# Generiske mekanismer i statisk typede programmeringsspråk

Stein Krogdahl
INF-5110, 24 april 2007

---

## Dagens temaer:

- Hva er generiske mekanismer, og hvorfor har vi dem?
- Hovedtyper av generiske mekanismer
- Hovedstrategier for implementasjon av generiske mekansimer.
- Litt om de nye generiske mekanismene i Java 1.5 og C# 2.0
- Noe vi har arbeidet litt med i OMS-gruppa/SWAT-prosjektet:
  Generiske *pakker* i objekt-orienterte språk

# Hvorfor generiske mekanismer?

- 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:
    - vil sette sammen program-biter som er programmert uavhengig av hverandre

- 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)

# Spesifikke problemer

- PROBLEM: Man ønsker å skrive programbiter som kan virke for flere typer
  - F.eks. skrive en sorterings-metode 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*
    - Noen Collection-objekter skal bare ha *personer*, andre bare *kjøretøy*. Og kompilatoren skal kunne sjekke at dette overholdes. *Dette er verre!*

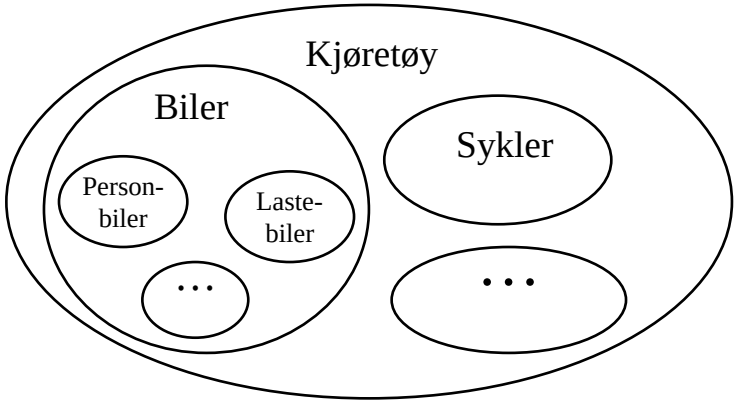
- MULIG LØSNING?:
  - La sorterings-algortimen ha typen som vanlig ”run-time”-parameter:

    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”, og vi kan lett får run-time-feil
```

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

# Generiske mekansimer

Problemet med å lage kodebiter som kan virke for flere typer, bedre løsning:

- 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 vanlig type
- Typelikhet: Hver gang man bruker 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 annen type

Viktig skille mellom generiske mekanismer:

- Kan de generiske parametrene bare være klasser?
- Eller kan de også være basis-typer som ”int”, ”bool”, ”float” osv.?

# Tre hovedtyper av generiske mekanismer

- Ren tekst-makro, som f.eks. i C:
  - Man står selv ansvarlig for både syntaks, og semantikk.
  - Får ingen hjelp av kompilatoren før makroen ”ekspanderes” til tekst og kompileres på vanlig måte

- Templates i C++:
  - Det man skriver som en generisk klasse blir bare sjekket syntaktisk,
  - men semantisk sjekk gjøres ikke før man vet i hvilken sammenheng og med hvilke faktiske parametere den skal brukes

- Generiske klasser i Java og i C#, og generiske mekanismer i Ada:
  - Den generiske klassen e.l. kan sjekkes både syntaktisk og semantisk uten å vite hvilken sammenheng den skal brukes i
  - Generiske parametere kan ”avgrenses”, for eksempel ved:
    - ”Skal være subklasse av en gitt klasse og/eller implementere en gitt interface”
    - Skrives typisk:

```
(generic) class A<T extends C, U implemets I>{
    …Vet mye om T og U, ved at de har alt hhv. C og I har …
}
```

# 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”
  - Fordeler: Lett å implementere, blir effektiv kode
  - Ulemper: Kan få mange kopier av nesten lik kode, som tar mye plass
  - Mindre problematisk å tillate basis-typer (”int” og ”float” osv.) som generiske parametere
  - Generiske i Ada og (templates) i C++ implementeres normalt på denne måten

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

- Java (1.5, med generiske) har normalt en ren homogen implementasjon
  - og får en del begrensninger av det, og av at de ikke ”turte” å utvide bytekoden
- C# (2.0, med generiske) har normalt en blandet implementasjon
  - har derfor færre begrensninger

# På språknivå: Generiske klasser i C# og Java

Anta at følgende er definert:

interface I{…}  class C{…}

Definisjon av en generisk klasse:

C#: class G<T> where T:C, I {
… her kan T brukes som en vanlig klasse/type
som har alt som er definert i I eller C …
}

Java: class G<T extends C implements I> {
… som over, men litt mer begrenset (mer senere) …
}

En mulig aktuell parameter:

class AC extends C implements I{ … } (og tilsvarende for C#)

Denne kan så brukes slik:

G<AC> g = new G<AC>( );

Man må altså angi parameteren hver gang, og det er viktig at
*samme parameterisering* gir *samme klasse*.

C#: Generiske parametere kan være både klasser, interfacer, og grunntyper
Java: Generiske parametere kan bare være klasser og interfacer

# 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 parametriseringen som er brukt av en generisk klasse.
- En generisk klasse oversettes til vanlig byte-kode, men etter at type-parameterene er erstattet av sin ”avgrensning” (ofte Object).
- Så, på *bruksstedet*, der man kjenner de faktiske parametere, gjøres ”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)

# Mer om generisk Java

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

- Man kan ikke (inni G) gjøre ”new T( )”. Må bruke såkalt ”factory-teknikk”
- Klassene G<A> og G<B> har ingen sub/super-relasjon til hverandre, heller ikke om A og B har det. (Altså ikke ”kovarians”, som for arrayer).
- Joker-notasjon (wildcard): Man kan bruke G<?> eller G<? extends A> eller G<? super A> som type på variable m.m. (A må da være subklasse av C). G<?> kan altså sies å være en supertype av G<B>, for alle lovlige B.

  Variabel ”G<?> g” kan da peke til objekter av alle klasser G<A>
  Dersom ”G<A> gA”, så går ”g = gA”, men ikke omvendt

- Man kan også bruke bare G som type. Dette kalles den ”rå” typen, og den likner på G<?>, men mer er tillatt.
- **Blandet kode**: Om man har et bibliotek som er laget med generiske klasser
  - men man i sitt program bare bruker klassene derfra som rå typer (uten parametere)
  - Da virker dette som om man bruker et tilsvarende bibliotek uten generiske klasser.
  - Man får advarsler fra kompilatoren der det kan bli feil under utførelsen
  - Dermed er det nokså greit å blande gammel og ny kode

# 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 GA = 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).
- Ikke noe som tilsvarer joker-notasjon.

# Generelt for både Java og C#

- Det tillates ikke å bruke parameteren T som superklasse i en (indre) klassedefinisjon.

- Man kan bruke såkalt ”F-bounded polymorphism”

  class G<T extends G<T>>

  En klasse som kan brukes som aktuell parameter for T er f.eks.:

  class A extends G<A>

- Man kan ha generiske parametere på metoder (gjelder også statiske metoder):

  Java: <U extends B>U getValue (int i, U u) {…kan anta at U har alt B har…}

  C#: U getValue <U> (int i, U u) where U:B{…kan anta at U har alt B har…}

  Ut fra den konteksten en metode kalles i og ut fra de aktuelle parametere finner Java/C# ”nesten alltid” den riktige” typen av U. Den behøver da ikke angis.

- Det blir et utall av nye regler for navnebinding etc, f.eks.:
  - Utgått nå: Må passe på at det ikke blir dobbeltdeklarasjon av de to m-metodene i:

    class G<T>{ …; void m(T a){…}; … void m(C a); …}

    Problemer med overloading om G gies aktuell-parameteren C (men er nå løst!)

  - Parameteren T er det greiest at bare er synlig i klassen G, ikke i subklasser.
  - Skriver ofte f.eks.: class H<U, V> extends G<V>{…}

# Interfacer til Collection-klassene i Java

Det kan virke kurant å gjøre om Collection-klassene i Java/C# til generiske varianter, men det er mange detaljer som skal stemme. Litt fra Javas generiske bibliotek (forenklet):

```
interface Collection<E>{
        void add(E o);
        bool containsAllOf (Collection<?> c); // Skal kunne gi med (som c) alt av type
        ...                                   // Collection<A>, for alle A
}

interface Comparator<T>{
        int compare(T o1, T o2);
        ...
}

interface SortedSet<E>{
        Comparator<? super E> comparator( );  // Må levere et Comparator-
                              // objekt som kan sammenlikne objekter av klassen E
        E first( );
        E last( );
        ...
}
```

# Implementasjon av generiske klasser i Java og C#

- Både Java og C# har definert egne ”mellom-språk” som de vanlige kompilatorene oversetter til
  - Java: Byte-kode, lagret sammen med ”metadata” i ”klassefiler”
    - Opprinnelig beregnet på interpretering, men blir nå oftest oversatt før kjøring: ”Just In Time”-(JIT)-kompilering
  - C#: CIL-kode. Likner mye på byte-kode (og kalles ofte også det)
    - *Bare* beregnet på videre kompilering: JIT-kompilering, eller mer tradisjonell kompilering gjort lenge før kjøringen

- Implementasjon av C# 2.0 (med generiske klasser)
  - Det er lagt til spesielle CIL-instruksjoner 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 alltid 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 ute allerede…
  - Derved fikk de en del begrensninger og rare mekanismer
  - Tillater ikke ”int” og ”float” som generiske parametere (men har automatisk ”boxing”)

# OMS-arbeid (i SWAT-prosjektet): Generiske *pakker* i stedet for generiske *klasser*.

- Grunner for å forsøke dette:
  - Implemetasjonen av begreper består ofte av flere klasser
  - Eks: Graf, med klassene *node* og *kant*.
  - Det er selvfølgelig derfor begrepet *pakke* er laget i Java
  - Det kan da være rimelig å parametrisere hele slike pakker som en enhet.
  - Hver instansiering blir et helt nytt sett med klasser, og under instansieringen kan man gjøre alle slags substitueringer/transformasjoner/utvidelser man måtte ha lyst til.
  - Typesystemet blir ikke mye verre enn i Java, og svært mye kan gjøres ferdig i kompilatoren

- Grunner til ikke å satse på dette:
  - Det blir noe mer statisk over pakker. Dette kan gjøre det hele mindre fleksibelt å bruke.
  - Man kan i stedet bruke klasser inne i klasser, og parameterisere den ytre klassen. (Kan f.eks. gjøre det omtrent som i språket Beta, der det finnes ”virtuelle klasser”).

# Mulig syntaks for generiske pakker

```
class C{ … }     / /  Definert f.eks. i en vanlig pakke P

generic package GP<T extends P.C> {
   class A{... ; T t = new T( ); ... } // Kan bruke T fritt, og anta at den har alt P.C har
   class B{ ...  tilsvarende …}
}

------------------------------  Så bruk av GP: ------------------------------------

class D extends P.C{ int i;} // D derved en lovlig parameter til GP

inst GP<D>;  // Dette blir en vanlig pakke, der D er satt inn for T
                  // Denne pakken blir samtidig importert (A og B blir direkte synlige)
A a = new A( ); a.t.i = 5;  // Vet her at a.t er av type D, så det er OK å be om ”i”
```

Merk at vi slipper parameter på A og B hele tiden (som vi må ha ved generiske klasser).

# Eksempel: Her er det greit med generiske pakker

```
generic package GRAPH <NodeData, EdgeData> {   / / Ingen spesifikasjon av parameterene
    class Node {
        Edge firstEdge;            / / Peker til første kant ut av noden
        NodeData nd;
    }

    class Edge {
        Node from, to;
        Edge nextEdge;             / / Peker til neste kant ut av samme noden
        EdgeData ed;
    }
}
-------------------------------- Ville se slik ut ved vanlige generiske klasser: ----------------------------------------

package GRAPH {                                 / /  Skriver en vanlig pakke på ”lukket” form
    generic class Node<NodeData, EdgeData> {
        Edge<NodeData, EdgeData> firstEdge;             / / Peker til første kant ut av noden
        NodeData nd;
    }

    generic class Edge<NodeData, EdgeData> {
        Node<NodeData, EdgeData> from, to;
        Edge<NodeData, EdgeData> nextEdge;              / / Peker til neste kant ut av samme noden
        EdgeData ed;
    }
}
```

- Merk at vi må ha begge data-typene som parametere til begge klassene
- Vi må her passe veldig på at vi bruker klassene Node og Edge med riktige parametere hele tiden.

# Mulig utvidelse 1: ”Ekspandering” av klasser ved instansiering

**NB: en litt annen definisjon og bruk av en liknende graf-pakke:**

```
generic package GRAPH expandable Node, Edge {
    class Node {
        Edge firstEdge;
        Edge insertEdgeTo(Node to){ ... }   / / Leverer den kanten som er satt in
    }

    class Edge {
        Node from, to;
        Edge nextEdge;                      / / Next in the list of edges leaving the same node
        void deleteMe( ){ ... }
    }
}
---------------------------------------- Bruk: ----------------------------------------

inst GRAPH with  Node =>City, Edge =>Road;  // Instansierer GRAPH-pakka, og importerer den
expansion City {                  / / Ekspansjon av Node
  String name;
  / / Inni en metode:
    int n = firstEdge.length;      / / OK, siden typen av "firstEdge" nå er Road
    City  c = new City( ... );
    Road  r = insertEdgeTo(c);  / / Her er det nå eksakt type-overenstemmelse hele veien ...
}
expansion Road {                  / / Ekspansjon av Edge
   int length;
}
```

# Mulig utvidelse 2:
# Slå sammen klasser fra flere generiske pakker til én

Vi vil lage klassene City og Road på nok en annen måte. Vi tenker oss en pakke med klasser for datene som skal være med i City og Road:

```
generic package GEOGRAPHYDATA expandable CityData, RoadData {
   class CityData  { ... , bynavn, antall innbyggere etc. …}
   class RoadData { ..., lengde, veikvalitet, etc. … }
}
```

Vi har dessuten den gamle pakka:

```
generic package GRAPH expandable Node, Edge { ... }
```

---------- Bruk: ----------

```
inst GEOGRPHYDATA with CityData =>City, RoadData =>Road;
inst GRAPH with Node =>City, Edge =>Road;

expansion City { ... Noe mer her? ...}  // Får alt fra både CityData og Node

expansion Road { ... Noe mer her? ...}  // Får alt fra både RoadData og Edge
```

---------- Tanker: ----------

- Hva skjer her om for eksempel CityData og Node hadde subklasser i pakkene? Går det bra? Skaper det noe interessant?
- Dette har et snev av noe som i det siste er kalt ”Traits” (metode-samlinger som kan settes sammen på forskjellige måter for å danne klasser).
- Dette er også omtalt (her på huset) som ”multippel statisk arv”, og dette kan *behandles ferdig i kompilatoren.*