Generiske mekanismer i statisk typede programmeringsspråk

Dette stoffet er Pensum, og det er bare beskrevet her

Mye her er nok kjent stoff for mange

INF5110 – 7. mai 2013

Stein Krogdahl

Hvordan kunne skrive type-uavhengige programmer og samtidig få fordelen av statisk typing?

Først: Oppsummering om statisk typing:

- Statisk typing i programmeringsspråk er godt for flere ting:
  - Kompilatoren kan finne inkonsistenser/feil allerede under kompilering
  - Den kan lage mer effektiv maskinkode
  - Virker som dokumentasjon (som kompilatoren tvinger deg til å holde konsistent)

- Men statisk typing kan også gjøre språket mindre fleksibelt
  - I utgangspunktet var typene disjunkte verdi-sett, som bare kunne blandes i helt spesielle situasjoner (f.eks. *integer + real*), men ellers ikke.
  - Dette gir ofte problemer når
    - Vi definerer egne typer (f.eks. recorder eller klasser), og vi samtidig:
    - Vi vil sette sammen program-biter som er programmert uavhengig av hverandre.
    - Vi vil skrive ”samme” programmet for forskjellige typer (tenk: sortering av *int* og *float*) må skrives som separate programmer.

- Dette er ett av de sentrale spenningsfelter ved språkdesign:
  - Lag et type-system som både er
    - Fleksibelt (og det krever ofte at en verdi (med gitt type) også kan håndteres gjennom variable av andre typer).
    - Mest mulig kan sjekkes av kompilatoren (som ofte krever at den vet hvilken type verdi en variabel angir)

Problemstillinger rundt generiske

- TYPISKE PROBLEMER: Man ønsker å skrive programbiter som kan virke for flere typer
  - F.eks. skrive et sorterings-program uten å kjenne typen av elementene som skal sorteres, bare at typen har to operasjoner ”eq(a,b)” og ”lt(a,b)”.
  - Lage en ”Collection-klasse” som kan virke for alle typer elementer
    - Alle Collection-objekter skal kunne inneholde alle typer elementer. *Det går greit i tradisjonelle OO-språk (se neste side)*
    - Noen Collection-objekter skal bare ha *personer*, andre bare *kjøretøy*. Og kompilatoren skal kunne sjekke at dette overholdes. *Dette er verre!*

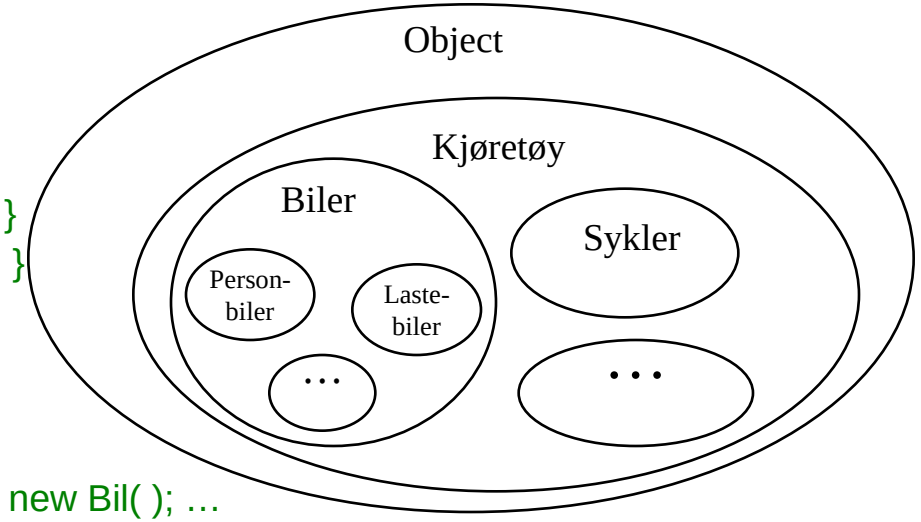
- MULIGE LØSNINGER?:
  - La sorterings-algortimen ha typen som vanlig ”run-time”-parameter:

    void sorter(type Elem, Elem[ ] verdier) { ... Elem e, f; ... }
  - La Collection-klassen ha en tilsvarende parameter på konstruktøren
    - Som angir hva slags objekter dette Collection-objektet skal kunne inneholde.

  - MEN NEI: Da mister vi alle fordelene med typer, angitt på forrige foil
    - Kompilatoren kjenner ikke typene, og kan dermed sjekke nokså lite.,

Objektorientering løste en del (en viktig side-historie)

- Objekt-orientering, med klasser og subklasser, gav oss en 3/4 god løsning på dette.
  - Ethvert objekt av klasse A er også medlem av alle superklasser av A, inklusive Object
  - Eller: Enhver ref-variabel typet med B kan også peke til objekter av subklasser av B.
- Derved har ikke typene lenger disjunkte verdisett, men kan være subsett (altså "subklasser") av hverandre.



- Eksempel: Typisk Collection-klasse:

```
class Collection {
    void add(Object o) { … implementasjon … }
    Object getOldest( ) { … implementasjon … }
}
```

- Ønsker en "collection" av biler:

```
Collection bilColl = new Collection( );
Bil b1, b2, b3; … b1 = new Bil( );  … b2 = new Bil( ); …
bilColl.add(b1);  … bilColl.add(b2);  // Får ingen feilmelding om vi i stedet legger inn "sykler"
b3 = (Bil) bilColl.getOldest( );          // Må gjøre "cast"
```

  - Ulempene ved denne løsningen er angitt i kommentarene i programmet over
  - Men dette er dog en klar forbedring i forhold til ikke å ha klasser/subklasser
  - Bibliotekene for Java og C# (*før* innføring av generiske) var bygget på dette prinsippet
  - Prinsippet kalles ofte "det generiske idiom"

Derfor:
Generiske mekanismer er ofte en bedre løsning.

- Ikke bruke type-parametere ved runtime eller det generiske idiom:
  - Vi lager en språkmekanisme slik at kodebiter kan parametriseres med typer/klasser
  - Disse parameterene leses og behandles av *kompilatoren* (ikke "run-time"-parametere)
  - Slike parametere kalles *generiske parametere*, og slike mekanismer: *generiske mekanismer*.
  - Typisk sytaks:

    (generic) class A<T, U>{ ...bruk av T og U som vanlige typer...}
  - Om C og D er vanlige klasser, kan nå A<C, D> brukes som en vanlig type
  - Typelikhet: Hver gang man skriver A<C, D> så anses det som samme typen
    - Men om man sier A<E, F> (eller A<C, F>) så anses det som en helt forskjellige typer
- Et viktig skille mellom forskjellige generiske mekanismer:
  - Er det bare *klasser* som er lovlige som generiske parametere?
  - Eller kan de også være basis-typer som "int", "bool", "float" osv.?

To hovedstategier for implementasjon av generiske mekanismer

- Heterogen implementasjon
  - Hver gang kompilatoren ser at en generisk klasse er brukt med en ny parametrisering lager den en ny tekstlig versjon, og kompilerer den på vanlig måte.
  - Implementasjonen er altså veldig ”makro-aktig”
  - Generiske i Ada og (templates) i C++ implementeres normalt på denne måten
  - Fordeler: Lett å implementere, blir effektiv kode
  - Lite problematisk å tillate basis-typer (”int” og ”float” osv.) som generiske parametere
  - Ulemper: Kan få mange kopier av nesten lik kode, som til sammen tar mye plass

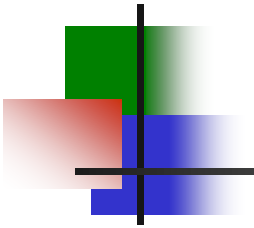
- Homogen implementasjon
  - Det lages bare én felles oversettelse til maskinkode av programbiten
  - Denne må være så generell at den kan brukes for alle parametriseringer
  - Fordeler: Man slipper duplisering av kode, og sparer dermed plass
  - Ulemper: Utførelsen tar som regel noe lengere tid.
  - Dette gjelder spesielt om man tillater ”int” og ”float” som generiske parametere

Implementasjon av generiske klasser i Java og C#

- Både Java og C# har altså egne ”mellom-språk” som de vanlige kompilatorene oversetter til
  - Java: Byte-kode
  - C#: CIL-kode.

- Implementasjon av C# 2.0 (med generiske klasser)
  - Det ble lagt til spesielle CIL-instruksjoiner for å håndtere generiske klasser
  - Hver generisk klasse blir alltid bare oversatt til én CIL-fil (i den forstand homogen)
  - Men under JIT-kompilering/loading lager man nye kodeversjoner når det er nødvendig (derved noe heterogen).
  - Alle instansieringer som bare har klasser som parametere får felles kode
  - De som har basis-typer som ”int” og ”float” som parametere får egen kode.
  - Grunnen er at effektiv kode bl.a. krever kjennskap til relativ-adresser i objekter

- Implementasjon i Java 1.5 (med generiske klasser)
  - Valgte å *ikke* lage nye byte-kode instruksjoner. Er for mange JVM-er ute allerede...
  - Derved fikk de en del begrensninger og rare mekanismer
  - Tillater *ikke* ”int” og ”float” som generiske parametere (men har automatisk ”boxing”)



Generiske klasser i Java (versjon 1.5)

Anta den generiske klassedefinisjon: class G<T extends C> { … }

- Java 1.5 har ingen runtime-representasjon av de parametriseringer som er brukt av en generisk klasse.
  - En generisk klasse oversettes til vanlig byte-kode, men først settes type-parameterene i tilfellet over til C (altså til den skranken som er gitt, ellers Object)
- Så, på *bruksstedet*, der kompilatoren kjenner de faktiske parametere, setter den inn ”casting” etc. ut fra kjennskap til disse. Derfor følgende egenskaper:
  - Man kan ikke ”caste” til G<A> eller si ”instanceof G<A>”
    Man kan ikke (inni klassen) ”caste” til T eller si ”instanceof T”.
  - Alle klasser G<A>, der A er en aktuell parameter, har felles sett av statiske variable og metoder. Derfor:
    - Typene på disse kan ikke avhenge av T
- Bruker altså samme runtime-koden for alle parametriseringer av en generisk klasse (homogen implementasjon)

Generisk C# (versjon 2.0)

- C#-kompilatoren lager en egen ”runtime-descriptor” for hver av de parametriseringer som er brukt av en generisk klasse.
- G<A> og G<B> blir derfor vanlige (separate) klasser, på en mer fullstendig måte enn i Java 1.5.
  - Hver parametrisering (= egen klasse) har sitt eget sett av statiske variable/metoder
  - Man kan bruke ”casting” og ”instanceof” helt fritt, både med parameteren T (inne i G) og med klassen G<A>.
  - Man kan (inni G) gjøre ”new T( )”, dersom man har spesifisert T slik:

    class G<T> where T: C, new( ) { … }

    Kan dog ikke bruke konstruktører med parametere.
- Man kan gi alias-navn til generiske klasser med aktuelle parametere:

  using GforA = G<A>;
- I tillegg til generiske klasser og interfacer, kan man ha generiske ”struct”er
- Bruker samme runtime-koden for G<A> for alle A, så langt det er mulig (oftest, så lenge den/de aktuelle parametrene bare er klasser/interfacer).