9
Fejlhåndtering.

Årsagen til fejl.

Erkendelse af fejl.

Håndtering af fejl.

Fejlerkendelse og -håndtering i objekt-orienterede sprog.

Fejlerkendelse og -håndtering i Eiffel.

Udbredelse af fejl i Eiffel.

Nuanceret fejlhåndtering i Eiffel.

PS1 -- Fejlhåndtering

Noter

Denne forelæsning handler om, hvordan man kan finde ud af, at fejl er opstået i et objekt-orienteret program, og hvordan man kan håndtere dem.

Konventionel fejl-erkendelse og håndtering.

```
P (fp: T) is
local
  ...
do
  if fejl-situation
  then ... ;
  ...
end
```

```
...
if pre-fejl-situation(ap)
then ... ;

P(ap);

if post-fejl-situation
then ... ;
```

- **Forurening af programmet.**
  De interessante dele af programmet drukner i fejl-check og -håndtering.
- **Tidskrævende at evaluere fejl-betingelserne.**
  Et program, der antages fejlfrit, betaler for evaluering af fejl-betingelserne.
- **Ansvarsfordeling?**
  Er det proceduren eller procedurekalderen, der har ansvar for at teste for fejl?

Noter

I et konventionelt program, hvor der ikke er indbygget en eller anden form for specifikation af hvad elementerne i programmet skal opfylde, er der ingen anden mulighed end at programmere eksplicitte check af, om der er fejl.

Hvis der er fejl, skal der reageres passende på disse. Beskrivelsen af disse fejl-reaktioner kan være generende, hvis man skal læse og forstå et program; I ekstremet kan det være vanskeligt at finde "de egentlige aspekter" af programmet mellem "undtagelseshåndteringen" (de fejl-behandlende aspekter). Det er derfor nyttigt at separere fejl-håndtering og mere normale programaspekter i programteksten.

Fejl-check koster udførelsestid. Det er specielt dyrt, hvis der testes mere end ét sted for en fejlsituation; for en sikkerheds skyld... Det er således nyttigt, hvis man "med ét slag" kan eliminere alle fejl-check, når man på et tidspunkt har tilstrækkelig tiltro til, at programmet fungerer korrekt.

Årsag, erkendelse og håndtering af fejl.

Årsag til fejl:

- Der er en fejl i programmet.
- Der er en abnorm tilstand i programudførelsens omgivelse.

Erkendelse af fejl:

- Via eksplicit programmerede tests i programmet for fejl-situationer.
- Via brud på assertions.
- Via brud på fundamentale regler (generelle assertions, så som numerisk overløb) eller overskridelse af ressourcegrænser (såsom mængden af tilgængeligt lager).

Håndtering af fejl:

- Programudførelsen afbrydes (det "går ned").
- Fejlen håndteres med henblik på genoptagelse af udførelsen.
  - sammenflettet med de øvrige dele af programmet.
  - i særligt afgrænsede dele af programmet.

PS1 -- Fejlhåndtering

Noter

Hvis årsagen til fejlen er, at programmet indholder en fejl, er det næsten altid det bedste at rette fejlen fremfor at lade programmet håndtere fejlsituationen på en eller anden "avanceret" måde.

Hvis specifikation og program er integreret, er det naturligt og elegant, hvis brud på specifikationen fører til en erkendelse af fejl. Når der er erkendt en fejl, havner vi i en *undtagelsessituation* som kan vare kortere eller længere, og som måske ultimativt vil afbryde programudførelsen. Vi siger ofte, at der opstår et *exception.*

Hvis der ikke i programmet (og i det understøttende programmeringssprog) er specifikations-elementer, som udtrykker, hvad programmet bør gøre, er der ikke anden udvej end at gøre fejl-erkendelse til en eksplicit del af programmet. Dette udelukker naturligvis ikke, at man kan være systematisk på forskellig vis omkring fejl-erkendelse, f.eks. ved at sikre at visse typer af fejl-erkendelse kan slåes fra på en enkel måde.

Man kan opfatte det som om, at maskinen og det underliggende operativ-system er specificeret ved en række implicitte assertions, som testes passende steder i basale biblioteks-operationer. Hvis der f.eks. divideres med 0, er det en fejl, som vi ønsker erkendt. (Vi ønsker *ikke* at maskinen går ned, og at den skal rebootes...). Tilsvarende, hvis filsystemet er fuldt, eller hvis programmøren afbryder programudførelsen.

Der er mange eksempler på programmer, som nødig skulle "gå ned" på grund af en fejl i programmet. Operativ-system programmet i en maskine, programmer der kontrollerer sikkerheden i farlige industrier, og programmer i livsvigtigt hospitalsudstyr.

I mange moderne programmeringssprog er der en tendens til at separere fejlhåndtering fra de andre aspekter af programmet. Dette løser delvis problemet, som består i, at man ikke kan overskue sit program på grund af mange fejl-check og på grund af megen fejl-håndterings kode.

Exceptions i Eiffel.

Et *exception* er en begivenhed under udførelsen af en operation, som umiddelbart forhindrer operationen i at opfylde sin postbetingelse.

Mulige former for exceptions under udførelsen af en operation i Eiffel:

- Der udføres en operation på et target, der er void.
- Kald af en operation, som ikke får opfyldt sin prebetingelse.
- Reglerne omkring løkkeinvariant eller -variant brydes.
- Operationens postbetingelse opfyldes ikke.
- Der kaldes en operation (på et objekt), som fejler.
- Et assertion i en **check** kommando beregnes til falsk.
- Der rejses et eksplicit exception (ved brug af *raise*).
- Der opstår en abnorm situation på maskinen, som programmet ikke eksplict har taget højde for.

Noter

I Eiffel er det ikke normalt, at programmøren bruger *raise* til eksplicit at bekendtgøre en fejl. Normalt sker dette ved, at et assertion (som er en del af specifikationen) bliver brudt.

Raise er en routine i klassen EXCEPTIONS, som omtales senere i denne forelæsning.

Fejlhåndtering i Eiffel.

```
class C
feature

  op1(...) is
  require pre-op
  do
     ...
  ensure post-op
  rescue
    fejlhåndtering for operation op1
  end

  op2(...) is ...

  op3(...) is ...

  default_rescue is
  do default-fejlhåndtering end

end
```

PS1 -- Fejlhåndtering

Noter

Denne slide viser på skematisk form, i hvilke dele af et program fejl håndteres i en klasse i Eiffel.

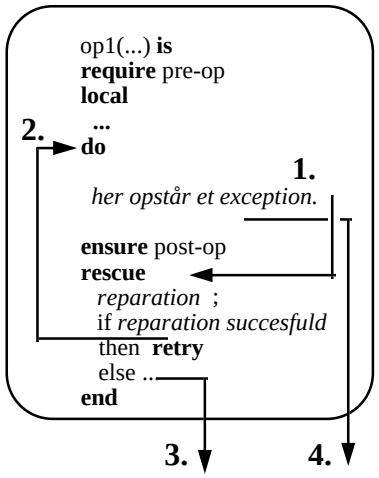
Det mest normale er nok, at en routine har sin egen "redningskrans" i form af en 'rescue clause'.

Hvis ikke en routine har en 'rescue clause', udgør routinen med navnet *default_rescue* routinens 'rescue clause'. Med andre ord, default_rescue spiller rollen af redningskrans for routiner, hvis de ikke har en selv. Det siger næsten sig selv, at en fælles redningskrans i en klasse ikke kan håndtere fejl så nuanceret som i routiner.

Hvis hverken routine eller klasse har en 'recue clause', er den benyttede 'rescue clause' tom for den pågældende routine.

Et ord om nedarvning i forbindelse med rescue: redningsklausuler i routiner nedarves sammen med routiner til subklasser; Også operationen *default_rescue* arves naturligvis af subklasser, og den kan redefineres efter sædvanlige regler.

Fejlhåndtering i en Eiffel-routine.



1. Der opstår et exception.
Rescue-konstruktionen aktiveres.

2. Efter reparation genudføres operationen, hvori exception opstod. Det anbefales af klasse-invarianten **and** pre-op opfyldes.

3. Operationen fejler.
Klasse-invarianten skal være opfyldt.

eller

4. Operationen lykkes.
Klasse-invarianten **and** post-betingelsen skal være opfyldt.

PS1 -- Fejlhåndtering

Noter

Det er væsentligt at slå følgende fast:

- Et exception (en undtagelse) er en *hændelse* der kan opstå under programmets udførelse.
- En routine kan enten *lykkes* eller *fejle*.
- Hvis en routine ikke lykkes forlader den rescue clause uden at udføre retry.

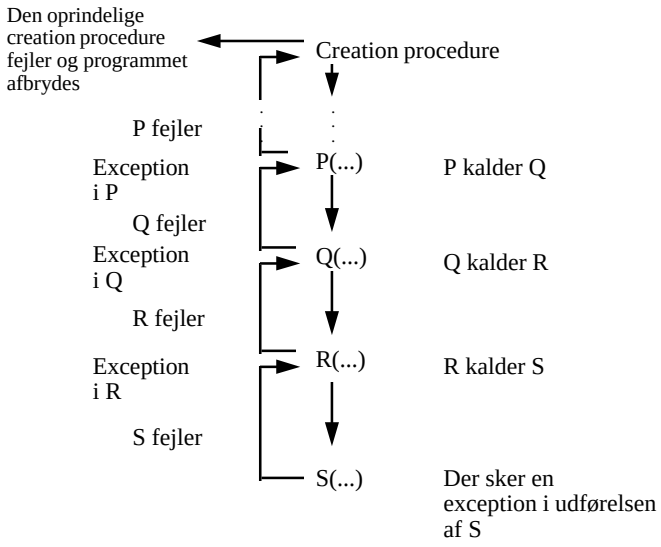
Der er ikke nogen mellemting mellem, at operationen lykkes, og at den fejler. Hvis post-betingelsen ikke kan opfyldes, skal operationen fejle.

Hvorfor skal den effektive pre-betingelse være opfyldt efter et retry (tilfælde 2 ovenfor)? Et retry starter routinen i den nuværende tilstand, uden at omgøre parameteroverførsel eller definition af lokale variable (Det er derfor ingen tilfældighed, at pilen 2. peger på **do**, og ikke **local**). Ved enhver begyndelse (eller som her,, nybegyndelse) af en routine skal den effektive prebetingelse være opfyldt. Hvis ikke man opfylder dette krav vil det være meget svært at programmere en ordentlig routine krop, idet vi ikke i starten af routinen er klar over vore forudsætninger (om vi kommer udefra, eller fra retry).

Det er værdifuldt, at invarianten af klassen er opfyldt, selv om operationen fejler (tilfælde 3). Det at operationen fejler er ikke ensbetydende med, at programmet bliver afbrudt. Tidligere kald af routiner kan vælge at reparere fejlen, således at programmet kører videre. Det vil da være fatalt, hvis der findes objekter, som er i en tilstand, der bryder med deres invarianter.

Det skal bemærkes, at kommandoen retry tekstuelt skal forekomme i rescue-delen af operationen. Det er således ikke tilladt at kalde en operation, hvori retry udføres. Specielt er det ikke muligt at udføre retry fra default_rescue.

Fejl-udbredelse i Eiffel (I).

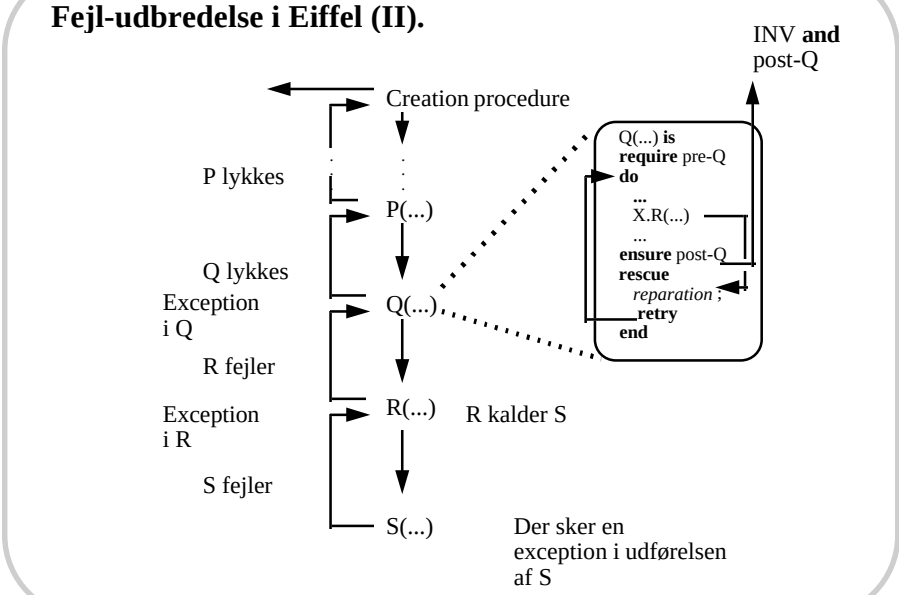


Noter

Det skitserede forløb er ikke nødvendigvis det eneste mulige.

Hvis én af de involverede routiner, f.eks. Q, håndterer fejlen, således at Q lykkes, ændres forløbet af fejlhåndteringen. Dette er vist på næste slide.

Fejl-udbredelse i Eiffel (II).



Noter

Det er vist, at Q, som kalder R på et objekt, der refereres af X, håndterer fejlen, som indirekte er opstået ved, at R fejlede.

Redningsklausulen i Q reparerer fejlen og genstarter Q. Vi antager, at dette resulterer i, at Q opfylder sin specifikation; R kaldes igen på objektet refereret af X, og tilsvarende kan S blive kaldt igen af R. Hvis Q's reparation har været effektiv, vil S ikke fejle igen.

Nuanceret fejlhåndtering.

Vi ønsker at kunne kende forskel på forskellige former for exceptions i en 'recue clause'.

```
op(...) is
require pre-op
do
  ...
ensure post-op
rescue
  if exception = ...
  then ...
  elsif exception = ...
  then ...
  else default_rescue
end
```

PS1 -- Fejlhåndtering

Noter

Klassen EXCEPTIONS indeholder en række mekanismer, der støtter en mere nuanceret fejlhåndtering. Exception, som er et heltal, betegner arten af den sidst forekomne exception. Klassen indeholder desuden en række andre informationer og mekanismer, som kan være nyttige for en mere nuanceret undtagelseshåndtering.

For at bruge faciliteterne i EXCEPTIONS i klassen C skal man sørge for, at C har adgang til EXCEPTIONS. Dette kan ske ved at lade C arve fra EXCEPTIONS. Vi vender tilbage til nedarvning i en af de følgende forelæsninger.

Klassen EXCEPTIONS er beskrevet i kapitel 29 i “Eiffel the Language”.

Om anvendelse af ‘exception handling’.

- Undlad for enhver pris at bruge exceptions som et implicit udført hop. Exception handling mekanismen er ikke en avanceret kontrolstruktur.
- Lad være med at udføre komplicerede handlinger i rescue-delen af et program.
  - “Oprydning før lukketid” er i de fleste tilfælde tilstrækkeligt.
- I de fleste programmer afspejler et exception en fejl i programmet. Det mest fornuftige er at afbryde udførelsen af programmet, og dernæst rette fejlen i programmet.
- I programmer, som kræver ekstrem robusthed, er det muligt via håndtering af exceptions i recue-delen af et Eiffel program at holde programmet kørende på trods af opståede exceptions.

Noter

“Oprydning før lukketid” kan enten være i interne data i programmet (sørge for, at objekter overholder klasseinvarianter), eller i eksterne data. Hvis programmet arbejde på eksterne data er det katastrofalt at disse efterlades i en inkonsistent tilstand, f.eks. som følge af “en halv opdatering”. Om muligt, skal man i rescue-delen af en opdateringsoperation sikre sig, at eksterne data er sunde, inden programmet får lov til at terminere.