-åra da Birger og jeg arbeidet på NR med front-enden til den Simula-kompilatoren som SIMULA AS siden bygget videre på. Jeg syntes da at kravet om å ikke kunne dotte seg inn i objekter med lokale klasser egentlig var unødvendig strengt, og jeg fremmet et forslag for SSG om å forandre denne regelen slik at

X.Y bare var ulovlig dersom typen (kvalifikasjonen) til X var en klasse med lokale klasser OG at attributtet Y var typet med en av disse lokale klassene.

Jeg nevnte også dette forslaget for Ole-Johan, og han mente med en gang at det måtte kraftigere lut til for å forhindre transplantasjon, i det minste noe om parametere til prosedyrer. Jeg fortet meg derfor med å trekke forslaget tilbake, og hadde tenkt å studere saken ordentlig før jeg eventuelt fremmet et bedre forslag. Men, som mye annet, kokte dette bort i andre gjøremål, men jeg mener å huske at jeg et år eller to senere, til min store overraskelse, oppdaget at SSG faktisk hadde vedtatt mitt opprinnelige forslag! Men jeg ser ikke noe spor av dette i dagens Simula-standard, så kanskje misforstod jeg noe her.

Så, er det mulig å forbedre mitt opprinnelige forslag til noe som faktisk virker? Ved å ta hensyn til Ole-Johans kommentar kunne vi forsøke med følgende regel:

> *Dersom X er typet med en klasse med lokale klasser så er X.Y bare lovlig dersom Y ikke er typet med noen av de lokale klassene i X eller superklasser av disse, og (om Y er et prosedyrekall) Y ikke har parametere som er typet med noen slik klasse.*

Jeg har forsøkt å finne situasjoner der dette ikke er nok til å forhindre transplantasjon, men jeg har ikke klart å finne noen slike. Jeg har imidlertid ikke noe slags bevis for at slike situasjoner ikke kan oppstå, og føler at saken absolutt må undersøkes nærmere før man eventuelt kan være sikker!

Må vi egentlig forhindre transplantasjon?

La oss så se på om transplantasjon egentlig er så farlig. Kanskje er det OK å si at om C er en klasse med en lokale klasse L, så skal vi anse klassen L i to forskjellige objekter av klassen C (eller subklasser) som samme klassen.

Om vi i første omgang ser bort fra kvasiparallell sekvensering er det (i hvert fall for meg!) ikke lett å finne programmer som ville føre til *virkelig* inkonsistente situasjoner, som altså bryter med noe *mer* enn det filosofiske prinsipp at lokale klasser i ett objekt alltid er forskjellige fra lokale klasser i et annet objekt. Man må jo da tåle situasjoner som de merket med «transplantasjon» i programmene over, og så se på fortsettelser av disse programmene som gir dypt inkonsistente situasjoner. Man må altså akseptere at man i stor skala får situsjoner der SL-kjeden og DL-kjeden fra et objekt eller prosedyrekall tar helt forskjellige veier, og det virker jo nokså skummelt.

Hva med kvasiparallelliet?

Om vi så tar inn kvasiparallellitet, altså «detach», «call» og «resume», så blir situasjonen kanskje verre? Jeg har nå lest kapittel 7 i standarden (om sekvensering) og det som der står er ikke uten videre lett å forholde seg til. Kapittelet består grovt sett av to deler, nemlig en første del som sier hvordan et konsistent system skal være. Det angis altså et antall «invarianter» (eller «konsistenbetingelser») som presumptivt ikke skal ødelegges av noen sekvens av lovlige operasjoner, og ut fra det bildet som dermed skapes (se eksempel i figur under) gjøres en mengde definisjoner, som f.eks. av «system head», «outermost system», «component», «main component», «operative», «operating», «operating chain», osv osv.

I andre delen av kapittelet beskrives så operasjonene «detach», «call» og «resume», og hva de skal gjøre i alle de forskjellige sitasjonene de kan kalles. Det som derved mangler er en forklaring som kan overbevise leseren om at invariantene beskrevet i første delen ikke blir forstyrret av kall på noen av disse operasjonene, eller av noen annen lovlig aktivitet i programmene. Men dette er nok dessverre en nokså komplisert sak å komme å finne fram til (og et slikt bevis ville neppe passe i en definisjon av Simula som den vi her viser til).

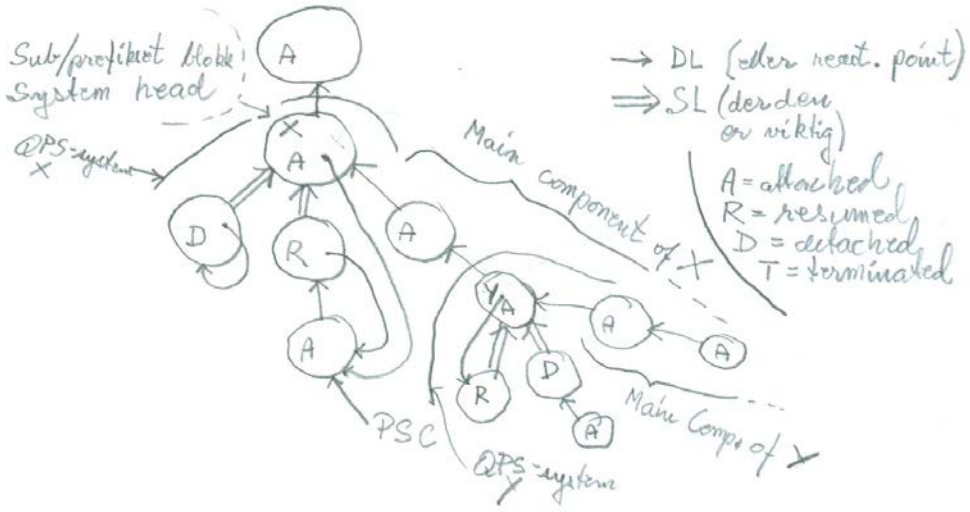
Men om man tenker seg at man har en slik overbevisende forklaring, eller vi bare satser på at en slik finnes, er det greit nok å definere hva det vil si at innføring av en ny operasjon (f.eks. at man fritt får lov til å dotte seg inn i klasser med lokale klasser) vil føre til «inkonsistens». Det skal rett og slett bety at de invariantene som beskrives i første del av kapittelet ikke lenger blir vedlikehold av enhver sekvens av lovlige operasjoner (nå inkludert den nye). Da ville ikke de begrepene som blir definert der forbli veldefinerte slik at de kan brukes som grunnlag for videre aksjoner. Det å påvise slik inkonsistens vil altså kunne gjøres ved å komme opp med en sekvens (altså et program) som bringer oss i en situasjon som ikke stemmer med invariantene i første delen.

Jeg har dessverre ikke fått sett på dette med noen dybde, og inntil videre får derfor de interesserte se om de kan finne en sekvens av operasjoner (inkluderert slik «ulovlig» dotting) som fører til at invarianten brytes.

Noen typiske invarianter som skal gjelde er:

- Sub/prefiksede blokker er alltid i «attached state»
- Når en blokk slutter å eksistere slutter også alle som har SL eller DL til den å eksistere
- Ingen blokk-instanser er «attached» (med DL) til et terminert objekt.
- Et objekt eksisterer ikke lenger enn den blokken der dens klasse er definert

Som støtte for vurderinger rundt dette kan man se på følgende figur:



- - - o o o - - -