UNIVERSITETET I OSLO

Institutt for informatikk

En homogen implementasjon av Package Templates i Java.

Masteroppgave

Finn Christian Brøndal
Anders Stensby Widgren

19. mai 2009

Forord

Denne oppgaven er skrevet i forbindelse med vårt masterstudium ved Institutt for Informatikk, Universitetet i Oslo. Oppgaven har bestått i å implementere den generiske mekanismen ”Package Templates” i Java.

Vi vil begge takke våre veiledere Stein Krogdahl og Fredrik Sørensen for utmerket veiledning og dyp tålmodighet i forbindelse med oppgaven.

Innhold

Figurliste

Kapittel 1

Innledning

I denne oppgaven vil vi presentere en implementasjon vi har gjort av Krogdahls, Sørensens og Møller-Pedersens forslag til ”Package Templates” i (1). Forslaget ble opprinnelig utformet av Stein Krogdahl i (2), men etter dette har det blitt foretatt en rekke endringer, og mekanismen har endret navn fra *GePEC, gjennom PaTEC* til *Package Templates* (1), også kalt bare *PT*.

PT er en generell generisk mekanisme som kan passe inn i mange objektorienterte språk. Vi har valgt å legge det inn i Java fordi dette er et velkjent programmeringsspråk og fordi det også har blitt ekspandert med andre generiske mekanismer i senere tid. En annen grunn er at Java er det programmeringsspråket vi begge har best kjennskap til og har brukt mest i vår utdannelse ved Universitetet i Oslo.

Formålet med den generiske mekanismen PT er å gjøre det enklere å skrive gjenbrukbar og modularisert kode som også er bedre organisert og lettere forståelig. Koden man vil gjenbruke skrives som såkalte ”pakke-templater”, som må ”instansieres” (under kompilering) før de blir vanlige Java-liknende pakker med klasser klare til bruk. De kan instansieres flere ganger i samme program og vil da hver gang gi opphav til et helt nytt sett klasser. Ved hver instansiering kan man tilpasse klassene i templaten på forskjellige måter, for eksempel ved å gi dem nye variable og metoder, eller å forandre navn på de gamle. Dessuten kan templater ha typeparametre som må angis ved instansiering. En av fordelene med PT er at det (i motsetning til templater i C++) er mulig å typesjekke templatene uavhengig av hvor de skal brukes.

Når flere templater instansieres kan klasser fra forskjellige instanser gis et felles tillegg, og dette vil bevirke at vi får én ny klasse som har alle attributtene (variable og metoder) til alle involverte templat-klasser, samt de gitt i tillegget. Vi sier da at disse templat-klassene blir ”sammen-slått” (merged), og får et felles tillegg. Denne mekanismen gir, om man vil lage en såkalt homogen implementasjon (se nedenfor), omtrent samme implementasjons-problemer som multippel arv, slik vi kjenner dette for eksempel fra C++.

For at den kompilatoren vi lager skal bli mest mulig anvendelig ønsker vi, om mulig, at den produserte koden er byte-kode som kan kjøres på en helt vanlig JVM. På samme måte ønsker vi helst en homogen implementasjon, da noe annet kan lage plassproblemer under kjøring (se under).

Vi ville gjerne ha et navn på den kompilatoren vi skal lage og vi følte det stod mellom følgende tre; PT-Java, JPT og PTJ. PT-Java er nok ikke et lovlig navn siden Java er et registrert varemerke. Vi syns JPT ruller litt bedre på tungen enn PTJ, så derfor valgte vi å kalle kompilatoren for JPT. JPT er altså en forkortelse for *Java Package Templates*. men det kan for så vidt også stå for *JastAdd Package Templates* (som skulle vise seg også å være et rimelig navn).

1.1 Homogen og heterogen implementasjon

Implementasjonen av PT kan (i likhet med andre generiske mekanismer) i hovedsak skje på to måter, og disse byr på ulike utfordringer.

En såkalt *homogen* implementasjon benytter per definisjon en felles runtime-kode for alle instansieringer av den generiske koden, og denne må da være generell nok til å fungere for alle aktuelle tilpasninger og typeparametre. Ved en homogen implementasjon holder det altså å kompilere den generiske koden én gang og gjenbruke denne koden uten å rekompilere. Den største ulempen forbundet ved en homogen implementasjon er at den vanligvis blir noe senere ved runtime. Dette er fordi koden må lages generell nok til at den virker for alle instansieringer og at det da gjerne blir en del ekstra tabelloppslag og liknende for å finne riktige aktuelle parametre etc.

Ved en såkalt *heterogen* implementasjon blir det generert egen runtime-kode for hver instans av den generiske kildekoden. Den totale kodens størrelse ved

runtime vil derved øke for hver instansiering, og hvis templater blir instansiert mange ganger i et program kan det oppstå plass-problemer. Den heterogene koden er altså spesial-kompilert for hver instansiering og vil derfor eksekvere like fort som vanlig ikke-generisk kode.

Det er ikke før laget noen homogen implementasjon av PT (eller av dens forgjengere) som kan kjøres direkte på en JVM. Det finnes én implementasjon som går på en utvidet JVM med noen ekstra, spesialtilpassede instruksjoner (3), og vi skal se på denne senere i oppgaven. Vi skal også se på en tidligere heterogen implementasjon (4) for å kunne vurdere om en slik implementasjon ville være bedre, på tross av at vi i utgangspunktet ønsker en homogen implementasjon.

1.2 Oppgavens inndeling

I *kapittel 2* skal vi først gi en presentasjon av *generiske mekanismer* og *typeparametere* i Java. Vi vil så beskrive *Package Templates*, slik Krogdahl et. al. har tenkt seg det i sin rapport (1).

I *kapittel 3* skal vi presentere de mest aktuelle utgangspunktene vi har for å programmere *frontend-delen* og *backend-delen* til vår kompilator. Kapittelet er ment å gi et godt grunnlag for å kunne velge hva vi vil bruke angående utgangspunkt og verktøy i selve arbeidet med kompilatoren.

Kapittel 4 omhandler programmeringsarbeidet og de data-strukturer og fil-formater vi bruker. Kapittelet gir også en mer detaljert innføring i de verktøyene vi valgte å bruke.

I det siste og konkluderende *kapittel 5* skal vi gi en oppsummering av oppgaven, samt diskutere noen problemstillinger som det kunne vært naturlig å arbeide videre med etter at denne oppgaven er levert.

Kapittel 2

Generiske mekanismer

I dette kapittelet skal vi gi et generelt innblikk i generiske mekanismer og hvilke muligheter man får ved å utvide programmeringsspråk med slike. I andre del av kapittelet skal vi se på den mekanismen vi skal implementere slik den er foreslått i artikkelen *Exploring the use of Package Templates for flexible re-use of Collections of related Classes* (1).

Generiske mekanismer innebærer at man kan skrive programbiter der noen av typene ikke er endelig fastsatt, men i stedet opptrer som såkalte *formelle typeparametre* til programbiten. Slike mekanismer gjør det mulig å skrive kortfattet kode med færre program-moduler siden man for eksempel ikke behøver å skrive en sorteringsalgoritme for hver elementtype. Merk at *typeparametre* ikke kan fungere som vanlige (dynamiske) metode-parametre. De aktuelle *typeparametrene* må fastsettes allerede under kompilering/loading for at type-systemet skal fungere. Derfor blir innsettingen av disse parametrene gjort i kompilatoren når programelementet blir satt sammen.

Vi vil benytte oss av et Java-liknende språk i eksemplene for å illustrere bruken av slike mekanismer, men først kan vi se på et eksempel med bruk av den generiske mekanismen i C++.

```
template<typename T>
class Liste {
   /* class contents */
  void leggTil(T o) { … }
}
Liste<Dyr> liste_av_dyr;
Liste<Bil> liste_av_biler;
```

Figur 2.1: Eksempel på templat i C++

Her er altså `T` en formell typeparameter mens `Dyr` og `Bil` vil være aktuelle typeparametre for `T`. Kompilatoren/loaderen vil si fra om man for eksempel prøver å sette inn bil-objekter inn i en "`liste_av_dyr`", og den kan også utnytte at i "`liste_av_dyr`" er det bare Dyr-objekter (ved at man slipper å gjøre type-kast).

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E>
{
    E next();
    boolean hasNext();
}
```

Figur 2.2: Eksempel på bruk av formelle typeparametere i generiske grensesnitter

Typeparametrene angis (både de formelle og de aktuelle) vanligvis i spiss-parenteser, slik som <E> over. Den formelle typeparametren kan bli brukt gjennom hele den generiske deklarasjonen, stort sett over alt der man kan bruke vanlige typer.

```
List<Integer> myIntList = new LinkedList<Integer>();
```

Figur 2.3: Eksempel på innsetting av aktuelle typeparametere

I figuren over ser vi kallet på den generiske typedeklarasjonen `List` er brukt som `List<Integer>` og dette står altså for en versjon av `List` hvor alle tekstlige forekomster av `E` har blitt byttet ut med `Integer`:

```
public interface IntegerList {
    void add(Integer x)
    Iterator<Integer> iterator();
}
```

Figur 2.4: Eksempel på aktuelle typer

I en heterogen implementasjon (som i C++) gjøres det også i praksis på denne måten. Der blir det laget en ny kopi av koden hver gang den brukes med en ny aktuell type-parameter. I en homogen implementasjon blir derimot deklarasjonen av noe generisk per definisjon aldri laget i flere eksemplarer. Det blir bare laget én oversettelse av koden, og den blir laget så generell at den kan brukes for alle aktuelle type-parametre etc. under utførelsen.

Generisk programmering tilbyr noen fordeler og en av disse er som sagt muligheten til å skrive mer kortfattet, gjenbrukbar og modularisert kode. Når man skriver en vanlig (ikke-generisk) klasse er det nødvendig å skrive metodene for alle typer man trenger dem for. Som et alternativ kan man altså benytte seg et programmeringsspråk som har generiske mekanismer, og da klarer det seg med å skrive dem én gang.

Det går et skille mellom de generiske mekanismer der man har begrensninger (”constraints”) på de formelle typene, (og dermed kan sjekke semantikken i for eksempel en generisk klasse), og de som semantikken ikke kan sjekkes før den generiske klassen har fått aktuelle parametre. Templater i C++ er et eksempel på det siste, mens vi i det følgende skal se på mekanismer av den første typen.

2.1 Typeparametrisering i Java

Generics i Java ble lagt til språket i 2004, som en del av den nye J2SE 5.0-standarden. I følge Sun Microsystems tilbyr generisk programmering *“a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the drudgery of casting”.* (5)

Motivasjonen for å innføre generiske typer i Java var blant annet å i større grad kunne utføre typesjekk under kompilering og unngå *ClassCastExceptions* under eksekvering. Kodesnutten i Figur 2.5 viser et typisk problem som kan

oppstå om man ikke benytter seg av typeparametrisering. Meningen fra programmereren er at listen skal inneholde bare Integer-objekter, men ved en feil blir det satt inn et String-objekt. Deretter prøver den å hente tilbake elementet den la til, og kaste det om til en Integer.

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // ClassCastException ved
                                  runtime
```

Figur 2.5: ClassCastException ved runtime

Kompilatoren kompilerer denne koden uten å returnere feil, men det oppstår en *Exception* under eksekvering. Denne typen problemer kan man unngå ved bruk av typeparametrisering. Da kan koden skrives om på følgende vis (i J2SE 5.0):

```
List<Integer> v = new ArrayList<Integer>();
v.add("test"); // ERROR, vil ikke kompileres (typefeil)
Integer i = v.get(0);
```

Figur 2.6: Typefeil ved kompilering

Den aktuelle parametertypen Integer er angitt innenfor spiss-parentesene i figuren over og det blir generert en `ArrayList` av samme type. Vi ser nå at den feilen med innsetting av en string blir oppdaget i kompilatoren. Ettersom typen allerede er kjent er det ikke lengre nødvendig å utføre en type-kasting til ønsket type når man skal hente ut et element fra listen, ettersom resultatet av `v.get(0)` er definert som `String` av kompilatoren. Dermed gir siste linje også en kompilatorfeil.

For å angi begrensninger på de aktuelle typeparametrene tar man i bruk nøkkelordet `extends` for eksempel slik:

```
class GC <T extends A> {
    …
}
```

Figur 2.7: Eksempel på begrensninger på de aktuelle typeparameterne

Dette indikerer at den aktuelle parameter-typen må være en subtype av klassen A for å bli akseptert som korrekt parameter.

Generiske typer kan i Java ikke bare begrenses av spesifikke klasser. Java tillater også såkalte jokertegn (skrives ?) til å spesifisere begrensninger på typeparametrene en gitt generisk klasse kan bli gitt. En generisk klasse kan for eksempel se slik ut:

```
class GC <T extends List <? extends Number>>{
    …
}
```

Figur 2.8: Eksempel på begrensninger på aktuelle typeparameterne med jokertegn

Dette betyr at vi for T for eksempel kan bruke `List<Integer>` som aktuell type. Om vi bare hadde skrevet `class GC <T extends List<number>>{…}` hadde ikke dette vært mulig siden `List<Integer>` ikke er en subtype av `List<Number>`, selv om Integer er en subtype av `Number`. Derfor er jokertegn meget nyttig.

2.2 Package Templates

Den generiske mekanismen vi skal implementere i denne oppgaven heter altså ”Package Templates”, forkortet PT. Den skiller seg på vesentlige punkter fra generiske klasser etc. som er diskutert over. I PT er det en *samling* av klasser, kalt en pakke (som i Java) som er grunnlaget for den generiske mekanismen. Vi skal nå se på denne mekanismen i noe detalj. Hvordan denne mekanismen kan implementeres vil vi diskutere i kapittel 3.

Vi bruker Java-eksempler i dette delkapittelet, siden det er i Java vi skal implementere PT. Kapittelet omhandler Package Templates slik det er fremstilt i (1). Noen av eksemplene er hentet herfra og tilpasset våre behov.

I (1) brukes følgende syntaks for vanlige (Java-) pakker:

```
package P {
    class A {...}
    class B {...}
}
```

Figur 2.9: Syntaks for pakker

For templatene foreslås denne syntaksen:

```
template T <formal type parameters>{

    class C {
        int foo;
        void enMetode( ){ …; }
        ...;
    }
    class D {
        String bar;
        void enMetode( ){ …; }
        ...;
    }
}
```

Figur 2.10: Syntaks for templater

Templatene har altså, bortsett fra typeparametrene, nesten samme syntaks som vanlige pakker. Merk at vi ikke får klasser i tradisjonell forstand før templaten blir *instansiert*. Instansieringen skjer under kompilering. En instansiering av denne pakken inni en vanlig pakke kan se slik ut:

```
package P {
    inst PT with C => G, D => H;

    class G adds{
        String navn;
        void enMetodeTil( ){ ...; }
        ...
    }
    class H adds{
        int lengde;
        void enMetodeTil(){ ...; }
        ...
    }
}
```

Figur 2.11: Syntaks for en pakke med navnendringer og tillegg

Som en del av en slik instansiering blir klassene i pakkene direkte tilgjengelige som om det var vanlige Java-pakker som ble importert. En templat kan bli instansiert flere ganger i det samme programmet. Hver instansiering vil gi et nytt sett av klassene i templaten og disse vil være helt uavhengige av klassene fra andre instansieringer. Hvis en klasse i en templat har statiske variabler vil hver instansiering av klassen få sine egne statiske variabler. Inst-setningen kan vi også angi navneendringer som skal bli utført. Alle stedene klassenavnene C og D forekommer vil de bli byttet ut med G og H.

Man kan også angi tillegg til klassene, og dette blir angitt i adds-delene i Figur 2.11. Vi vil også kalle denne delen for en ”utvidelse” eller et ”tillegg” som blir lagt til i denne instansieringen. I dette tillegget er det lov å angi metoder og variabler (kalt attributter, med et fellesord). Denne koden blir lagt inn i klassen som en del av instansieringen.

2.2.1 Enkelt eksempel

Vi skal nå vise et eksempel som viser tillegg til klasser og navne-endringer. Følgende templat implementerer noen basale egenskaper for grafer:

```
template Graf {
   class Node{
      Kant forsteKant;
      Kant settInnKantTil(Node til){ ... }
      void vis(){ ... }
      ...
   }
   class Kant{
      Node fra, til;
      Kant nesteKant;
      void slettMeg(){ ... }
      void vis(){ ... }
      ...
   }
}
```

Figur 2.12: Enkelt eksempel med grafer

I eksempelet nedenfor blir så denne templaten instansiert som en del av pakken `VeiOgByGraf`, for å bruke templatens graf-implementasjon til å implementere strukturer av byer og veier. I denne instansieringen får `Node`

navnet By og Kant navnet Vei. Begge klassene får også et tillegg med flere attributter (angitt som ”adds-del” til klasse-deklarasjonene med de nye klassenavnene):

```
package VeiOgByGraf{
   inst Graf with Node => By, Kant => Vei;

   class By adds{
    String navn;
    void enMetode( ){
        ...
        int n = forsteKant.length; // 1
        By c = ... ;
        Vei v = settInnKantTil(By til); // 2
        ...
    }
    ...
   }
  class Vei adds{
   int lengde;
   void enMetode(){
      ...
      String s = to.navn; // 3
      ...
      int n = nesteKant.length; // 4
      ...
   }
   ...
}
```

Figur 2.13: Eksempel på pakkeinstansiering

Merk at instansieringen logisk sett også inkluderer By og Vei der tilleggene blir gitt. Forekomster av navnene Node og Kant vil i Graf-templaten derfor bli forandret til By og Vei.

Som en del av instansieringen av Graf vil Node og Kant forsvinne som typer og typingen i pakken VeiOgByGraf vil være som om den var laget direkte for byer og veier. Utsagnene merket med 1, 2, 3 og 4 i Figur 2.13 vil derfor være riktig typet. Ingen objekter av klassene Node og Kant vil bli generert, siden også ”new Node()”vil bli endret til ”new By()”, og det vil heller ikke være noen variabler, parametre eller metoder som er typet med de gamle klassene.

Legg merke til at i utsagn 2 (i Figur 2.13) vil det ikke være noen type-problemer siden metoden `settInnKantTil` har fått følgende signatur i pakken `VeiOgByGraf`:

```
Vei settInnKantTil (By til){ ... }
```

Figur 2.14: Typeendring av metoder

Metoder som er definert i en klasse i templaten kan bli altså overstyrt av en metode som blir definert i et tillegg til en templat-klasse som en del av instansieringen. Hvis et tillegg til klassen `By` har en ny definisjon av metoden `settInnKantTil` typet som metoden over, så vil dette overstyre metoden `settInnKantTil` i klassen `Node` (selv om typingen ikke umiddelbart ser lik ut).

Alle klassedefinisjoner fra en templat blir til vanlige klassedefinisjoner som en del av en instansiering. Hvis en klasse ikke får nytt navn og ingen tillegg blir definert, så blir klassen i pakken slik den er definert i templaten.

Det å bytte navn er ikke bare tillatt for klasser når man instansierer en templat. Man kan for eksempel bytte navn på metoder slik som `settInnKantTil` til `settInnVeiTil`. Syntaksen for dette er:

```
inst Graf with
    Node => By (settInnKantTil -> settInnVeiTil), ...
```

Figur 2.15: Eksempel på endring av navn på metoder

Det er viktig å merke seg at det kun er lovlig å endre navnet på deklarasjonene gjort i templaten selv (inkludert templater som har blitt instansiert i denne, se under). Dette ekskluderer navn som stammer fra eksterne grensesnitt som er implementert av templat-klasser.

2.2.2 Flere instansieringer og instansiering av templater inni templater

En templat kan bli instansiert inni en annen templat. Hver gang den ytterste templaten blir instansiert vil den innerste templaten også bli instansiert. For eksempel kan man i pakken `VeiOgByGraf` bytte ut nøkkelordet *package* med *template* og programmet vil fremdeles være lovlig. Ved å instansiere

`VeiOgByGraf` kan man definere enda flere tillegg i klassene `By` og `Vei`. Sykliske strukturer av templater som instansierer templater ikke er lov.

Som nevnt kan man instansiere samme templat flere ganger i samme skop. Vi kan for eksempel bruke Graf-templaten til å lage klassene `By` og `Vei` som over, og i samme skop bruke dem til å lage et system for et strømnett med kraftledninger og trafoer.

Når man instansierer samme pakke mer enn én gang i et skop må man endre navn på de instansierte klassene slik at man ikke får klasser med samme navn i skopet. Hvis man for eksempel ikke endrer navn på Kant-klassen ved to instansieringer av `Graf` i samme skop betyr det ikke at Kant-klassen er felles for instansieringene, men man vil få en feil ved at man får to klasser med samme navn.

2.2.3 Klassestrukturen i en templat

I en templat kan man definere en vanlig klassestruktur med subklasser. Alle klasser i en slik struktur kan få tillegg og nye navn når man instansierer den, og ikke bare bladklassene. Figur 2.16 til Figur 2.18 viser et eksempel på dette:

```
template Trafikksimulator {
  class Automobil{
    int fart;
    Automobil neste;
    ...;
  }
  class Bil extends Automobil{
    int antallSeter;
    ...;
  }
  class Varebil extends Automobil{
    int lengde;
    ...;
  }
}
```

Figur 2.16: Eksempel på bruk av subklasser

Og bruken av det:

```
package YrkesTrafikksimulator {
  inst Trafikksimulator with
    Automobil => Kjoretoy, Bil => Privatbil,
    Varebil => Arbeidsbil;

  class Kjoretoy adds{
    string merke;
    ...;
  }
  class Privatbil adds{
    int bagasjeromsVolum;
    ...;
  }
  class Arbeidsbil adds{
    int nyttelast;
    ...;
  }
  ...
}
```

Figur 2.17: Bruk av sub-klasser i en pakke

Resultatet blir omtrent som klassestrukturen YrkesTrafikksimulator skissert under:

```
class Kjoretoy{
  // attributter fra Automobil med
navnendringer:
  int fart;
  Kjoretoy neste; ...;
  // tillegg i Kjoretoy:
  String merke; ...;
}
class Privatbil extends Kjoretoy{
  // attributter fra Bil med navnendringer:
  int antallSeter; ...;
  // tillegg i Privatbil:
  int bagasjeVolum; ...;
}
class Varebil extends Kjoretoy{
  // attributter fra Varebil med navnendringer
  int lengde; ...;
  // tillegg gjort i Arbeidsbil:
  int nyttelast; ...;
}
```

Figur 2.18: Tekstlig eksempel på ferdig kode

I koden over ser man at alle klassene har fått de nye og riktige typene. Som nevnt tidligere kan en metode definert i `Automobil` bli redefinert i klassen `Kjoretoy`. Da blir kall til denne metoden, både fra `Automobil` og `Bil`, kall til metoden definert i klassen `Kjoretoy` med mindre den ikke er redefinert i klassen `Bil` også. I så fall blir de samme kallene gjort til denne metoden.

2.2.4 Templater med typeparametere

Som nevnt tidligere kan templater også ha typeparametre. Kodeeksempelet i Figur 2.19 viser en templat som definerer linkede lister hvor hver liste har et objekt og et antall elementer. Hver av disse er representert av et internt objekt (av klassen `AuxElem`) som refererer til det virkelige listeelementet (objekt av klassen `E`). Dette betyr at et gitt E-objekt kan være i mange forskjellige lister og det kan opptre mer enn én gang i samme liste.

```
template ListeAv<E>{
  class Liste{
     AuxElem forste, siste;
     void settInnBakerst(E e){ ... }
     E fjernForste(){ ... }
  }
  class AuxElem{
     AuxElem neste;
     E e;  // Referanse til det virkelige elementet
  }
}
```

Figur 2.19: Eksempel på en templat med typeparametere

Man kan også oppnå et liknende resultat med generiske klasser (klasser med typeparametre). Fordelen med templater og typeparametre er at etter instansieringen av `ListeAv`, for eksempel med parametren `Person`, trenger vi ikke å nevne denne parametren i hver forekomst av "`new Liste`", slik vi må når vi bruker generiske klasser. Både klassen `Liste` og klassen `AuxElem` må ha typeparametre og de må også ha samme aktuelle parametre for å lage konsistente lister når man bruker generiske klasser. Denne situasjonen, at den samme typeparametren må gis til mange klasser, inntreffer ofte og da er det fint å kunne definere typeparametren på pakkenivå.

Det kan angis begrensninger (”constraints”) på typeparametrene og disse skal spesifiseres inne i templaten. Som et eksempel kan vi anta at hvert objekt av klassen E skal holde orden på hvor mange ganger den har vært brukt i en liste. For å implementere dette kan vi kreve at klassen E har to metoder ”`void hevAntall()`” og ”`void senkAntall()`”, som blir kalt i metodene `settInnBakerst` og `fjernForste`. Et eksempel på bruk av dette er:

```
template ListeAv<E>{
  constraint E has{void hevAntall(); void senkAntall();}
  class Liste{
     AuxElem forste, siste;
     void settInnBakerst(E e){e.hevAntall(); ... }
     E fjernForste(){forste.senkAntall; ... }
  }
  class AuxElem extends E{
     AuxElem neste;
     E e;  // Referanse til det reelle objektet
  }
}
```

Figur 2.20: Eksempel på begrensning av typeparametere

Vi har begrenset E ved å liste opp, etter nøkkelordet *has,* et antall metoder som den må implementere. Man kan også begrense en typeparameter ved å liste opp et antall grensesnitt den må implementere, eller en klasse den må være en subklasse av. Ved å angi begrensningene inne i templaten kan vi naturlig også begrense typeparametrene ved en klasse som er definert inne i templaten, og dette kan være beleilig i enkelte tilfeller.

2.2.5 Sammenslåing av templatklasser under instansiering

Det viser seg at det er veldig nyttig ved instansiering av to eller flere templater at klassene i én templat kan bli slått sammen med klasser i andre templater. Dette gjøres ved at to eller flere klasser fra forskjellige templat-instansieringer deler en felles tilleggs-klasse. Dette vil da ende opp som en klasse med navnet til tilleggs-klassen, og den nye klassen får unionen av alle attributtene av de instansierte klassene (alle med den gitte navnendringen), sammen med attributtene i tilleggs-klassen. Man sier da i PT-terminologi at de to klassene er *sammenslått* (”merged”). Eksempelet under vil illustrere hvordan denne mekanismen virker og hvordan den kan være nyttig.

Som i eksemplet i kap. 2.2.1, kan vi anta at vi har Graf-templaten som er angitt der, og at vi igjen vil bruke denne som en basis for å lage klassene By og Vei. Dog, i stedet for å legge til de ekstra attributtene vi vil ha i By og Vei i tilleggs-klassene, kan vi denne gangen anta at vi har en annen templat, GeografiData, med klassene ByData og VeiData hvor disse ekstra attributtene er definert.

```
template GeografiData {
   class ByData{String navn; ...;}
   class VeiData{int lengde; ...;}
}
```

Figur 2.21: Eksempel på en templat

Nå kan vi definere klassene By og Vei som sammenslåing av Node og ByData, og av Kant og VeiData, med tilleggs-klassene By og Vei. Dette gjør vi ved å instansiere begge templatene slik:

```
package VeiOgByGraf{
  inst Graf
       with Node => By, Kant => Vei;
  inst GeografiData
       with ByData => By, VeiData => Vei;

  class By adds{...}
  class Vei adds{...}
  ...
}
```

Figur 2.22: Sammeslåing av templat-klasser under instansiering

Resultatklassene By og Vei vil ha alle attributtene fra henholdsvis Node og ByData og fra Kant og VeiData. Klassene Node, ByData, Kant og VeiData vil ikke finnes i den nye konteksten. De er, i begge instansieringen, byttet ut med typene By og Vei. Syntaksen for sammenslåing av klasser fra templater kan virke noe indirekte, men ut fra de mange hensyn som må taes er den så langt blitt ansett som den beste. Det er regler for hvordan navnekollisjoner skal behandles i forbindelse med sammenslåing, men disse går vi ikke inn på her.

2.2.6 Hvordan unngå indirekte multippel arv

PT er ment å kunne fungere i språk som ikke tillater generell multippel arv, slik

som for eksempel Java. Men på grunn av sammenslåings-mekanismen nevnt over, kan man indirekte få flere superklasser til en klasse, selv om man ikke skriver det eksplisitt. Som et eksempel kan man anta at templat-klassene `Node` og `ByData` i eksempelet over har henholdsvis superklassene A og B. Da vil klassen `By` indirekte få de to superklassene A og B. Dette vil man unngå og man må da lage noen restriksjoner.

Den første restriksjonen er så enkel som at man forbyr templater å ha superklasser definert utenfor templaten. Derfor må man også forby templat-klasser å ha typeparametre (klasser) som superklasser. Dette ser kanskje litt drastisk ut, men man må huske på at templat-klasser fortsatt fritt kan implementere grensesnitt definert utenfor templaten, og dette vil i stor grad gjøre opp for ulempene superklasse-regelen vil føre med seg.

Man vil imidlertid ikke ha så drastiske restriksjoner inni en templat siden dette ville ødelegge muligheten til å ha hierarkier av klasser inne i en templat, noe som er betraktet som meget verdifullt. Derfor er følgende regel introduserert:

```
Hvis, i et sett av instansieringer, to eller flere
templat-klasser erblir sammenslått, må de superklassene
de har (som, hvis de eksisterer, også ervære templat-
klasser) også bli sammenslått i de samme
instansieringene.
```

Figur 2.23: Regel for å hindre multippel arv, kopiert fra (1)

Det må også legges til et krav om at en sammenslåing ikke må resultere i en syklisk superklassestruktur.

2.2.7 Om sammenslåing og konstruktører

La oss se på situasjonen når to eller flere klasser fra forskjellige instansieringer blir sammenslått. Vi tenker oss for eksempel følgende templater:

```
template T {
  class A{...; {
                ...;
      A(C c, D d){...init. kode ...}
                ...;
  }
}
template U {
  class B{...; {
                ...;
      B(F f){...init. kode ...}
                ...;
  }
}
```

Figur 2.24: Eksempel på en templat før sammenslåing

T og U blir så instansiert, og klassene A og B blir sammenslått under navnet X, som følger:

```
package P {
  inst T with A => X;
  inst U with B => X;

  class X adds{ ...se nedenfor ... }
}
```

Figur 2.25: Eksempel på sammenslåing av templater

Her må klassen X få konstruktører som passer til både de gitt i A og de gitt i B, men dette er helt klart vanskelig å gjøre automatisk. Derfor kreves det at alle nødvendige konstruktører blir gitt eksplisitt i tillegget X. Dog, inni de nye konstruktørene må vi være i stand til å kalle konstruktørene til A og B. Det ordinære nøkkelordet ”super” vil nå være tvetydig. Navnene A og B finnes heller ikke lengre etter instansieringen siden begge er erstattet med X.

Løsningen er å tillate at navnene A og B blir brukt i tillegget X, men kun i svært spesielle situasjoner: `super[A]` og `super[B]`. De nødvendige konstruktørene til X kan da skrives på følgende måte:

```
class X adds{
  ...;
  X(C c, D d){
    ...;
    super[A](c1, d1);
    super[B](f1);
    ...;
  }
  X(F f){
    ...;
    super[B](f2);
    ...;
    super[A](c2, d2);
    ...;
  }
  ...;
}
```

Figur 2.26: Eksempel på tillegg til en klasse ved sammenslåing av to templater

2.2.8 Om sammenslåing og virtuelle metoder

Vi får liknende problemer som over når vi har virtuelle metoder med samme navn og parametertyper definert i mer enn én av de sammenslåtte klassene. Vi må derfor kreve at i slike situasjoner må de virtuelle metodene bli redefinert i tilleggs-klassen. Vi trenger da en måte å kalle versjonene i hver av de sammenslåtte templat-klassene og (i samme stil som over) kan vi gjøre dette på følgende måte (hvor ”vm” er navnet til den virtuelle metoden):

```
package P {
 inst T with A => X; //A har virtual int vm(c,d)
 inst U with B => X; //B har virtual int vm(c,d)

 class X adds{
    void vm(C c, D d){
        ...;
        int a = super[A].vm(c1, d1);
        int b = super[B].vm(c2, d2));
        ...;
    }
 }
}
```

Figur 2.27: Eksempel på virtuelle metoder ved sammenslåing av to templater

Kapittel 3

Implementasjonsalternativer

I dette kapittelet skal vi presentere og diskutere de valgene vi måtte gjøre angående fremgangsmåte for å implementere Package Templates. Vi har valgt å dele denne diskusjonen opp i to hoveddeler, det som angår frontend- og det som angår backend-delen av kompilatoren. Mellom disse delene har vi et del-kapittel som skisserer hvordan typede oo-språk med enkel arv tradisjonelt organiseres under kjøring.

I delkapittelet om frontend-delen skal vi se på følgende to hovedalternativer: det å skrive hele kompilatoren fra bunnen av selv eller å ta en eksisterende kompilator og utvide den. I forbindelse med det siste alternativet har vi valgt ut to eksisterende kompilatorer for Java som kan være interessante i denne sammenheng. Vi skal se nærmere på hvordan disse er bygd opp og hvordan de kan fungere som et grunnlag for å implementere Package Templates.

I delkapittelet om backend-delen skal vi presentere en homogen og en heterogen implementasjon av (en tidligere versjon av) Package Templates. Den homogene implementasjonen er laget av Sørensen (3) mens den heterogene er laget av Blomfeldt (4).

Til slutt i dette kapittelet skal vi oppsummere det vi har kommet fram til og velge hva slags løsning vi vil gå for både angående frontenden og backenden.

3.1 Frontend-delen

Frontenden er den delen av kompilatoren som analyserer kildekoden til programmet og bygger opp en mellomrepresentasjon av programmet. Denne

prosessen kan i hovedsak deles opp i tre faser; leksikalsk analyse, syntaktisk analyse og semantisk analyse.

Leksikalsk analyse skal sette sammen enkelttegn til ”ord”, gjerne omtalt som *leksemer*, og avgjøre hvilken overordnet betydning leksemet har. Denne analysen ser ikke på hvilken sammenheng leksemene står i, men bygger kun på hvilken tegnsekvens de består av.

Den syntaktiske analysen parserer sekvensen av leksemer for å identifisere den syntaktiske strukturen av programmet. Denne fasen bygger et (abstrakt) syntakstre, som erstatter den lineære strukturen av leksemer i programmet. Denne trestrukturen er bygget opp etter grammatikken som definerer språkets syntaks.

I den semantiske analysen legger kompilatoren inn semantisk informasjon i syntakstreet og bygger gjerne opp en egen *symboltabell*. En symboltabell holder orden på de deklarasjonsnavnene som finnes i programmet og hva som er synlig hvor. Symboltabellen har funksjonen *lookup(deklarasjonsnavn)*, som leverer deklarasjonen navnet skal bindes opp mot ut fra den angivelsen funksjonen kalles i. Denne fasen utfører semantiske sjekker slik som typesjekking, objekt-binding, og den gir advarsler eller stopper programmer med feil. Den semantiske analysen trenger som regel et komplett syntakstre, og det betyr at denne fasen logisk sett følger etter parse-fasen og kommer før kodegenereringsfasen.

Slik som vi ser det er det i hovedsak to måter å lage frontenden på. Den ene er å skrive den selv fra bunnen mens den andre er å ta en eksisterende kompilator og utvide den. Vi skal nå se nærmere på disse valgene og hvilke kompilatorer som kan være aktuelle å utvide.

3.1.1 Skrive en kompilator fra bunnen

Om vi velger å skrive en kompilator fra bunnen av vil det være mange valg vi må ta, for eksempel hvilket språk kompilatoren skal skrives i. Det mest relevante for vår del er språket Java. Dette språket har vi mest erfaring med fra andre kurs samtidig som språket er godt utbredt. Andre aktuelle språk kunne vært C++ eller C# for å nevne noen. Det taler imidlertid for Java at vi

allerede har kjennskap til gode verktøy for å lage leksere og parsere som bruker Java som implementeringsspråk.

En del av disse verktøyene er laget for å virke sammen. En av disse kombinasjonene er JFlex og CUP, hvor JFlex tar seg av det leksikalske analyseverktøyet, også kalt en skanner, og CUP er det tilsvarende for parseren. Dette er en kombinasjon vi har brukt ved en tidligere anledning og vet hvordan fungerer.

JFlex er som sagt et verktøy for å lage skannere i Java, og det er også skrevet i Java. JFlex ble laget som en etterlikning av en annen kjent lekser, JLex, men (som begge presiserer) deler disse ingen kode. Noen av fordelene med JFLex som teamet bak fremhever, er at den lager en rask skanner, har full unicode-støtte, har en god mekanisme for syntaksspesifikasjon og en sterk plattform-uavhengighet. De leksikalske reglene i en JFlex-spesifikasjon består av et sett med regulære uttrykk og for hver av disse kan man angi Java-kode som blir eksekvert hver gang skanneren finner en match med det tilhørende regulære uttrykket.

JFlex leser inndatafilen med alle de regulære uttrykkene og lager ut fra det et skanner-program i Java. Når den kjører vil den alltid velge det regulære uttrykket som har det lengste treffet. Det vil si at selv om den har lest et ord som matcher et regulært uttrykk vil den prøve å lese ett tegn til for å se om også det ordet gir treff på et eller annet regulært uttrykk. Til slutt vil den bruke det siste som gav treff.

```
Number            = [0-9]+
Float       = [0-9]+.[0-9]+

%%

"null"            { return new Symbol(sym.NULL, yyline,
                  yycolumn); });}
"new"             { return new Symbol(sym.NEW,yyline,
                  yycolumn); }

{Number}       { return new Symbol(sym.INT_LITERAL, yyline,
            yycolumn, new
            Integer(Integer.parseInt(yytext()))); }
            }

{Float}     {     {return new Symbol(sym.FLOAT_LITERAL,
            yyline, yycolumn, new
            Float(Float.parseFloat(yytext()))); }
```

Figur 3.1: Eksempel på inndata til JFlex

JFlex er i hovedsak laget for å passe sammen med CUP, men den kan også brukes sammen med andre parsere slik som BYacc/J, ANTLR eller kun som en skanner. JFlex er et gratisprogram lisensiert under GPL-lisensen.

CUP er et tilsvarende system som genererer LALR-parsere fra enkle BNF-spesifikasjoner. Den gjør det samme som den kjente YACC parseren for C-programmering, og den har faktisk flere funksjoner enn YACC. CUP er skrevet i Java og bruker som spesifikasjoner en BNF-grammatikk og Java-kode som skal utføres på bestemte punkter under parseringen. I eldre versjoner av CUP var man tvunget til å skrive entydige grammatikker, men nå er det støtte for funksjoner (presedens, assosiativitet etc.) som gjør det mulig å spesifisere prioritet og bindinger for terminalene. Dette betyr at man kan ta i bruk tvetydige grammatikker etter at man har angitt presedenser og bindinger slik at den blir entydig.

```
/* Terminaler (Leksemer returnert av skanneren). */
Terminal          SEMI, PLUSS, MINUS, GANGE, DELE;
Terminal Integer  NUMMER;

/* Ikke terminaler */
non terminal           expr_liste, expr_del;
non terminal Integer   expr, term, factor;

/* Presedens og bindinger*/
precedence left   PLUSS, MINUS;
precedence left   GANGE, DELE;

/* Grammatikk */
expr_liste  ::= expr_liste expr_del
            | expr_part
                        ;
expr_del    ::= expr SEMI;
expr        ::= expr PLUSS expr
            | expr MINUS expr
            | expr GANGE expr
            | expr DELE expr
            | NUMMER
            ;
```

Figur 3.2: Eksempel på grammatikk og presedens i CUP

Vi har begge brukt CUP og JFLEX før og synes de var greie å arbeide med. Siden begge disse fortsatt blir utviklet og vi har kjennskap til dem er de et foretrukket valg dersom vi velger å skrive kompilatoren fra bunnen av.

3.1.2 Jikes

Jikes (6) er et mulig alternativ å starte fra dersom vi vil bygge ut en ferdig Java-kompilator. Det er en Java 1.4 kompilator som skal gi en meget effektiv kode. Den er skrevet i C++ og var opprinnelig utviklet av IBM på T.J. Watson Research Center, men er nå vedlikeholdt av et open source-samfunn og den er OSI-sertifisert open source software. Jikes er en kompilator som kan oversette Java-kode til byte-kode eller til binært maskinkode-format. Jikes er ment å følge både ”The Java Language Specification” og ”The Java Virtual Machine Specification” på en meget streng måte. Det gjør at Jikes ikke støtter supersett eller andre variasjoner av språket. Jikes utfører også en avhengighets-analyse av koden som gir to meget nyttige egenskaper: ”Incremental builds” og makefile-produksjon. Den prøver også å hjelpe programmereren til å skrive bedre kode på forskjellige måter. Blant annet har Jikes alltid prøvd å få til gode forståelige feilmeldinger og advarsler for å bistå

programmereren til å forstå programmeringsfeilene. For eksempel vil Jikes fra versjon 1.19 og senere, si fra om en del vanlige programmeringsfeil, slik som for eksempel ”unreachable statements”.

Det er viktig å nevne at Jikes ikke er, og heller ikke prøver å være, et komplett utvikler-miljø. Den er kun en kommandolinjekompilator. Den må altså ikke ses på som en erstatter for mer komplette verktøy, slik som Eclipse som er et sofistikert grafisk IDE (Integrated Development Environments).

Jikes blir ikke lengre aktivt utviklet eller vedlikeholdt. Den siste versjonen som er gitt ut er versjon 1.22 som kom i 2004. Den støtter delvis Java 5, med tanke på nye klasser, men ikke nye språk-egenskaper.

3.1.3 JastAddJ

JastAddJ er et system for å lage kompilatorer for utvidelser av objekt-orienterte programmeringsspråk, slik som Java. Systemet er utviklet på Universitetet i Lund i Sverige av Görel Hedin, Eva Magnusson og Torbjörn Ekman (7). Tanken bak JastAddJ er at det skal det skal være lett å utvide Java-kompilatoren med ny funksjonalitet. For å definere den nye funksjonaliteten, bruker man Java og JastAdds eget Java-liknede språk, med blant annet en EBNF-parser og muligheten til å angi en attributtgrammatikk.

JastAddJ bygger på en kombinasjon av flere deklarative teknikker (Reference Attributed Grammars, RAG) og imperative teknikker (vanlig Java-kode) til å implementere kompilatoren. RAG er en utvidelse av vanlige attributtgrammatikker der man tillater attributtene å referere til noder i det abstrakte syntakstreet, og attributtene i nodene kan bli aksessert utenifra via disse referansene.

RAG tillater også at navneanalyser blir spesifisert på en enkel måte for språk med komplekse skopingmekanismer, slik som arv i objektorienterte språk. Dette gjør det mulig å bruke det abstrakte syntakstreet i seg selv som en symboltabell og etablere direkte koblinger mellom identifikatorer og deklarasjoner ved hjelp av referanseattributtene. Videre oppførsel, enten det er deklarative eller imperative, kan lett spesifiseres ved å bruke disse koblingene. RAG-modulene bir spesifisert i en utvidelse av Java og blir oversatt til vanlig Java kode av systemet. JastAddJ støtter alle ikke-sirkulære

avhengigheter mellom attributter, og tillater dermed generell multipass kompilering.

Internt er JastAddJ bygd på toppen av en LL-parsergenerator JavaCC, men det meste av JastAddJ er uavhengig av den underliggende parserteknologien. Ved siden av JavaCC bruker JastAddJ programpakken JJTree som sitt underliggende trebygger-system. JJTree tillater at vi på en enkel måte spesifiserer hvilke AST-noder som skal bli generert under analysen. En *stakk* blir brukt til å gi programmereren kontroll over hvilken rekkefølge man skal sette inn de forskjellige nodene, slik at strukturen av det konstruerte AST ikke trenger å være helt lik strukturen gitt av grammatikken. For eksempel kan uttrykk som er analysert til å være en liste bli enkelt gjort om til et binært AST.

Mange eksisterende parsere har bare elementær støtte for de videre fasene i en kompilering. Ofte er støtten begrenset til enkle handlinger under trebyggingen. Systemer som støtter mer avansert behandling er ofte basert på spesielle mekanismer som for eksempel attributt-grammatikker uten RAG og/eller algebraiske spesifikasjoner. Disse systemene har ofte sitt eget formuleringsspråk og det kan være vanskelig å integrere håndskrevet kode, spesielt hvis det er ønskelig å få effektiv kode for objekt-orienterte språk som Java.

3.2 Tradisjonell organisering av oo-språk ved runtime

Vi skal i dette del-kapittelet besktive den tradisjonelle organiseringen ved runtime for vanlig Java, og for andre tradisjonelle objektorienterte språk uten multippel arv for klasser. Dette som bakgrunnsinformasjon til diskusjonen om den tilsvarende implementasjonen for PT.

I et runtime-miljø for slike språk er det alltid noe informasjon om programmet som må være tilgjengelig ved eksekvering. Hvis språket har automatikk for frigjøring av minne, også kjent som ”søppeltømming”, må runtime-miljøet ha full kontroll over objektene som blir generert under kjøring slik at søppeltømmeren kan finne frem til objekter som ikke lenger er i bruk.

I tillegg til å kunne kalle statiske og andre ikke-virtuelle metoder må runtime-miljøet også kunne gjøre kall til de riktige versjonene av virtuelle metoder, samt å håndtere dannelsen av nye objekter og initieringen av dem. Hvis

runtime-miljøet i tillegg skal ha mekanismer for observering (og kanskje til og med modifisering) av egen kode, på engelsk *runtime reflection*, utover vanlig type-kasting, må også informasjonen om klassene, metodene og felter være tilgjenglig ved eksekvering.

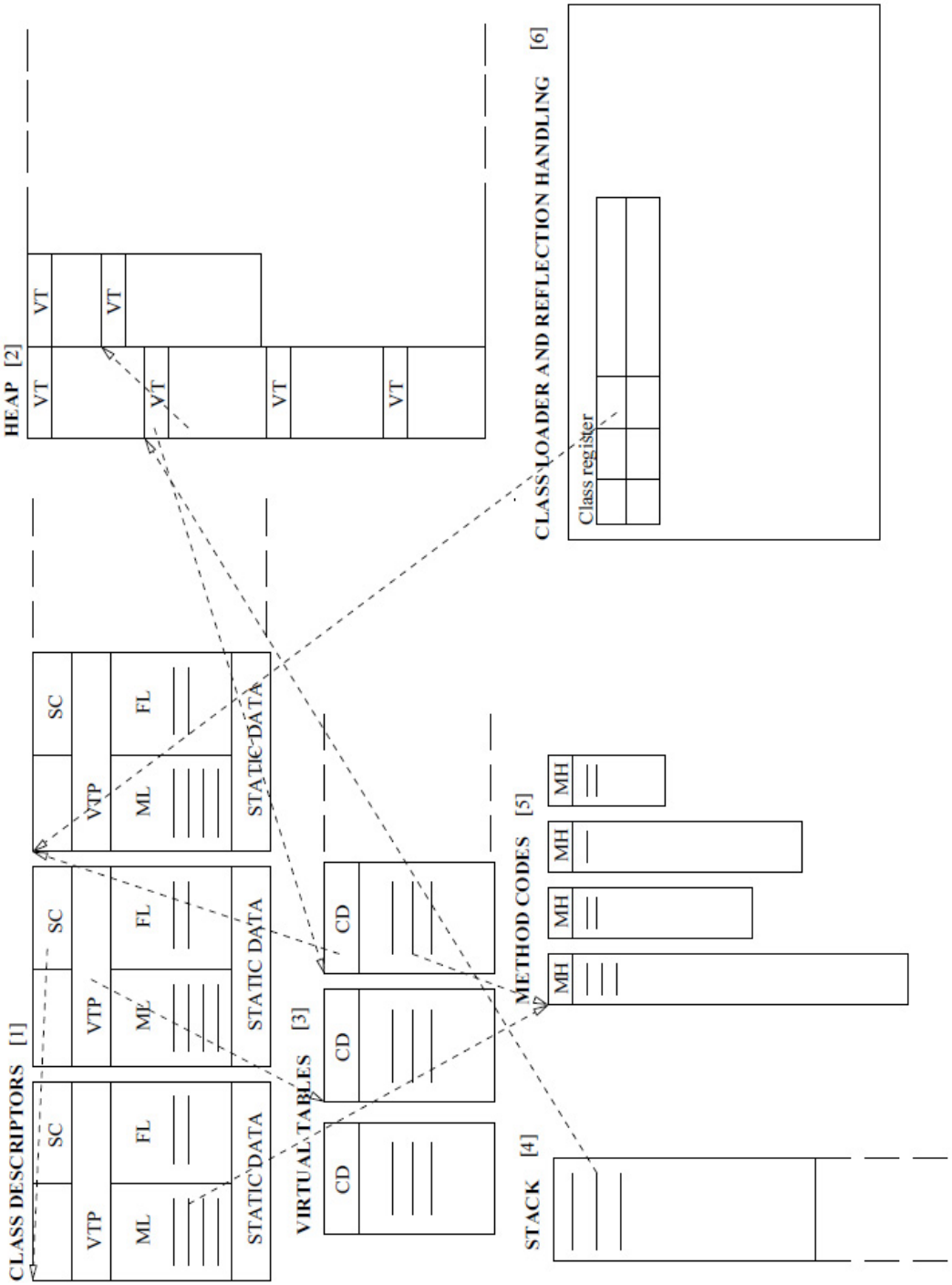
I et runtime-miljø uten generiske mekanismer eller multippel arv, og hvor alle pekere er typet, kan dette gjøres relativt effektivt og på en oversiktlig måte. Men når vi gir muligheten til sammenslåing av klasser ved templat-instansieringer, slik vi tidligere har beskrevet i kapittel 2, blir ting vesentlig mer komplisert. Hvordan dette kan behandles skal vi se på i kapittel 3.3.

3.2.1 Datastruktur

En vanlig datastruktur for runtime-miljøet er å ha en klassedeskriptor for hver klasse i programmet. Denne deskriptoren vil holde relevant informasjon om klassen og dens objekter som det er behov for under eksekvering. Dette kan være

- Størrelsen som trengs i minnet, når det blir opprettet et nytt objekt av klassen.
- Typen av feltene i objektet, og hvor de ligger.
- Virtuelltabellen (denne blir brukt når det blir gjort kall på virtuelle metoder i et objekt av klassen).
- Adressen til deskriptoren for superklassen. Denne brukes bland annet ved type-kasting, og ved instanceof.
- Navnet til klassen og navnene på alle felter og metoder, noe som kan være nyttig ved debugging etc.

Alle objektene som blir opprettet blir lagt på en *heap*. Dette er ledig minne som runtime-miljøet bruker for å allokere nye objekter. Når metoder blir kalt bruker man en såkalt aktiveringsblokk til å holde på parametre, lokale variable og for eksempel informasjon om hvor man skal returnere. For hvert metodekall blir denne informasjonen lagret på en *stakk* slik at aktiveringsblokken til siste kall ligger på toppen av stakken.



Figur 3.3 Organisering av runtime i et objektorientert programmeringsspråk, kopiert med tillatelse (3)

Figur 3.3 skisserer de essensielle datastrukturene i et typisk runtime-miljø for et objektorientert programmeringsspråk. De prikkete linjene viser hvordan strukturen er koblet sammen med pekere. Av de ulike delene et slikt miljø inneholder finner vi:

1. *Klassedeskriptor*
   Denne genereres av kompilatoren/loaderen og ble beskrevet i starten av dette delkapittelet. I tillegg til det som der er nevnt kan den ha en liste med alle metodene (ML), inkludert konstruktørene, og adressen til den respektive koden, samt at også de statiske variable kan legges her.

2. *Heap*
   Heapen inneholder alle objekter med tilhørende data. I hvert objekt av en klasse ligger alltid adressen til virtuellt abellen for klassen. Dermed kan man finne denne og derfra klassedeskriptoren slik at man kan fastslå klassetilhørigheten til et hvilket som helst objekt når man trenger det.

3. *Virtuelltabell*
   Det finnes altså en virtuelltabell for hver klasse, og denne har lengde lik antall virtuelle metoder som det er i klassen, medregnet alle dens superklasser. Disse metodene har alle en egen indeks slik at det skal være lett å slå dem opp i tabellen, og finne hvilken versjon av metoden som skal brukes i objekter av den aktuelle klassen. Virtuelltabellen, kan godt ses på som en del av klassedeskriptoren, men ofte organiserer man det slik at objektene har en peker til virtuelltabellen, og at denne har peker videre til klassedeskriptoren, slik som vist i Figur 3.3.

4. *Stakk*
   En stakk brukes altså av runtime-miljøet til å lagre aktiveringsblokkene for metodekall, slik som omtalt over.

5. *Metoder*
   Det er forskjellige måter å organisere koden til metodene under eksekvering. Noen velger å bruke metainformasjon som legges på

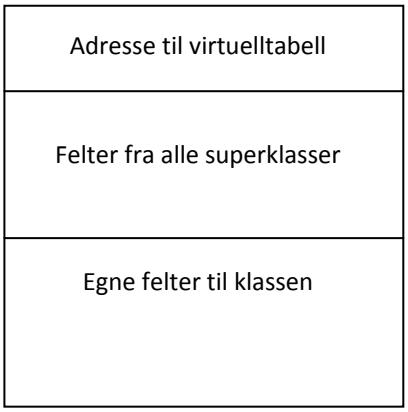
toppen av metoden som en ”header”. Her kan man for eksempel finne informasjon om størrelsen på aktiviserings-blokkene eller lokale variabeler. Det er ikke alltid dette er nødvendig da koden som kaller metoden ofte kan ta seg av minneallokering av de lokale variablene og lignende.

6. *Class loader og reflection handling*
   Med tanke på dynamisk loading har en class loader en oversikt over alle klassene og mer informasjon om dem, for eksempel resten av metainformasjon som kan være nyttig ved for eksempel debugging. Ut fra dette, og med informasjonen fra kompilatoren, setter loaderen opp de endelige runtime-tabellene.

Ved å bruke Figur 3.3 kan vi beskrive de mest grunnleggende oppgavene som blir utført under eksekvering ved hjelp av runtime-miljøet.

3.2.2 Organisering av objekter

Et objekt kan organiseres på mange måter, men en typisk variant er vist i Figur 3.4. Et objekt vil alltid være organisert på slik at feltene fra superklassen kommer før feltene til klassen som arver disse.



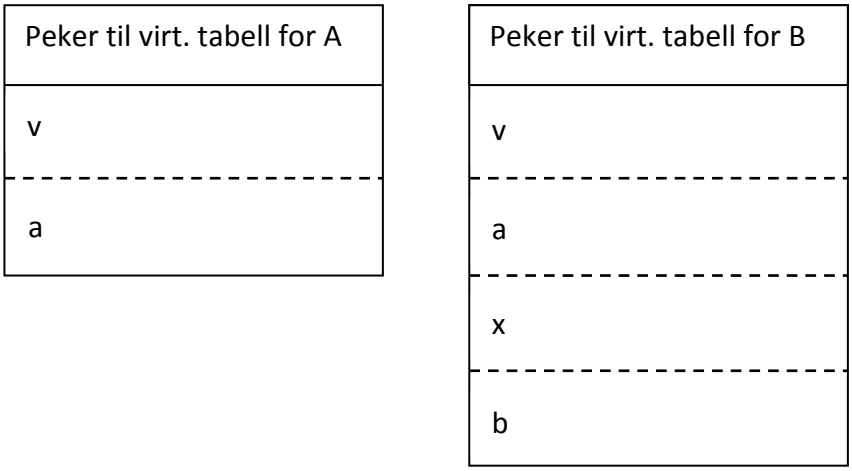
Figur 3.4 Oversikt over objektlayout

Som et eksempel kan vi se på klassene A og B.

```
class A { int v;  A a; static int w; }

class B extends A { int x;  B b;  static int y; }
```

Figur 3.5 Oppretting av klassene A og B i et Java-lignende språk

Klassen B arver fra A, og vi ser da i Figur 3.6 at feltene til A plasseres foran feltene til B.



Figur 3.6 Organisering av felter i klasse A og B

Den relative adressen til feltene i et objekt blir beregnet av kompilatoren og satt inn i kildekoden som aksesserer feltene. Ved en aksess til for eksempel `A.v` eller `B.x` vil den relative adressene til feltet `v` eller `x` bli lagt til adressen til objektet. Dette vil også fungere hvis en variabel av typen A peker på et objekt av klassen B ettersom feltene til A kommer først i B. Senere i oppgaven vil vi se at denne objektlayouten vil by på problemer når vi, som i PT, introduserer sammenslåing av flere klasser (”merging”).

De statiske feltene i klassen blir, som nevnt tidligere, oftest lagret i klassedeskriptoren. Etter at disse blir allokert vil de aldri forflytte seg og man kan derfor sette inn denne absolutt-adressen i byte-koden der man aksesserer feltet. For eksempel vil man for feltet A.w bruke denne adressen for å aksessere feltet direkte.

3.3 Backend-delen og runtime-system for PT

I denne delen vil vi presentere ulike metoder vi kan benytte for å implementere den generiske funksjonaliteten Package Templates skal ha. Som nevnt tidligere er det i all hovedsak to måter å implementere denne på, en homogen og en heterogen metode. Disse metodene har ulike egenskaper, og valget mellom dem må bygges på en avveiing av eksekveringstid og minnebruk.

Hovedtanken bak Sørensens homogene metode er å sette av et bestemt register eller en lokasjon der den relative adressen til starten av feltene i en templat-klasse alltid er tilgjengelig når en metode i denne klassen eksekveres. Denne adressen kan da med fordel lastes inn i et registerer når den generiske koden kalles.

Hovedtanken bak Blomfeldts heterogene metode er, for hver instansiering, å sette inn alle endringene tekstlig i byte-koden, for så og loade denne nye byte-koden sammen øvrig kode.

Etter å ha vært gjennom begge alternativene bør vi ha oversikt nok til å kunne ta et valg om hvilken metode som er mest hensiktsmessig å benytte når vi skal implementere Package Templates.

3.3.1 Litt mer om homogen og heterogen implementasjon

Som nevnt tidligere kan implementeringen av en generisk mekanisme grovt sett skje på to måter. Ved en homogen implementasjon vil man ha en felles runtime-kode for alle instanser av en generisk pakke. Denne koden må da være generell nok til å fungere for alle de aktuelle parametrene og klasse-tillegg man kan komme til å instansiere pakken med.

I en heterogen implementasjon vil man altså for hver instans man oppretter av den generiske koden generere dette som egne kodebiter. Hvis den generiske koden blir brukt mange ganger kan dette føre til en stor utvidelse av runtime-koden, også kjent som ”code bloating”. En fordel med dette er at den genererte koden blir spesialisert for de ulike instansieringene og dermed ikke er nevneverdig tregere å eksekvere enn vanlig ikke-generisk runtime-kode. I en homogen implementasjon vil derimot eksekveringen ofte gå noe saktere.

Ikke alle generiske mekanismer er like egnet til en homogen implementasjon. Hvis man for eksempel ønsker typeparametrisering, men mangler en enhetlig måte å referere til verdier av de forskjellige typer på, bør mekanismen implementeres heterogent. Dette er for eksempel tilfelle i C++ der det brukes en heterogen implementering av *templat*-begrepet, ettersom primitive typer og klasser behandles ulikt. I Java har man også primitive typer, men alle primitive typer har i tillegg en tilsvarende klasse, for eksempel *int* og *Integer*. Dette gjør at en homogen implementasjon kan være aktuell for dette språket.

En forutsetning for å kunne benytte en homogen implementasjon er å ha muligheten til å foreta en statisk typesjekk av koden uten at en faktisk parametrisering er angitt og dette er jo tilfellet for PT.

3.3.2 Sørensens homogene implementasjon av PT

Ved kun å tillate enkel arv i et programmeringsspråk vil organiseringen av runtime-miljøet altså kunne bygges på kjente relative og adresser i objektene slik som angitt i kapittel 3.2. Disse adressene vil på en enkel måte kunne beregnes av kompilatoren og settes inn i runtime-koden. Deretter vil adressene direkte benyttes til effektiv aksess av diverse verdier og objekter under eksekvering.

Det oppstår imidlertid problemer med dette ved multippel arv der en klasse kan arve attributter fra mer enn én annen klasse, samtidig som man ønsker en homogen implementasjon. Når en klasse kan arve fra mer enn én superklasse kan ikke klassen organiseres på samme måte som før. Ved enkel arv er det naturlig å legge feltene til superklassen foran variablene i klassen som arver disse, men ved multippel arv vil ikke dette fungere, ettersom en peker til et objekt av subklassen, typet med en av superklassene, ikke vil vite om superklassens variable ligger på toppen av subklasse-objektet, eller om andre superklasser har fått denne plassen. Og det er alltid én som ikke ligger først.

PT har ikke multippel arv i vanlig form, som for eksempel i C++. Men i forbindelse med et antall templat-instansieringer kan flere klasser fra forskjellige templater bli sammenslått (”merged”) til én klasse, og få et felles tillegg (gitt i en ”adds-del”).

For en homogen implementasjon av PT gir dette omtrent samme problemer som for multippel arv. Man skal da lage ferdig i kompilatoren én kjørbar kode for hver templat og de relative adressene i klassene skal virke for alle instansieringer av templaten, også de der klassene slås sammen med andre, og dens variable ikke blir liggende først.

For å implementere dette bruker Sørensen en datastruktur ved runtime som kommer i tillegg til deskriptorene for de forskjellige klassene. Denne strukturen inneholder de relative adressene til de forskjellige delene av objektet og av den virtuelle tabellen, samt annen informasjon om klassene og deres konstruktører. Dette er organisert slik at det kan benyttes effektivt under kjøring, og disse tabellene kalles *Package Instance Descriptors (PIDs)* (3). De blir generert av kompilatoren/loaderen for hver instans av en generisk pakke. Inne i hver PID finner vi *Class Instance Descriptors (CIDs)* (3) for alle klasser som finnes i pakken og i pakker den har instansiert. De ulike PIDene for de ulike instansene av den generiske pakken har lik layout, men forskjellig data. Den relative adressen til informasjonen inne i PIDen er dermed kjent ved *load-time* og kan settes inn i runtime-koden. Når den generiske koden kjøres vil PIDen til den aktuelle instansiering alltid være tilgjengelig, for eksempel i et bestemt register.

Ved hvert metodekall til en metode definert i en templat sender man da med, som en ekstra parameter, adressen til den instansieringen der metoden er definert. Ved kall gjort utenfor de generiske pakkene er denne PID-adressen kjent av kompilatoren, men når kallet gjøres innenfra en templat må en annen teknikk brukes for å skaffe riktig PID-adresse til kallet.

Et viktig poeng ved Sørensens metode er at han legger til noen nye bytekode-instruksjoner. Det er disse som slår opp i PID- og CID-tabellene, slik at dette kan gå greit og raskt.

3.3.2.1 Et generelt eksempel

Da Sørensen utformet sin metode forelå det en litt annen spesifikasjon av PT enn den som gjelder i dag. De fleste forskjellene er relatert til at PT er en utvidelse av det han implementerte ved at han kun tillater at ”noen” klasser i en pakketemplat er såkalt ”ekspanderbare” (altså, kan få en adds-del). I eksemplet under, som er hentet fra (3), vil derfor ikke alt være som for den

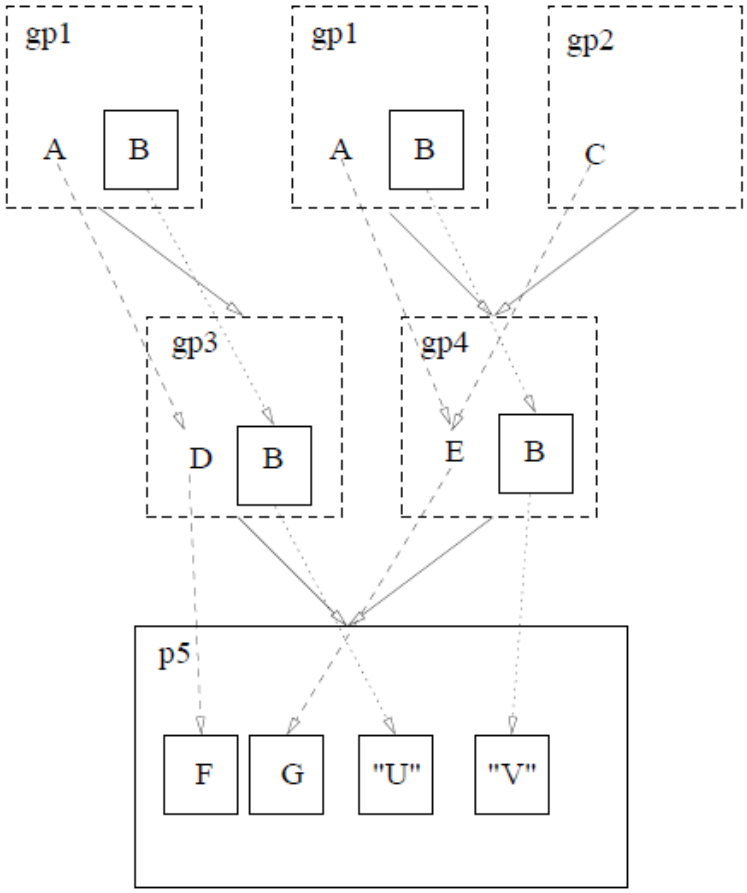
spesifikasjonen vi jobber med. For eksempel vil *package generic* tilsvare begrepet *templat* og *ekspandable* angir hvilke klasser som kan ekspanderes. Endelig vil *import generic* tilsvare det som i dag skrives ”inst”, og *expansion* vil tilsvare ”adds”.

Vi ser på følgende eksempel:

```
// gp1:
package generic gp1 expandable A;
public class A { ... }
public class B { ... }

// gp2:
package generic gp2 expandable C;
public class C { ... }

// gp3:
package generic gp3 expandable D;
import generic gp1 with A -> D;
public class D expansion { ... }

// gp4:
package generic gp4 expandable E;
import generic gp1 with A -> E;
import generic gp2 with C -> E;
public class E expansion { ... }

// p5:
package p5;
import generic gp3 with D -> F rename B -> U;
import generic gp4 with E -> G rename B -> V;
public class F expansion { ... }
public class G expansion { ... }
```

Figur 3.7: Kodeeksempel fra tidligere PT-versjon. kopiert med tillatelse (3)

Som vi ser i Figur 3.7 er *package p5* en ordinær *ikke-generisk* pakke, men den inneholder importer (altså instansieringer, skrevet ”import generic”) av pakker som er generiske. Pakker som p5, som inneholder generiske importer kaller Sørensen *expansion packages* (ekspansjonspakker). Når en klasse fra en slik pakke lastes inn i runtime-miljøet er det en del informasjon som må genereres. Ved hjelp av rekursive kall laster man først inn alle pakkene som ekspansjonspakken importerer, deretter oppretter man datastrukturen bestående av alle CIDene og PIDene.

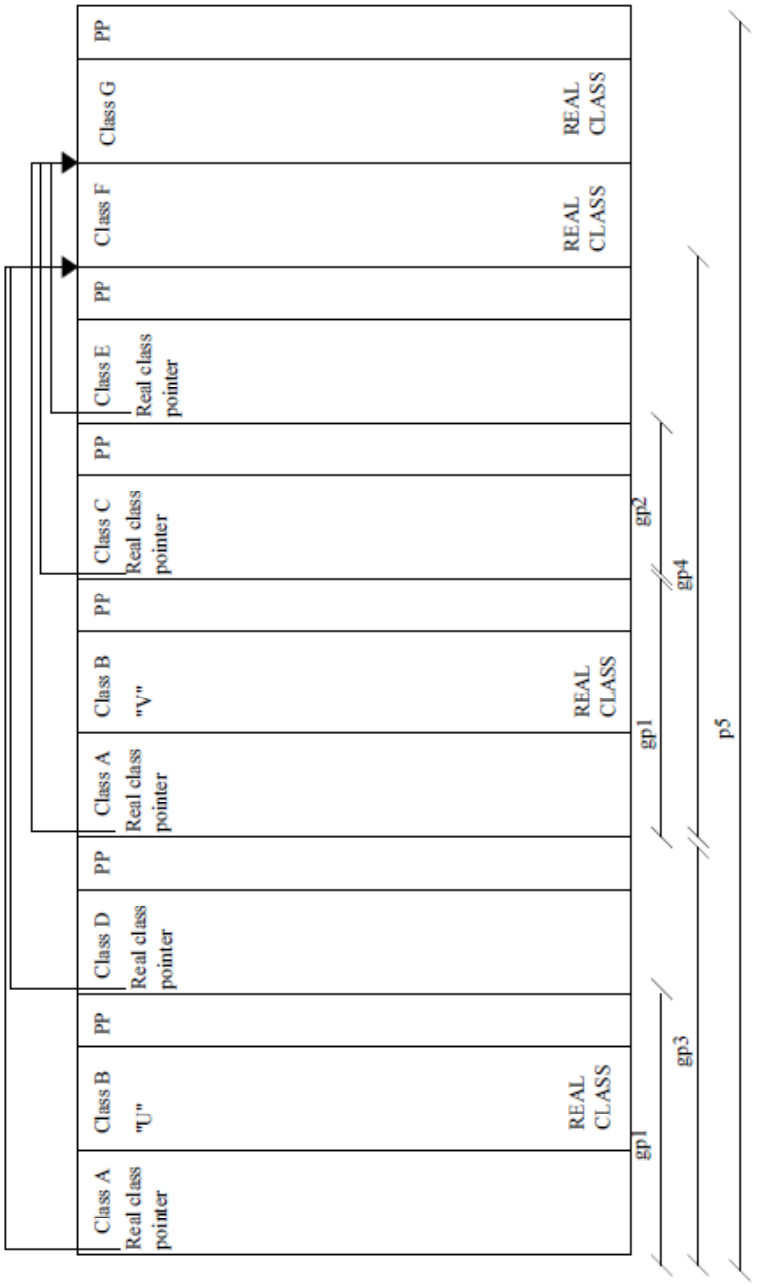


Figur 3.8: Diagram over de ulike pakkene, kopiert med tillatelse (3)

I Figur 3.8 ser vi en oversikt over de ulike pakkene slik de er instansiert i Figur 3.7. Firkantene med stiplet linje representerer instanser av de generiske pakkene gp1, gp2, gp3 og gp4, mens firkanten med heltrukken linje er pakken p5. Vi kan se at gp3 importerer gp1 og at klassen D ekspanderer A. Pakken gp4 importerer både gp1 og gp2, og E ekspanderer klassen A og klassen C. Pakken p5 importerer gp3 og gp4 samtidig som de to klasseinstansene av B får nye navn, henholdsvis U og V.
En PID vil inneholde en CID for hver klasse i pakken og en PID for hver generisk import som blir foretatt. På denne måten vil PIDer opptre inne i andre PIDer, der den ”ytterste” PIDen tilhører en ekspansjonspakke, slik som vi så i p5.

Når en PID blir generert av kompilatoren/loaderen, vil minnet denne opptar bli allokert som et sammenhengende areal. Dermed vil PIDen til ekspansjonspakken p5 oppta et stort sammenhengende minneområdet med

alle de andre CIDene og PIDene fra de generiske importene. PIDene blir laget av en rekursiv metode i postfiks rekkefølge, og det medfører at CIDene til klassen i den generiske pakken som blir importert kommer før ekspansjonspakkens egne CIDer. Figur 3.9 viser PIDen til package p5.



Figur 3.9: PID til package p5, kopiert med tillatelse (3)

Linjene under PIDen i Figur 3.9 viser hvor stort minneområdet PIDen for hver av pakkene opptar. På venstre siden av linjen er adressen til starten av denne instansen. En CID til en ekspanderbar klasse har adressen til CIDen til klassen som den er en del av.

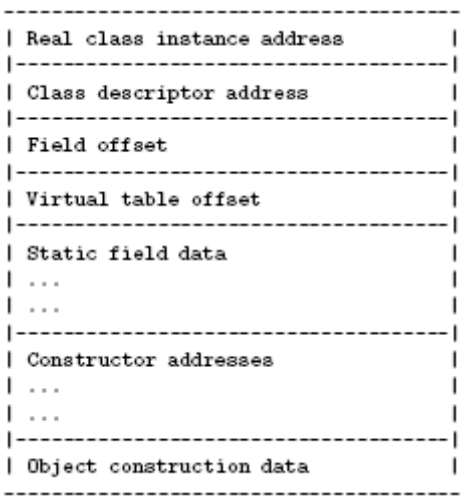
Fra venstre mot høyre kan vi se postfix-organiseringen av klassene samtidig som vi ser at de generiske pakkene gp3 og gp4 er lokalisert innenfor pakke p5, i tillegg er gp1 og gp2 i sin tur inne i gp3 og gp4. Det kommer også frem at gp1 opptrer i to tilfeller innenfor PIDen. Til sammen utgjør dette den komplette strukturen til pakken p5 og vi kan her også greit finne alle CIDene til klassene i layouten.

Klassene det blir generert objekter av, det vil si klasser som ikke er ekspanderbare, kalles *real classes*. Bare (og bare disse) disse klassene vil være synlige utenfor templat-instansieringene i pakken, eksempelvis for pakke p5 vil dette være p5.F, p5.G, p5.U og p5.V.

I tillegg til CIDene blir det generert ordinære klassedeskriptorer (CD) for alle *real class* i pakken. CDene gir støtte for bruk av de endelige klassene fra en pakke ved vanlig ikke-generisk kode. CDene til en *real class* er nesten den samme som en helt ordinær CD, eneste forskjell er at det legges til et felt som holder adressen til CIDen og PIDen til den tilhørende klassen. For helt vanlige klasser utenfor pakketemplatene vil dette være en NULL-pekere. De statiske verdiene i en *real class* vil bli lagret i CIDen og ikke CDen. Ved kall inne fra en pakketemplat til en metode i en utvidbar klasse må adressen til PIDen følge med som en ekstra parameter. Det er viktig å legge merke til at layouten til alle PIDene og CIDene vil forbli lik for alle instanser av en generisk pakke.

3.3.2.2 Oppbygging av CIDene

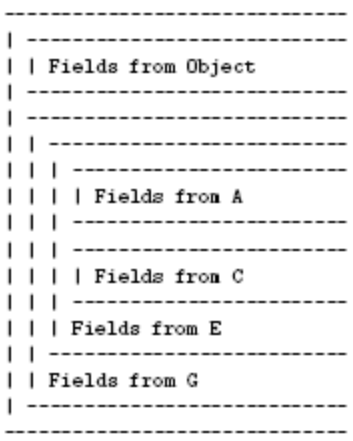
Som nevnt tidligere inneholder PIDen til den komplette pakketemplaten (p5 i eksempelet over) CIDene til alle klasseinstansieringene som er gjort direkte etter indirekte fra pakken p5. Oppbygningen av en slik CID er vist i Figur 3.10.



Figur 3.10 Class Instance Descriptor (CID), kopiert med tillatelse (3)

Øverst i en CID finner man adressen til ”real class”-objektet for klassen. For en ekspanderbar klasse A vil denne adressen angi CIDen til den ikke-ekspanderbare klassen som A er blitt ekspandert til. *Class descriptor address* er, for ”real classes”, adressen til den vanlige klassedeskriptoren til klassen. *Field offset* er den relative adressen til klassens variabeler internt i real class objektet som tilsvarer denne klasseinstansieringen. Deretter kommer *en virtual table offset* som, på tilsvarende måte, er den relative indeksen til metodene i den virtuelle tabellen i *real class*-objektet. Til slutt ser vi det reserverte minneområde for de statiske variablene i klassen, fulgt av adressen(e) til konstruktøren(e) i klassen og data til disse.

3.3.2.3 Oppbygging av et objekt



Figur 3.11 Layout av klassen G, kopiert med tillatelse (3)

I Figur 3.11 ser vi oppbyggingen av et objekt av klasse G, fra Figur 3.7. Her er altså G en ekspansjon av klassen E, der E igjen ekspanderer både A og C.

Inne i et objekt av klassen G finner vi variablene til klassene A, C og E liggende i den naturlige rekkefølge. Relativadressene til disse er lagret som *field offset*, i de tilsvarende CIDene. I eksempelet med klassen G er *field offset* til klassen A og klassen E lik. Field offset vil her være adressen der starten av feltene til klassen A begynner. Field offset til C er adressen til starten av feltene til klassen. G i seg selv har ingen offset da feltene til G blir aksessert på samme måte som en vanlig klasse, der kompilatoren har full oversikt over alle relativadressene.

3.3.2.4 Om hvordan PIDene og CIDene brukes

Hovedideen for bruk av PIDene og CIDene ved runtime er som følger: Når man kaller en metode, si `m(…)`, som opprinnelig er definert i en klasse (utvidbar eller ikke) i en generisk pakke gp, så får kallet på `m(…)` med seg en ekstra parameter som er en peker til PIDen til den aktuelle instansieringen.

Relativ-adresser til CIDene og til andre PIDer inne i PIDen til gp vil være kjent når koden til `m(…)` kompileres/loades, og man kan derfor lett finne de riktige CIDene hvor relativ-adressen til klassens variable i objektet ligger. Når det gjøres et kall på en metode i en indre pakke-instansiering kan vi på samme

måte greit finne den riktigte PID-adressen som skal med som ekstra parameter.

Når man, fra et sted utenfor alle generiske pakker, skal gjøre et kall på en metode definert i en generisk pakke, må også dette kallet ha med en peker til den aktuelle PIDen. Dette går greit for i dette tilfellet er plasseringen av den riktige PIDen kjent av kompilatoren/loaderen.

Kall på virtuelle metoder implementeres med virtuelle tabeller knyttet til CDene, omtrent som forklart i kap. 3.2. På samme måte som for en utvidbar klasses variable, må også klassens indekser i virtuelltabellen få et tillegg før de brukes. Dette tillegget står i klassens CID (virtuell table offset). Endelig må elementene i virtuell-tabellen ha noe ekstra informasjon. I tillegg til peker til metode-koden må den inneholde peker til den aktuelle PIDen.

3.3.2.5 Den virtuelle maskinen (JVM)

Den virtuelle maskinen er bygget etter prinsippene beskrevet i kapittel 3.2 der denne har, en klasse-loader, en klasse diskriptor med virtuelle tabeller, en stakk, en heap og en interpreterer.

Maskinen laster klasser fra klasse- eller pakke-filene og eksekverer instruksjonene fra metodene som kalles. Når en klasse fra en pakketemplat blir lastet inn i runtime-miljøet vil alle de generiske importene av pakker bli rekursivt lastet inn i systemet. Deretter vil datastrukturen bestående av CIDer og PIDer bli generert.

Den virtuelle maskinen starter lastingen av et program med hovedklassen. Etter dette søker systemet gjennom byte-koden på utkikk etter referanser til andre klasser som ikke finnes i systemet for deretter å laste inn disse. På denne måten blir alle klassene som trengs under eksekvering funnet og lastet inn i runtime-miljøet slik at eksekveringen av main-metoden i hovedklassen kan starte.

Ved innlasting av en klasse vil layouten til feltene og den virtuelle tabellen settes opp. Når dette har skjedd for alle de nødvendige klassene vil byte-koden til metodene ses gjennom og alle referanser til konstant-området blir erstattet. Dette skjer ved at referansene blir byttet ut med absolutte adresser

til for eksempel klasser og statiske variable eller relative adresser til metoder i den virtuelle tabellen eller felter i et objekt.

Da dette er gjennomført vil byte-koden være klar for interpretereren.

Som nevnt tidligere har Sørensen utvidet interpretereren til støtte en del nye bytekode-kommandoer. Det er 10 stykker, og disse er: getstaticgeneric, putstaticgeneric, getfieldgeneric, putfieldgeneric, invokevirtualgeneric, invokespecialgeneric, invokerealgeneric, invokestaticgeneric, newgeneric og checkcastgeneric.

Under ser vi på hvordan noen av disse nye bytekode-kommandoene virker:

- invokevirtualgeneric
  Denne instruksjonen blir brukt når en virtuell metode i en ekspanderbar klasse blir kalt i en generisk pakke.

```
int x = a.m2(0);
```

  Anta, for koden over, at `m2(int)` er en virtuell metode i klassen A, hvor `a` er lokalvariabel nr 5, returtypen til metoden er en int og at `x` er lokalvariabel nr 6. Da vil kompilatoren generere følgende kode:

```
100: aload 5
102: iconst_0
103: invokevirtualgeneric #25; //Method gpck/A.m2:(I)I
106: istore 6
```

  Konstanten blir bestemt ut fra to verdier. Den første er den relative adressen til virtuell tabell offset (VTO) som ligger i starten av PID, kalt addrVTO videre i eksempelet. Den andre er den relative adressen til metoden i A-klassens del av tabellen, kalt metInd videre i eksempelet. Disse to blir satt inn i byte-koden sammen med størrelsen til parametren.

  Når instruksjonen invokevirtualgeneric blir interpretert bruker interpretereren størrelsen til parametren til å finne objektet metoden skal kalles i. Objektet blir, som man kan se av eksempelet, dyttet på

stakken foran parametrene. Adressen til den virtuelle tabellen blir lest fra starten i objektet. Adressen i VTO blir lest fra summen av addrVTO og currPID. Indeksen til metoden i den virtuelle tabellen blir funnet ved å legge sammen adressen til tabellen, VTO og metInd. På denne adressen ligger adressen til koden til metoden og den riktige PIDen for kallet. Metoden blir så kalt fra denne adressen med den nye PIDen som parameter.

- getstaticgeneric
  Denne instruksjonen blir brukt når et statisk felt i en generisk klasse skal bli aksessert inne i en generisk pakke. Et statisk felt i en generisk klasse er lokalisert i PIDen. Verdien til feltet blir kopiert til toppen av stakken.

```
A tmp = A.a;
```

Vi kan anta, i koden over, at `tmp` for eksempel er lokalvariabel nr 5 og at `a` er et statisk felt i klassen A typet med A. Kompilatoren vil gi følgende kode:

```
100: getstaticgeneric #25; //Field gpck/A.a:Lgpck/A;
103: istore 5
```

Vi kan se at typen til feltet også er en del av konstanten. Typen til feltet er alltid lokaltype til pakken. Det statiske feltet har en relativ adresse fra starten av PIDen som blir beregnet under link-time. Den relative adressen bytter ut indeks 25 når den blir satt inn i byte-koden.

Når getstaticgeneric instruksjonen blir interpretert vil interpretereren legge til den relative adressen til det statiske feltet til currPID. Deretter vil verdien bli kopiert fra denne adressen til toppen av stakken.

- getfieldgeneric
  Denne instruksjonen blir brukt når et felt i et objekt i en ekspanderbar klasse blir aksessert. Verdien til feltet blir kopiert til toppen av stakken.

```
int x = a.y;
```

Anta, i koden over, at x er lokalvariabel nr 6, at a er lokalvariabel nr 4 av typen A og at feltet y i den ekspanderbare klassen A er av typen int.

```
100: aload 4
102: getfieldgeneric #27; //Field gpck/A.y:I;
105: astore 6
```

Som i invokevirtualgeneric trenger man to relativ adresser for å aksessere et felt i en ekspanderbar klasse. Den første er relativ-adressen til feltet y i A sin del av objektet. Den andre er relativadressen til starten av A sine felt i objektet.

CIDen til A inneholder dette felt-offsetet og denne CIDen har en relativadresse fra starten av PIDen som er kjent av kompilatoren. Relativadressen til y innenfor A sin del av objektet er også kjent av kompilatoren. Disse to verdiene kan bli beregnet under link-time og satt inn i byte-koden, slik at interpretereren kan beregne riktig total relativadresse for y i objektet.

3.3.3 Blomfeldts heterogene implementasjon

Blomfeldt (4) har som del sin hovedoppgave laget en heterogen implementasjon av omtrent den samme versjonen av PT som Sørensen brukte. I motsetning til Sørensen har han valgt å utvide en eksisterende kompilator. Han valgte *Espresso* som utgangspunkt for sin kompilator, og hans utvidede kompilator som har fått navnet *COFFEE*.

Blomfeldts masteroppgave omhandler i teorien en blandingsimplementasjon mellom et homogent og heterogent sluttresultat. Blomfeldt sjekker de generiske pakkene separat og kompilerer dem til en slags spesiell byte-kode der de formelle parametrene og andre ting som forandres er merket for lett å kunne substitueres med noe annet. Denne koden, som legges i såkalte GP-

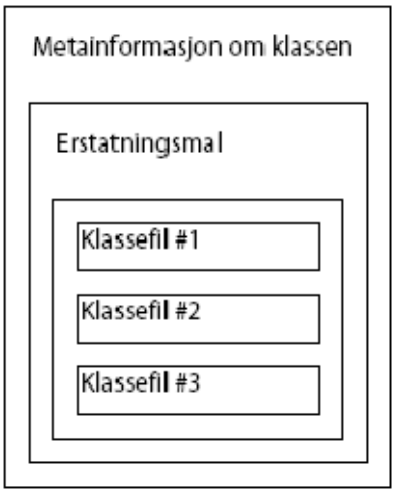
filer, lagres i ett eksemplar for hver generisk pakke. Derved har vi en homogen implementasjon så langt. De spesielle GP-filene blir, i forbindelse med vanlig loading, først kjørt gjennom en pre-loader som setter på plass de aktuelle utvidelsene og parametrene. Deretter gjøres en vanlig loading av dette hver gang det er en instansiering av den generisk pakken. Dermed vil for eksempel riktige adresser og endelige relativadresser i objektene settes direkte inn i koden, som for vanlig Java. I og med at lageret dermed fylles opp med like mange eksemplarer av koden som det er instansieringer, blir dette altså til slutt en heterogen implementasjon.

3.3.3.1 Forenklinger

Blomfeldt har gjort noen forenklinger i språket i sin heterogene løsning. Den størst forenklingen er at han ikke har implementert sammenslåing av flere klasser ved instansiering og dette forenkler implementasjonen relativt mye. Han har heller ikke fullt ut implementert språkets muligheter til å kombinere den generiske mekanismen med typeparametrisering, verken på klassenivå eller pakkenivå.

3.3.3.2 Generering av GP-filer

Når en generisk pakke kompileres vil *C0FFEE* altså lage en såkalt GP-fil for hele den generiske pakken. Denne filen inneholder klassefiler for alle klassene i den generiske pakken, samt informasjonen som trengs i forbindelsen med den generiske importen av pakken. Dette er en slags ”oppskrift” for hvilke substitusjoner etc. som må gjøres under en instansiering av pakken. Klassefilene i GP-filen er strukturert i henhold til *JVM-spesifikasjonen*, men i tillegg til dette vil GP-filene altså inneholde denne substitusjons-oppskriften.



Figur 3.12: Filformatet til GP-filer, kopiert fra (4)

Når *C0FFEE* kompilerer en generisk pakke, vil den først finne ut hvilke navn som må endres i en instansiering av pakken. Dette vil for eksempel alltid innbefatte klassenavn i den generiske pakken og selve navnet på den generiske pakken. Disse navneforekomstene kaller han *erstatningsforekomster*. Det er viktig å finne et godt prinsipp for navngivingen av pakker og klasser slik at man kan få tilbake forståelige feilmeldinger hvis det oppstår kjørefeil i programmet.

Når *C0FFEE*-kompilatoren tolker en generisk import av en generisk pakke, vil den bruke informasjon i GP-filen til pakken til å sjekke at programmet er riktig, og til å lage en såkalt ”erstatningsmal” for denne pakka instansiert på dette stedet i programmet. Erstatningsmalen vil da inneholde tilstrekkelig informasjon til at hans spesielle pre-loader kan lage en såkalt ”erstatningspakke”. Erstatningspakken vil være en ordinær *Java-pakke* som beskriver resultatet av instansieringen av den generiske pakken. All den nødvendige informasjonen fra GP-filen og instansieringen vil da være satt inn i erstatningspakken, og C0FFE vil ha erstattet navnene til de utvidbare klassene med navnene til de aktuelle tilleggs-klassene. Dermed er det erstatningspakkene og de delene som er skrevet som vanlige pakker som til slutt blir loadet som til ordinære klassefiler.

3.3.3.3 Oppsummering

For å oppsummere Blomfeldt sin metode i korte trekk er det minst tre hovedsteg i kompileringen av et program med generiske pakker til ordinær byte-kode (som kan kjøres på en standard JVM). Først må de generiske pakkene kompileres og det må genereres en erstatningsmal for hver av dem som angir erstatningsforekomstene. Deretter må de generiske pakkene instansieres av en pre-loader, og da opprettes en erstatningspakke for hver instansiering. Det er også mulig å ha generiske importsetninger i generiske pakker. Hvis dette er tilfellet legges informasjonen om importen som metainformasjon i den genererte GP-filen, i stedet for at det opprettes en erstatningspakker. Til slutt må programmet, bestående av brukerskrevet kode og erstatningspakkene, slås sammen til ordinære klassefiler. De to første stegene her er det Blomfeldt har jobbet med, mens det siste steget ble dekket av Espresso, den kompilatoren han utvidet, slik den forelå i utgangspunktet.

3.4 Vurderinger

Vi skal her diskutere det vi har beskrevet over og på bakgrunn av det ta valgene om hvordan vi vil organisere oss i vår implementasjon.

3.4.1 Frontend

Et av valgene vi kan gjøre er altså å skrive frontend-delen selv fra bunnen av. Siden vi skal implementere Package Templates for et så komplekst språk som Java er det ikke gjort i en håndvending å skrive kompilatoren og det er mange små og store ting som må behandles på riktig måte for å støtte hele språket (noe som i utgangspunktet var viktig for oss). Det vil ta lang tid å få på plass alt og få det til å virke ordentlig, og den vil mest sannsynlig ikke bli i nærheten av en standard Java-kompilator når det gjelder hurtighet, feilsjekking, feilmeldinger og så videre. Mye av dette arbeidet vil heller ikke være relevant for oss siden vi først og fremst skal se på det å implementere PT. Dermed vil den største delen av arbeidet ikke omhandle essensen i vår oppgave, noe som naturlig nok ikke er ideelt.

Hvis vi skulle implementert Package Templates i en forenklet utgave av Java ville denne løsningen vært en mye ”hetere” kandidat. Fordelen ved å skrive

den selv er at vi da direkte kan tilpasse den til PT og slippe mange ”lure” løsninger som antageligvis må til for å omarbeide en eksisterende kompilator.

Fordelen med å gå ut fra Jikes eller JastAddJ er at de er ferdige systemer, slik at man starter med noe som virker og er gjennomprøvd. Med disse er det riktignok mye å sette seg inn i før man kan begynne å implementere Package Templates, men når dette er gjort bør oppgaven være vesentlig mer overkommelig. Vi har altså valgt at kompilatoren skal kjøres på en vanlig JVM, og grunnen til dette er at vi syns dette vil være enklere for alle som skal bruke denne kompilatoren. Hvis vi hadde valgt å oversette til en utvidet byte-kode (slik Sørensen gjorde) måtte vi også implementert en utvidet JVM.

Om vi ville velge denne løsningen ville for eksempel Kaffe være et godt grunnlag å starte på. Kaffe er en implementasjon av Javas virtuelle maskin med tilhørende klassebibliotek. Kaffe er gratis programvare, lisensiert under GNU Public License. Open source software, som Kaffe, lider imidlertid ofte av at det er dårlig dokumentert, men tilgjengjeld er det ofte gode miljøer rundt dem slik at man får den hjelpen man trenger. Dette gjelder nok til en viss grad både Jikes og Kaffe, men dessverre er det ikke særlig mye miljø rundt disse lenger.

Hovedproblemet med Jikes er at den ikke lengre blir vedlikeholdt og videreutviklet og siste versjon av den er kun for Java 1.4. En annen ting er at Jikes er skrevet i C++ og det er et språk vi har vært lite borti. Vi har ikke funnet noe som viser at Jikes har blitt brukt på samme måte som vi har tenkt til å bruke det, nemlig implementere nye språkbegreper og funksjoner.

Siden JastAddJ er utviklet i undervisningssammenheng finnes det mye god dokumentasjon og gode eksempler på hvordan man kan bruke dette systemet. Der er det også beskrevet hva som må gjøres for å utvide kompilatoren med ny funksjonalitet akkurat slik som vi skal gjøre. JastAddJ er en komplett Java-kompilator noe som gjør at vi kun trenger å fokusere på å implementere Package Templates.

Slik som vi ser det står vi i grunn igjen med to muligheter, skrive den selv eller bruke JastAddJ. Grunnen til at vi bare ser på JastAddJ (og ikke Jikes) er at den i mye større grad er tilrettelagt for å kunne videreutvikles. Dessuten har vi ikke funnet mye teknisk stoff om hvordan Jikes er bygd opp og hvordan det vil

være å videreutvikle den. Både med en egen kompilator og med JastAddJ er man i stor grad avhengig av å utvikle og vedlikeholde kompilatoren selv i fremtiden. Derved, på bakgrunn av det vi nå vet om de to valgene, er det JastAddJ som er det klare valget når man ser på tiden vi har tilgjengelig og på dokumentasjon. Det er selvsagt mye å sette seg inn i JastAddJ, men i forhold til å skrive hele kompilatoren selv er det mye tid å spare på det, og ikke minst vil vi da kunne konsentrere oss om det som er hovedproblemstillingen i oppgaven, nemlig å implementere Package Templates i Java.

3.4.2 Backend

Både Sørensen (3) og Blomfeldt (4) har laget to spennende implementasjoner av tidligere versjoner av Package Templates. Sørensen lagde en egen JVM for å kjøre programmet på en homogen måte, mens Blomfeldt lagde en slags preloader for å kunne kjøre det på en vanlig JVM, men på heterogen måte.

Fordelen med Blomfeldts heterogene metode er at han får oversatt homogent fram til ”nesten” byte-kode. Ulempen med den heterogene implementasjonen er at runtime-koden blir generert for hver instans av den generiske kildekoden. Den generiske kodens størrelse ved runtime vil derved øke lineært for hver instansiering. Hvis templater blir instansiert i stor stil vil det oppstå en ”oppblåsning” av runtime-koden. Dette er et kjent fenomen som også kalles ”code-bloat”. Den heterogene koden er spesialisert for de aktuelle typeparametrene og kan eksekvere like fort som vanlig ikke-generisk kode.

Vi nevnte i starten av delkapittelet om Blomfeldts metode at han har gjort noen forenklinger i sin løsning. Disse var at han ikke har implementert sammenslåing av flere klasser og at han heller ikke fullt ut har implementert muligheten til å kombinere den generiske mekanismen med typeparametrisering, verken på klassenivå eller pakkenivå. Det å skulle gjøre Blomfeldts løsning til en fullstendig implementasjon av PT er mye jobb. Navnekollisjoner er et av problemene som oppstår i forbindelse med sammenslåing av klasser, og som må bli håndtert på en eller annen måte som ikke er så lett å se for seg.

Sørensens metode byr på mye komplisert tabellbruk, og vil kunne bli noe tregere under utførelsen siden det ved aksess og metodekall må gjøres en del

tabelloppslag etc. Fordelen er at metoden er strengt homogen og derved krever mindre plass ved eksekvering enn en heterogen implementasjon.

Det å gjøre om Sørensens metode slik at den vil virke på en vanlig JVM er litt jobb, men, slik som vi skal se i kap. 4, helt overkommelig. Vi ser for oss at mye av den koden Sørensen har skrevet i sin JVM er mulig å bruke på nytt. En mulig løsning er å legge koden til de kallene som må endres i byte-koden i et eget klassebibliotek og heller kalle disse når man treffer på et slikt kall. Det viser seg imidlertid at datastrukturene også må forandres en del, slik at dette skjemaet ikke kan brukes helt rett fram.

Vår konklusjon er at en homogen løsning krever litt mer fikling for at det skal virke, men at vi vil sitte igjen med en mer elegant løsning. På en annen side er den heterogene løsningen mer rett fram, men det er vanskelig å si om den ene løsningen er mer arbeidskrevende enn den andre. Det som er viktig for oss er at det skal kunne kjøres på en helt vanlig, uforandret JVM. Derfor har valget falt på å lage en homogen løsning som kan kjøres på en vanlig JVM, men mest mulig etter de prinsipper som er brukt i Sørensens løsning.

Kapittel 4

Vår implementasjon

I forrige kapittel kom vi fram til at vi skulle bruke JastAddJ som et utgangspunkt i vår kompilator og at vi skulle implementere PT på en homogen måte.

I dette kapittelet skal vi først gi en litt dypere beskrivelse av JastAddJ og deretter gi en kort introduksjon til BCEL, Byte Code Engineering Library, som er et verktøy vi skal bruke i implementasjonen av backenden. BCEL gir oss muligheten til å lese og lage klassefiler og å manipulere dem.

Det sentrale i dette kapittelet er at vi presenterer vår implementasjon av Package Templates. Vi har, i likhet med i kapittel 3, valgt å dele opp presentasjonen i to deler; frontenden og backenden.

I delen om frontenden vil vi beskrive hvilke endringer vi har gjort i JastAddJ for at kompilatoren skal godta tilleggene i syntaksen for PT og gjøre riktige semantikksjekker av de nye delene. Vi skal også skrive litt om problemene som dukket opp underveis og hvordan vi løste dem.

Vi har valgt å avslutte kompileringen ved første feil vi finner, i de sjekkene vi selv har lagt til. Hvordan JastAddJ gjør dette avhenger av hvor alvorlig feilen er. Semantiske feil blir samlet opp og skrevet ut til slutt, men ved mer alvorlige feil blir kompileringen avsluttet.

I delkapittelet om backenden vil vi beskrive den runtime-strukturen vi bruker, og hvilken byte-kode vi produserer. Vi har altså ikke, slik som Sørensen gjorde,

lagt noen egne nye instruksjoner til byte-koden. Derved kan programmene vi produserer kjøres på en vanlig JVM.

Måten vi har tenkt å implementere PT på er at vi lar JastAddJ lage byte-kode som om templatene var mer eller mindre vanlige pakker. Deretter vil vi gå igjennom denne byte-koden og forandre den der det er nødvendig. Disse forandringene går stort sett ut på å bytte ut en eller fler eksisterende instruksjoner med en eller flere nye instruksjoner for, ved runtime, å få plassert ting riktig på stakken og få gjort det som er foreskrevet i PT. Vi må av og til endre på rekkefølgen av objektene på stakken fordi vi sender med et objekt (som tilsvarer Sørensens PID-tabell) som en parameter ekstra til alle metodene. Dette objektet bruker vi fordi det viste seg vanskelig å bruke PIDer og CIDer på tilsvarende måte som Sørensen har gjorde. Siden vi skal bruke ren byte-kode må vi gjøre en del forandringer i forhold til Sørensens datastruktur. Alle disse momentene vil bli nærmere forklart i kap. 4.4.

Vi har valgt å gjøre noen begrensinger ved implementasjonen av PT og i stedet fokusere på å få en mest mulig ferdig kompilator. Disse begrensningene er:

- Ikke lov med arv inne i templater
- Templater kan ikke inneholde instansieringer av andre templater
- Ikke lov med nye konstruktører med parametere
- Ikke lov med grensesnitt i templater
- Ikke lov med typeparametre til templaten

Disse begrensningene forenkler implementasjonen av JPT vesentlig. Den blir mindre tidkrevende, uten at det går alt for mye ut over funksjonaliteten til JPT. Det å fjerne disse begrensningene synes heller ikke å kreve noen prinippielt nye mekanismer.

Ellers sier vi at det ikke skal være lov å ha main-metoder i templater. Dette er fordi en templat ikke skal være mulig å bruke uten at den er instansiert.

4.1 JastAddJ

En første oversikt over JastAddJ ble gitt i kapittel 3.1.3. Her skal vi først gi en noe dypere beskrivelse, basert på måten JastAddJ er fremstilt i (8).

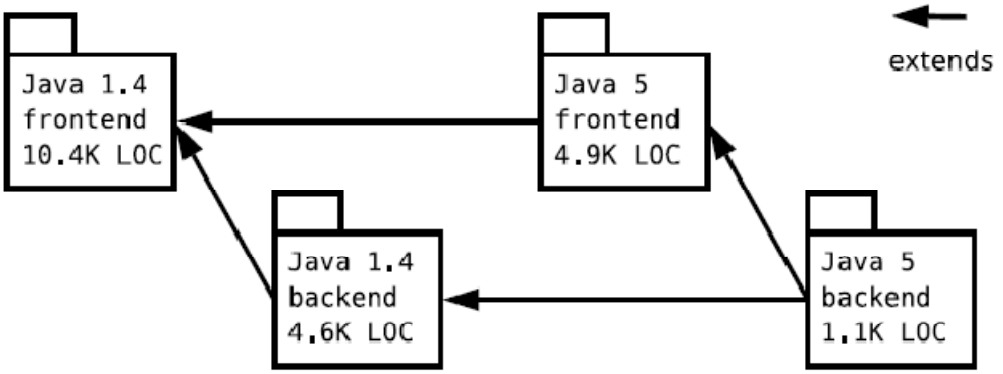
Det er JastAdd Extensible Java Compiler, kalt JastAddJ, vi skal bruke til å implementere Package Templates. Den er bygget på metakompilatoren JastAdd som gir avansert støtte for å lage modulære og utvidbare kompilatorer. Når man bruker den som den er virker den på samme måte som en vanlig Java kompilator, men den kan også enkelt utvides med nye språkbegreper.

4.1.1 Arkitekturen

4.1.1.1 Hovedkomponenter

JastAddJ består av fire hovedkomponenter: en Java 1.4 frontend og en backend, og tillegg til disse to komponenter som gjør den til en Java 5 frontend og backend. Hver av disse fire komponentene er representert som en katalog med gjenbrukbare kildekodefiler (JastAddJ-filer og datafiler til en parser-generator), et hovedprogram (i Java) og en byggefil. Backendene er utvidelser av frontenden og disse er definert i egne tilleggsfiler. Java 5-komponentene er altså kun utvidelser av Java 1.4-komponentene, da disse kun inneholder det som trengs for å utvide Java 1.4-implementasjonen til Java 5. Komponenter for nye utvidelser av språket, analyser eller backender kan bli laget ved å definere komponenter som på samme måte utvider de eksisterende JastAddJ komponentene.

Frontendene parser Java-filer, leser klassefiler den er avhengig av, printer feilmeldinger fra kompileringen og printer normaliserte versjoner av filene, ved siden av å bygge et abstrakt syntakstre. Disse verktøyene viser eksempler på hvordan man lager kilde-til-kilde oversettelsesverktøy og analyseverktøy slik som plug-in-baserte typesjekkere. Hovedprogrammene i backenden parser går gjennom det abstrakte syntakstreet, printer feilmeldinger fra kompileringen og produserer klassefiler.



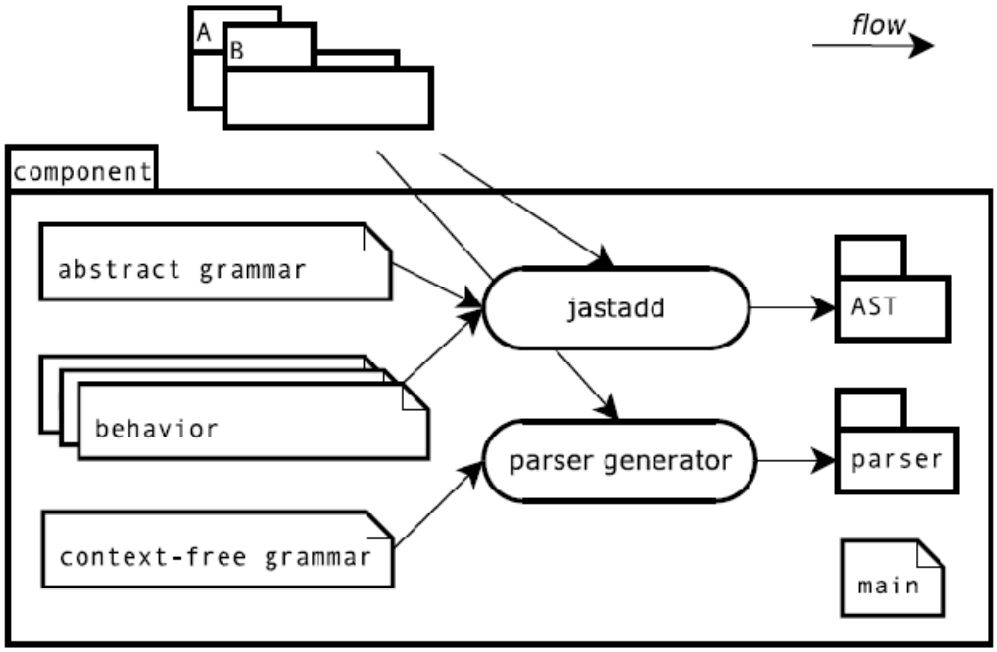
Figur 4.1: Hovedkomponentene i JastAddJ, kopiert fra (8)

4.1.1.2 Produksjonsarkitektur

For å lage et tillegg i kompilatoren må man spesifisere fire ting:

- den abstrakte grammatikken som bestemmer strukturen til det abstrakte syntakstreet (AST).
- adferdsspesifikasjon som definerer hvordan AST'et skal brukes etter at den er ferdig definert.
- en kontekstfri grammatikk som definerer hvordan programtekst skal oversettes til et AST.
- og et hovedprogram som leser program-filen, kjører analysen, bygger AST, og bruker AST adferden til å generere utdata (som først og fremst er klasse-filer med byte-kode).

Den abstrakte grammatikken og adferdspesifikasjonen er spesifisert i et spesielt språk definert for JastAdd. JastAdd-verktøyet tar denne spesifikasjonen og genererer et objekt-orientrert klassehierarki i Java for det abstrakte syntakstreet. For parsingen blir en tradisjonell parser-generator brukt som genererer Java-kode for en parser. En komponent kan bruke andre komponenter på nytt ved å inkludere deres abstrakte grammatikk, adferd og kontekstfrie grammatikk i genereringsprosessen.



Figur 4.2 JastAddJ arkitektur, kopiert fra (8)

4.1.1.3 Modularisering

Den abstrakte og den kontekstfrie grammatikken kan deles opp i flere moduler på en rimelig enkel måte, omtrent som objekt-orienterte klassehierarkier som kan bli utvidet med nye subklasser. Årsaken til at JastAddJ enkelt kan utvides er at adferden kan spesifiseres på en deklarativ måte, med et eget JastAdd-språk.

Utviklerne bak JastAdd beskriver den modulære utvidelsen fra Java 1.4 til Java 5 som en utfordrende oppgave. Dette gjelder spesielt den generiske mekanismen i Java 5, siden den går på tvers av aspektene ved kompileringen med type og navne-analysene på en innviklet måte. På grunn av deklarativiteten til JastAddJ kan disse aspektene skrives som egne moduler, der spesifiseringsrekkefølgen er irrelevant. Grunnen til å gruppere sammen et sett med egenskaper til en modul er å fremme gjenbruk og forståelighet av koden. Man kan dekomponere JastAddJ i to nivåer. På det høye nivået finner vi de såkalte komponentene, som er Java 1.4 frontenden eller Java 5 backenden. Dekomposisjonen er gjort basert på hvilke store komponenter som er forventet å bli gjenbrukt, for eksempel Java 1.4 frontend. På det lave nivået finner vi modulene, som er filer som inneholder for eksempel

grammatikken, navneanalysen eller typeanalysen. En komponent blir delt inn i individuelle moduler, filer, hvor hver modul består av et sett med attributter, likninger, etc. Denne dekomponeringen er hovedsakelig gjort for å øke forståligheten av koden. Figuren under viser dekomposisjonen av komponenter til moduler for Java 1.4 frontenden og Java 5 frontenden. Dette illustrerer hvordan forskjellige typer oppdelings-kriterier kan bli brukt under modulariseringen. Vi kan se at generisk mekanismer er en stor del av Java 5 utvidelsen.

Java 1.4 frontend	*LOC*	*Java 5 extension*	*LOC*
Abstract grammar	261		47
Behavior		Behavior	
Name analysis	2 481	Enhanced for	65
Type analysis	1 387	Autoboxing	197
Definite assignment	1 054	Static imports	110
Exception handling	208	Generics	2 394
Constant expressions	467	Varargs	141
Anonymous classes	124	Enums	339
Class files	475	Annotations	369
Unreachable statements	127		
Prettyprinter	788		
Misc	659		
Bytecode reader	1157		689
Context-free grammar	1053		538
Main program	111		20
total	*10 352*		*4 909*

Figur 4.3: Moduler i Java 1.4 og Java 5 frontenden, kopiert fra (8)

4.1.2 Designprinsipper

Et program er representert i kompilatoren som et abstrakt syntakstre (AST). Tre-nodene er objekter av AST-klassene som er generert fra den abstrakte grammatikken som definerer både et klassehierarki og et komposisjonshierarki. Attributter som er referanser til AST-noder kalles referanseattributter og de løser mange problemer med å uttrykke deklarativitet. Dette står i kontrast til tradisjonelle attributt-grammatikker som ofte krever veldig tungvinte spesifikasjoner for alt som har med bindinger

og deklarasjoner å gjøre. Til tross for at attributtene blir definert deklarativt ved likninger i JastAddJ er de tilgjengelige for vanlige programmerere gjennom metoder i AST-klassene.

4.1.2.1 Grafer lagret i det abstrakte syntakstreet

Trestrukturen til det abstrakte syntakstreet gir en grunnleggende hierarkisk fremstilling av programmet. Man kan bevege seg gjennom treet ved hjelp av metoder generert som en del av AST-klassehierarkiet. For eksempel kan man bruke, `getCondition()` eller `getBody()` for å aksessere de to barna til en while-node. For mange kompileringsproblemer er ekstra grafstrukturer nyttige, for eksempel ved arve-grafer, metodekall-grafer, etc. Ved å bruke referanseattributter får man lagret slike grafer i selve det abstrakte syntakstreet.

Slike grafer kan godt være sykliske. Ta for eksempel de to gjensidig avhengige klassene A og B, hvor A har en variabel av typen B og klassen B har en variabel av typen A. Dette eksempelet gir en syklisk graf. I praksis er mange slike grafer sykliske, så det er veldig nyttig at det underliggende deklarative systemet tillater dem.

Disse grafene kan legges inn i treet ved å definere tilleggsattributter. For eksempel, sett at vi vil definere en *graf* som fanger subtype-relasjonen. Den viktigste informasjonen for å sette opp denne grafen finnes i dekl-attributtene til *extends*- og *implements*-klausulene i klassedeklarasjonene. Denne kan representeres som en del av treet ved å legge et attributt, *supertype*, til KlasseDeklarasjonen.

4.1.2.2 Det abstrakte syntakstreet som den eneste datastrukturen

Ved å bruke et AST med tillagte referanseattributter er det rimelig rett fram å bruke det abstrakte syntakstreet som den eneste datastrukturen. For eksempel, i stedet for å bruke separate symboltabeller slik som tradisjonelle kompilatorer, er den tilhørende informasjonen gjort om til pekere fra navn til deklarasjoner i det abstrakte syntakstreet.

JastAddJ-attributtene kan ha parametre som gjør det mulig å definere passende API'er til det abstrakte syntakstreet. I stedet for å bruke de

tradisjonelle symboltabellene for å finne identifikatordeklarasjoner er mange av AST-nodene utstyrt med en metode *Declaration lookup(String identifier)* som vil returnere den deklarasjonsnoden som er synlig fra den noden man er i. Gjennom treets kanter og referanseattributter kan problemer bli delegert til andre noder og deres attributter. For eksempel kan lookup-metoden for en vanlig identifikatortilgang bli definert etter betingelser fra andre lookup-metoder i omliggende klasser eller dens superklasser.

4.1.2.3 Deklarative utvidelsesmekanismer

JastAdd kombinerer altså vanlige objekt-orienterte utvidelsesmekanismer med noen deklarative mekanismer som er spesielt rettet mot de trebaserte beregningene. Språkstrukturen i programmet som kompileres er spesifisert av en objekt-orientert abstrakt grammatikk, og JastAdd bruker denne til å generere et Java klassehierarki for AST-nodene, som også inkluderer konstruktører og et gjennomløps-API.

Syntetiserte attributter har mye til felles med virtuelle metoder uten bivirkninger. Et attributt i en klasse eller en likning kan bli overstyrt i en subklasse. Arvede attributter sprer informasjon om den gjeldende konteksten nedover i treet. Noden som leser attributtet sin verdi trenger ikke å tenke på hvordan verdien er definert, men bare at det er en foreldrenode som gir en definisjonslikning. Sirkulære attributter kan brukes når det er sirkulære avhengigheter mellom likninger, og de blir da evaluert ved hjelp av iterasjon.

Når man bruker det abstrakte syntakstreet som den eneste datastrukturen er det viktig at man kan spesifisere nye metoder/variabler til nodene i treet, for eksempel under semantisk analyse. Ikke-terminale attributter er attributter som selv er subtrær, og som er satt inn i det eksisterende abstrakte syntakstreet. Disse attributtene kan bruke andre attributter og gir en mulighet til å definere nye trær som funksjoner av et eksisterende tre. JastAddJ støtter også betingede omskrivinger som kan bruke attributter til å definere kontekstavhengige forandringer til det abstrakte syntakstreet. Dette kan for eksempel bli brukt til å skrive om det abstrakte syntakstreet til en ny form som er mer egnet for beregninger senere.

4.1.2.4 Eksempel på JastAdd spesifikasjon

Her skal vi gi en kort introduksjon til JastAdds spesifiseringsspråk ved hjelp av et enkelt eksempel. Anta at programmet inneholder en abstrakt klasse A og de fire klassene B, C, D og E som alle skal brukes til noder i et AST. Klassene D og E er subklasser av A. D-noder skal ha to barn kalt myB og myC, av typene B og C. Noder av typene B, C og E har ingen barn. Dette angis som følger:

```
abstract A;
B;
C;
D: A ::= myB : B
myC : C;
E: A;
```

Anta nå at A skal ha et syntetisert attributt sa av typen int og med en standardverdi 42. D overstyrer standardverdien med en likning som setter sa til 4711. Attributtet og likningen er introdusert inn i klassene A og D ved inter-type deklarasjoner.

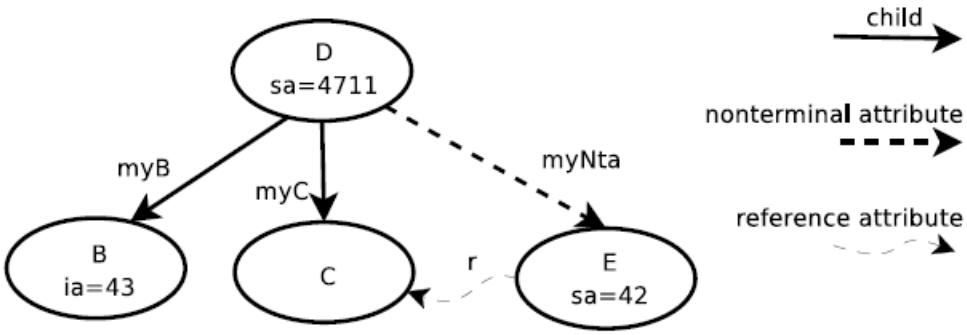
```
syn int A.sa() = 42;
eq D.sa() = 4711;
```

B deklarerer så et arvet attributt, ia, av typen int, og E deklarerer et arvet attributt, r, av typen C. Fordi C er en AST-klasse, blir da r et referanseattributt. D deklarerer et ikketerminal-attributt myNta av typen E, og gir den en standardverdi, new E().

```
inh int B.ia();
inh C E.r();
nta E D,myNta() = new E();
```

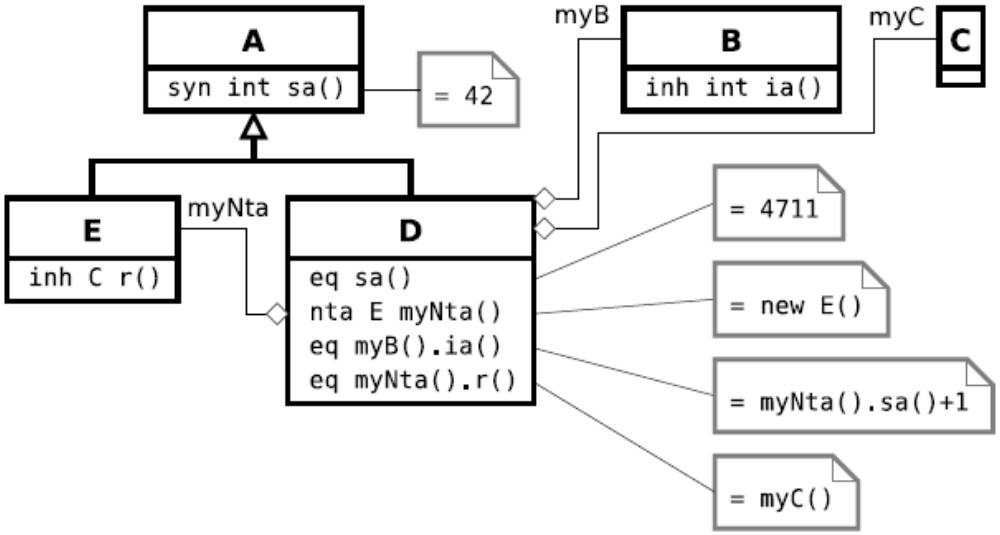
De arvede attributtene må bli definert i en foreldrenode. D definerer *ia* av sitt myB-barn ved å bruke sa-attributtet av sitt myNta ikketerminal-attributt. D definerer også sin referanseattributt r av myNta som likningen til sitt myC barn.

```
eq D.myB().r() = myNta().sa() + 1;
eq D.myNta().r() = myC();
```



Figur 4.4: AST med tilhørende attributter for språkeksempelet, kopiert fra (8)

Figur 4.4 viser det abstrakte syntakstreet med de tilhørende attributtene for språkeksemeplet. Det abstrakte syntakstreet med D-, B- og C- nodene blir laget av parseren, mens E-objektet og r-, ia- og sa-attributtene blir laget automatisk av attributt-evalueringen. Figuren under viser klassediagrammet for språkeksempelet.



Figur 4.5: Klassediagram for språkeksempelet, kopiert fra (8)

4.1.3 Semantisk sjekking i JastAddJ

En viktig del av navnesjekking er å binde hver ”bruksforekomst” av et navn til en deklarasjon ut fra binding og synlighets-reglene i språket. Slike

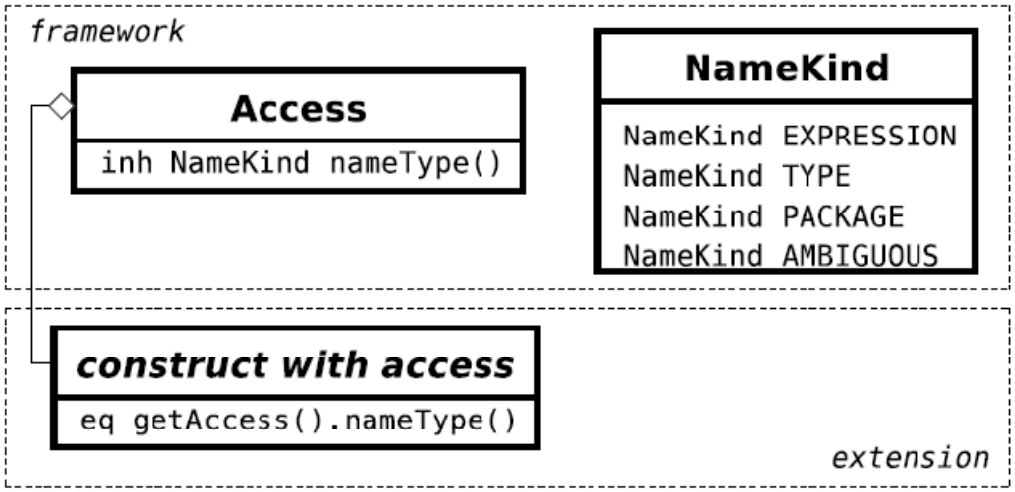
bruksforekomster kalles også aksess-noder senere oppgaven, og er selve bruken av for eksempel variabel-navn innad i programkoden som skal kompileres. Java sin navnesjekking dreier seg om to hovedproblemer. Synlighetsreglene må si alt om forskjellige kombinasjoner av skop, slik som nøstede skop og arv, og om aksess til eksterne medlemmer.

4.1.3.1 Synlighet

Settet med navn som er synlige fra en trenode er i JastAddJ definert av metoden *lookup(String)* som finnes i hver node-klasse, og som returnerer den riktige, synlige deklarasjonsnoden. *Aksess*-noder bruker *lookup*-metoden til å definere sitt dekl-attributt. *Lookup*-metoden er deklarert som et arvet attributt som betyr at dens verdi blir bestemt av verdien i foreldrenoden. Når en aksess-node slår opp med lookup-metoden (og setter svaret ned i en lokal variabel) sier vi da at navnet blir ”bundet”.

4.1.3.2 Fastslå betydningen av navnene

Meningen av en bruksforekomst av et navn i Java er i aller høyeste grad kontekst-sensitiv og en parser vil som regel lage en *Aksess*-node for hver bruksforekomst av et navn. Java Language Specification, JLS, definerer de spesifikke reglene for hvordan man først skal klassifisere kontekstfrie navn etter deres syntaktiske kontekst og så reklassifisere de som er kontekstuelt tvetydige. JastAddJ følger denne implementasjonen. Siden detaljene i den implementasjonen er nokså kompliserte, er det å utvide dette rimelig enkelt: en ny språkkonstruksjon som har et *Aksess*-barn trenger bare å oppgi en likning som definerer dets *nameKind,* som kan være en type, en pakke, et uttrykk, eller noe tvetydig (enten en pakke eller en type, dette kommer an på konteksten). Videre tolking av tvetydige navn blir automatisk utført av Java 1.4 frontenden. Figuren under viser utvidelsesgrensesnittet som bestemmer betydningen av navnene. Rammeverket deklarerer det arvede attributtet *nameType* for *Aksess*-nodene. En ny språkkonstruksjon som har en *Aksess*-node trenger bare å spesifisere en likning for dette attributtet.



Figur 4.6: Utvidelsesgrensesnittet for å bestemme betydningen av navn, kopiert fra (8)

4.1.4 Typesjekking

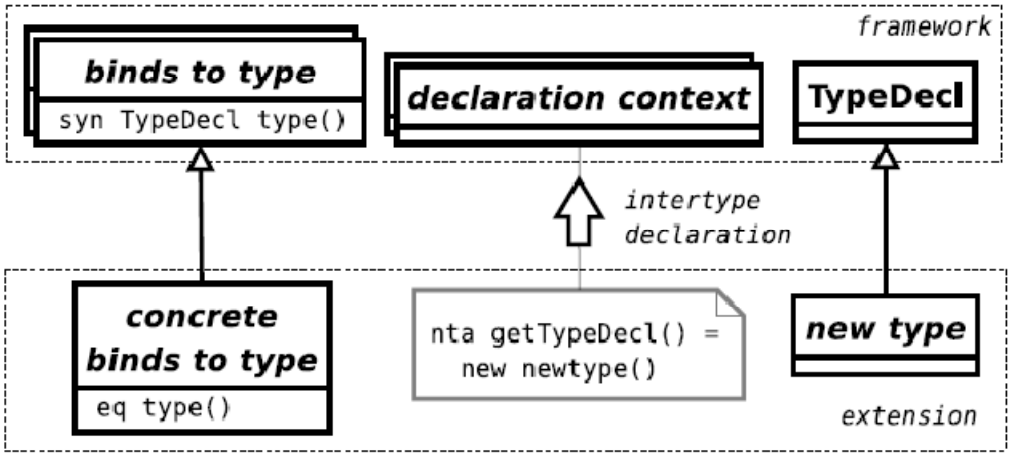
To viktige oppgaver under typesjekkingen er å finne typen til hvert uttrykk og å bestemme om to typer har en subtyperelasjon. Måten dette er implementert på er høyst påvirket av at det abstrakte syntakstreet er den eneste datastrukturen. Dette krever at typene er representert som noder i det abstrakte syntakstreet, og at typen til et uttrykk er representert ved et referanseattributt som peker til den riktige typenoden.

Det å utvide språket med nye typer koker derved ned til de følgende to problemene. Det første er å utforme AST-representasjonen til de nye typene, det andre er å utvide subtypetestene slik at nye instanser av de nye typene kan bli sammenliknet med hverandre og andre eksisterende typer.

4.1.4.1 Typerepresentasjon

Java har både *eksplisitt* og *implisitt* deklarerte typer. For de *eksplisitte* deklarerte typene, som klasser og grensesnitt, bruker vi deres deklarasjonsnode som typerepresentasjon. For *implisitte* deklarerte typer, som primitive typer og arrayer, legger vi til AST-noden som en del av attributtevalueringen, for eksempel etter parsingen. Dette blir i JastAdd gjort deklarativt gjennom bruken av ikke-terminaler. Et ikketerminal-attributt er en barnenode som blir definert av en likning og som ikke blir laget av parseren. I

JastAddJ blir ikke-terminal attributter evaluert på forespørsel, for eksempel, ikke-terminal attributtnoder blir laget automatisk så fort de blir aksessert.



Figur 4.7: Utvidelsesgrensesnittet for typerepresentasjon, kopiert fra (8)

Det finnes i JastAdd abstrakte klasser for uttrykk og deklarasjoner som skal *binde seg til en type*. Hvis utvidelsen introduserer en subklasse til disse, for eksempel den merket ”*concrete binds to type”* i figuren over, må de også ha en likning for bindingen til den ønskede typenoden. Typenoden kan være en instans av den eksisterende *TypeDecl*, eller en ny *type* fra utvidelsen. Det vanlige er at likningen som definerer *type-bindingen* bruker navnesjekkingens dekl-attributt for å finne ønsket type.

Anta at en utvidelse introduserer en ny type. I de tilfellene utvidelsen også introduserer *eksplisitte* deklarasjoner av disse typene kan nodene bli laget automatisk av parseren. Hvis de nye typene er *implisitt* deklarert må de bli laget som ikke-terminal attributter. I det tilfellet må de bli laget som barna til en annen AST-klasse, etter deklarasjonskonteksten. Bruken av inter-type deklarasjoner tillater en utvidelse å legge attributter og likninger til eksisterende AST-klasser i rammeverket.

4.1.5 Bestemte tilordninger

Java Language Specification sier at hver lokal variabel og hvert blanke ”final” felt må ha få tilordnet verdi før verdien dens kan bli aksessert. En Java-kompilator må derfor utføre en konservativ flytanalyse for å være sikker på at hver kontrollflyt-sti til en variabel tilgang har minst en tilordning til den

variabelen. En liknende analyse er nødvendig for å være sikker på at en ”final” variabel ikke er tilordnet mer enn en gang.

4.2 BCEL – The Byte Code Engineering Library

BCELs funksjon er å tilby brukeren en enkel og praktisk måte og analysere, lage, og manipulere Java klassefiler. BCEL kan lese klassefiler og presentere den i en intern objekt-struktur som inneholder all informasjon om klassene, slik som metoder, felter og, kanskje det mest relevante for oss, bytekode-instruksjoner.

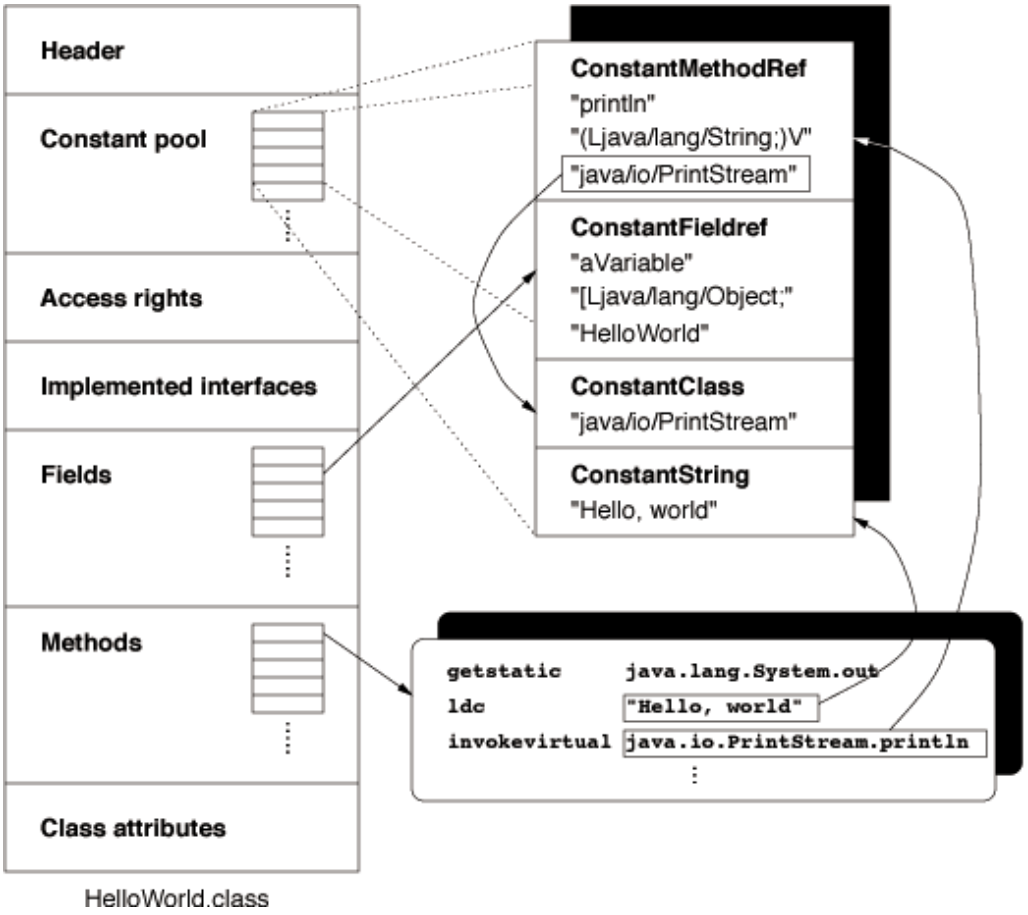
Objekt-strukturen som genereres av BCEL ut fra en klassefil kan greit manipuleres av et program, for deretter å bli skrevet tilbake til fil igjen. I tillegg til dette kan man lage klasser fra scratch ved runtime om dette er nødvendig.

BCEL inneholder også funksjonalitet for å verifisere byte-koden, og denne har navnet JustIce. Denne gir vanligvis noe bedre indikasjon på hva som kan være feil i den genererte koden enn de vanlige meldingene fra JVM.

4.2.1 Java-klassefiler

Figur 4.8 viser et forenklet eksempel over innholdet i en klassefil. Denne starter med et hode som igjen inneholder et ”magisk nummer” (0xCAFEBABE) og versjonsnummeret. Etter dette følger konstant-området, som kan ses på som den tekstlige delen av et objekt. Videre har vi aksessrettighetene til klassen (public, static osv.), eventuelle grensesnitt klassen implementerer, en liste over alle felter og metoder i klassen og til slutt ”attributtene” til klassen, som for eksempel kildefilen til klassen. Man kan også legge til brukerspesifisert innhold i klassefilen. Et eksempel på dette vil være TemplatFile-attributtet som vi legger til alle klasser i en templat. Dette attributtet forteller hvilken templat klassen er en del av.

Ettersom all informasjonen som trengs til beregning av referanser til klasser, felter og metoder er kodet som tekst-konstanter, utgjør konstant-området den største delen av en gjennomsnittlig klassefil. Konstant-området utgjør som regel rundt 60 % av klassefilen (9), mens instruksjonene tar i gjennomsnitt bare 12 % av plassen.



Figur 4.8: Format til Java-klassefil (hentet fra (9))

I figuren over viser boksen oppe til høyre et utsnitt av konstant-området, mens boksen med runde hjørner under denne viser noen instruksjoner fra en metode i klassen. Instruksjonene som er angitt representerer en direkte oversetting av et relativt kjent kodesegment:

```
”System.out.println(”Hello, world”);”
```

4.2.2 Bruk av BCEL i JPT

I vår implementasjon av PT benytter vi altså oss av BCEL for å endre den byte-koden som JastAddJ lager for de klassene som er definert i templater. Vi

bruker den også til å generere nye klasse-filer fra bunn av, og til å laste inn klasser i forbindelse med semantikksjekking under kompilering. Dette skjer etter at JastAddJ har generert den midlertidige byte-koden, men før den forkaster det abstrakte syntakstreet og leter etter nye Java-filer i skopet. I stedet for å la JastAddJ gjøre seg ferdig med kompileringen har vi valgt å hoppe inn i denne mellomfasen ettersom informasjonen vi her har tilgang til vil både være de midlertidige klassefilene og syntakstreet.

Som et eksempel på bruken av BCEL i JPT-kompilatoren har vi valgt å vise hvordan vi legger til attributtet `my$Real$Class` i alle klasser innenfor en templat, (som faktisk skal gjøres under implementasjonen av JPT).

Først laster vi inn en liste av alle klassene i en templat, denne finner vi i syntakstreet ved å utføre:

```
List<ClassDecl> classes = ((PTTemplateDecl)
                            unit.getChild(2).getChild(0)).
                             getClassDeclList();;
```

Figur 4.9: Liste over alle klasser i en templat

Deretter løper vi gjennom klassene og finner navnet på disse ved å kombinere navnet på pakken og navnet på klassen. Dette navnet bruker vi til å la BCEL laste klassefilen til den respektive klassen, etter denne endringen ligger alt til rette for å starte manipuleringen av byte-koden.

BCEL sin representasjon av en klassefil er en meget fleksibel objektstruktur. For å lage et nytt felt av typen *java.lang.object* trenger vi kun å opprette et nytt objekt av klassen `FieldGen` (som representerer et vanlig i felt/variabel i BCEL-strukturen) for deretter å legge dette nye feltet til objekt-strukturen til hver av klassene i templaten.

```
ClassGen manipulateMe = new ClassGen(clazz); //templat klasse

String fieldName = "my$Real$Class";

FieldGen myRealClass =  new FieldGen(Constants.ACC_PUBLIC,
                        Type.CLASS, fieldName,
                        manipulateMe.getConstantPool());
```

Figur 4.10: Kode for å legge til et nytt felt i en klasse

Etter at endringene er utført kan det hele igjen skrives ut som en velformatert klassefil.

4.3 Frontend-delen i vår implementasjon

I dette underkapitelet skal vi se på de endringer vi har gjort i frontenden til JastAddJ for at programmer med generiske pakker og utvidbare klasser skal bli godtatt av kompilatoren. Vi skal begynne med den syntaktiske delen, før vi går over til det abstrakte syntakstreet og semantikksjekkingen. Vi skal også se litt på problemene vi har støtt på i forbindelse med dette.

Vi har valgt å legge alle våre tillegg til kompilatoren i egne filer, men lagt dem på samme sted som tilhørende og lignende filer. Dette har gjort det enkelt og oversiktelig å jobbe, samt at det vil være enkelt å vedlikeholde og oppdatere Package Templates i framtiden.

4.3.1 Fremgangsmåte

Vi skal lage noen enkle tekstfiler for hver templat og pakke. For templatene vil filnavnet være navnet på templaten og ha fil-forslag .pt og for pakkene vil filnavnet være navnet på pakken med fil-forslag .pk. Disse tekstfilene skal inneholde litt informasjon om templatene og klassene. Denne informasjonen vil typisk være en liste over alle klassene for templatene slik at man raskt kan slå opp og se om den klassen man vil utvide eksisterer. For pakkene vil tekstfilen inneholde navnet på alle klassene, hvilke klasser som blir utvidet og hvilke klasser som får nye navn i instansieringen. En .pt-fil blir brukt under semantikksjekken av en instansiering for å finne ut om klassen eksisterer og den gir oss klasse-navnet slik at vi kan laste inn klassen med BCEL. Under ser vi hvordan Graph.pt ser ut for templaten Graph.

```
name::Graph
classes:::
  class::templatetest/Node
  class::templatetest/Edge
```

Figur 4.11: Innholdet til Graph.pt

I figuren under ser vi hvordan filen VeiOgByGraf.pk ser ut for instansieringen av templaten Graph i pakken VeiOgByGraf. Vi ser at det opprinnelige klasse-navnet kommer først, etter srcClass::, og så kommer det nye klasse-navnet,

etter destClass::. Under en klasse kommer alle metodene som er i klassen, og som skal være med i instansieringen. De metodene som skal få nye navn vil komme først i lista og så kommer alle de resterende metodene med samme navn i srcMethod og destMethod, siden de kun er lagt til uten nye navn for at JastAddJ skal finne dem under sin kodegenerering.

```
name::VeiOgByGraf
classes:::
  srcClass::templatetest/Node
  destClass::packagetest/By
      srcMethod::insertEdgeTo

destMethod::settInnKantTil
      srcMethod::display
      destMethod::display
      srcMethod::test
      destMethod::test
      …
      …
  srcClass::templatetest/Edge
  destClass::packagetest/Vei
      srcMethod::deleteMe
      destMethod::deleteMe
      srcMethod::display
      destMethod::display
```

Figur 4.12: Et utdrag fra VeiOgByGraf.pk

4.3.2 Skanneren og parseren

Vi må selvsagt legge til noen nye leksemer (tokens) i skanneren for at JPT skal kunne lese PT-programmer. Vi skal ikke gå inn i detaljene ved dette på annen måte enn å vise et utdrag fra filen de er beskrevet i, patec.flex, i Figur 4.13.

```
<YYINITIAL> {
  "template"      { return sym(Terminals.TEMPLATE); }
  "namespace"     { return sym(Terminals.NAMESPACE); }
  "inst"          { return sym(Terminals.INST); }
  "with"          { return sym(Terminals.WITH); }
  "=>"            { return sym(Terminals.DOUBLEARROW); }
  "adds"          { return sym(Terminals.ADDS); }
  "->"            { return sym(Terminals.SIMPLEARROW); }
}
```

Figur 4.13: Språkbegrepene til PT beskrevet i patec.flex

Også på det syntaktiske nivå må det legges til flere regler for å dekke hele PT.

Denne er også nokså rett fram og sette opp i JastAdd sitt spesielle språk, og hvordan dette typisk tar seg ut er vist i Figur 4.14

```
CompilationUnit compilation_unit =
      namespace_declaration.n import_declarations.i?
      patec_declarations.pt?{:PTCompilationUnit ptc = new
      PTCompilationUnit(n.getID(),new List(),i,pt);
      ptc.setChild(ptc.convertList(), 0); return ptc; :}
  ;

List patec_declarations = patec_declaration.pt
      {: return !(pt instanceof EmptyPTDecl) ? new List().add(pt)
      : new List() ; :}
  ;

PTDecl patec_declaration =
      TEMPLATE IDENTIFIER LBRACE class_declarations.c? RBRACE
      : return new PTTemplateDecl(IDENTIFIER, c); :}
      | PACKAGE IDENTIFIER LBRACE ptinst_declarations.i?
      ptclass_declarations.c? RBRACE
      {: return new PTPackageDecl(IDENTIFIER,i,c); :}
  ;

IdUse namespace_declaration =
  NAMESPACE name_decl.n SEMICOLON {: return n; :}
  ;
```

Figur 4.14: Et utdrag fra den syntaktiske beskrivelsen til PT

4.3.3 Det abstrakte syntakstreet

Siden PT er en utvidelse av Java vil vi trenge noen nye nodetyper for å kunne representere PT-programmer som syntakstrær. Det var en utfordring å finne ut hvordan vi skulle få de nye node-objektene inn i det eksisterende syntakstreet rent JastAddJ-teknisk, men når vi først fant ut av dette gikk det egentlig ganske greit å få det til. Det vi fant var at vi måtte utvide rot-noden for det eksisterende treet til å godta den syntaktiske beskrivelsen til PT og subtreet vi lager for PT. Første linje i Figur 4.14 uttrykker at vi til rot-noden, CompilationUnit, legger til vår syntaktiske beskrivelse av PT. Dette nye treet vil da få en PTCompilationUnit som rot-node og vil ha videre støtte for PT-begreper. I Figur 4.15 ser vi i første linje at PTCompilationUnit er en subnode-klasse av JastAddJs CompilationUnit.

De nye node-klassene våre er:

- *PTCompilationUnit*. Denne noden utvider JastAddJs *CompilationUnit* og har en liste med *importDecl* og *PTDecl* som parametere til konstruktøren.
- *PTDecl*, som utvider *ASTNode og er en felles superklasse for PTTemplateDecl og PTPackageDecl.*
- *PTTemplateDecl*, utvider *PTDecl*. Denne inneholder navnet på templaten og en liste med *PTClassDecl* som parametere til konstruktøren.
- *PTPackageDecl*, utvider *PTDecl*. Denne inneholder navnet på den generiske pakken, en liste med *PTInstDecl* og en liste med *PTClassDecl* som parametere til konstruktøren.
- *PTClassDecl*, utvider *ClassDecl*.
- *PTInstDecl*, utvider *TypeDecl* og tar en liste med *PTMergeDecl* som parameter til konstruktøren.
- *PTMergeDecl*, utvider *AstNode*. Denne sender det opprinnelige klassenavnet, det nye klassenavnet og en evt. liste med metoder som skal få nye navn til konstruktøren.
- *PTMethodMergeDecl*, utvider *ASTNode*. Denne sender det opprinnelige metodenavnet og det nye metodenavnet til konstuktøren.

I tillegg til de over lagde vi også node-klassene under som blir brukt hvis noen av listene som nodene har er tomme. Dette gjør det enklere å sjekke om lista er tom.

- *EmptyInst*, som utvider PTInstDecl.
- *EmptyPTClass*, som utvider PTClassDecl.
- *EmptyPTDecl*, som utvider PTDecl.
- *EmptyClass*, som utvider ClassDecl.
- *EmptyPTMerge*, som utvider PTMergeDecl.
- *EmptyPTMethodMerge,* som utvider PTMethodMergeDecl.

Slik som med alle våre andre tillegg til kompilatoren befinner også dette seg i en egen fil, patec.ast.

```
PTCompilationUnit : CompilationUnit ::= ImportDecl* PTDecl*;
PTDecl : ASTNode              ::= ;
PTTemplateDecl : PTDecl       ::=   <ID:String> PTClassDecl*;
PTPackageDecl : PTDecl        ::=   <ID:String> PTInstDecl*
                                    PTClassDecl*;
PTClassDecl : ClassDecl       ::= ;
PTInstDecl : TypeDecl         ::=   PTMergeDecl*;
PTMergeDecl : ASTNode         ::=   <SrcClass:String>
                                    <DestClass:String>
                                    PTMethodMergeDecl*;
PTMethodMergeDecl : ASTNode ::=     <SrcMethod:String>
<DestMethod:String>

EmptyInst : PTInstDecl        ::= ;
EmptyPTClass : PTClassDecl    ::= ;
EmptyPTDecl : PTDecl          ::= ;
EmptyClass : ClassDecl        ::= ;
```

Figur 4.15: Våre tillegg til det abstrakte syntakstreet i filen patec.ast

4.3.4 Semantisk analyse

Den semantiske analysen av en templat kan gjøres omtrent som i Java, siden de semantiske regler er svært like de for vanlige pakker. Dog må det gjøres noe spesielt omkring nøkkelordene *namespace* og *template*.

Den semantiske analysen av en instansiering er det derimot en del mer jobb med. Denne baserer seg på at templaten allerede er lagt inn i det abstrakte syntakstreet, og er laget klassefiler av.

Ved den semantiske sjekken starter vi å sjekke om templaten som skal instansieres er kompilert og om det har blitt laget en .pt-fil. Hvis det ikke finnes en .pt-fil for templaten avsluttes kompileringen. Finner man .pt-filen for templaten blir klassenavnene lest inn og lagt i en liste. Deretter blir navnene til klassene som skal instansieres i pakken hentet ut fra det abstrakte syntakstreet, og det sjekkes at de klassenavnene vi har fra treet ikke finnes blant klassenavnene fra templaten. Hvis en klasse skal få et nytt navn må det også sjekkes om det er et lovlig navn. Det eneste vi anser som ulovlig navn er hvis det nye klassenavnet allerede er et eksisterende klassenavn. Navnene må selvsagt følge Java-standarden.

Det er også lov til å endre metodenavnene, på samme måte som klassenavnene. Vi sjekker da om navneendringene er lovlige ved først å hente

ut listen av metodene fra treet. Deretter leser vi, med BCEL, inn klassefilen til klassen definert i templaten, der metoden ligger. Metodenavnene fra treet blir sjekket opp mot de som ligger i klassefilen og hvis en metode ikke ligger i klassefilen er det feil. Det må sjekkes at det nye metodenavnet ikke allerede er brukt i klassen.

```
inst Graf with Node => By (settInnKantTil -> settInnVeiTil)
```

For å oppsummere dette kort går vi igjennom inst-setningen i eksempelet over. Først sjekker vi om Graf.pt eksisterer, gjør den det sjekker vi om den inneholder klassen `Node`. Så sjekker vi om det nye navnet `By` er en lovlig navnendring. Deretter om klassen `Node` inneholder metoden `settInnKantTil` og til slutt sjekker vi om det nye metodenavnet, `settInnVeiTil`, ikke er brukt før i klassen. Hvis alt dette stemmer er dette så langt en lovlig instansiering av templaten `Graf`.

Tillegget til en klasse, altså adds-delen, har samme struktur som en normal klasse. Det eneste som skiller den fra å være en helt ordinær klasse er at man kan ta i bruk metoder som er definert i templaten, uten å definere dem først. Dette gjør JastAddJ sin semantikksjekk fungerer som vanlig, unntatt for disse metodene som er definert i templaten. Disse må vi sjekke om finnes på samme måten som for inst-setningen. Dette gjelder om både metoder som har fått nytt navn og metoder som ikke har fått nytt navn. Måten vi sjekker dette på er ved å bruke inst-setningen for vi vet at hvis kompilatoren har kommet til tilleggs-klassene, så har den allerede sjekket inst-setningen. Vi må også gjøre dette for klasser det blir laget nye objekter av. Framgangsmåten for dette er den samme som for metodene.

4.3.5 Selvforskyldte problemer

På JastAdds nettsider finnes det et eksempel som viser hvordan man kan utvide JastAddJ til å kunne sjekke under kompileringen visse ting omkring mulige nullpeker-overtredelser. Dette eksempelet viser også hvordan vi skal få lagt til våre utvidelser så vi lastet det ned og begynte å endre det til å implementere Package Templates. Dette skulle vise seg på mange måter å være en stor feil, siden backenden i denne versjonen var mye mer endret enn vi trodde. Det som skjedde var at frontenden vi hadde laget virket så lenge vi

kjørte bare den, men når vi skulle kompilere alt sammen for å få ut kompilatoren vår krasjet den under kompilering hver gang den skulle gjøre noe med typene til en deklarasjon. Etter mye frustrasjon og feilsøking endte det med at vi lastet ned JastAddJ, og ikke et eksempel, og la inn kun de nye filene vi hadde laget. Dermed virket alt som det skulle. Det vi lærte fra den feilen var hvordan vi skulle lage de forskjellige modulene slik at vi slapp å endre de allerede eksisterende filene i JastAddJ.

4.4 Backend-delen i vår implementasjon

Vi skal starte med å forklare litt om hvordan vi har tenkt å gå fram for å løse problemene rundt kodegenereringen og implementasjonen av backenden. Deretter skal vi se på hvordan organiseringen ved runtime skal foregå. Til slutt skal vi se på hvordan vi må justere den byte-koden som JastAddJ har produsert så den stemmer med definisjonen av PT og de runtime-strukturene vi skal bruke.

4.4.1 Fremgangsmåte

Som vi nevnte i innledningen til kapittel 4 er planen å la JastAddJ lage byte-kode for PT-programmer, som om det var helt vanlig Java. Vi må da få den til å tro at templatene er vanlige pakker, og vi må legge inn noen ekstra mekanismer for å få den til å behandle inst-setninger og tilleggs-(adds-) klasser riktig.

Vi vil så bruke BCEL til å lese inn disse klassefilene, og til å forandre byte-koden så den gjør det den skal. Vi vil da vite hvilke instruksjoner som må byttes ut, og med hva. De nye instruksjonene må sørge for at pekere blir riktige og at riktige klasser, metoder og variabler blir eksekvert.

Vi finner de stedene vi skal forandre ved å gå sekvensielt gjennom den objekt-representasjonen som BCEL lager ut fra klassefilen. Dette kaller vi et *justerings-gjennomløp*, og dette må gjøres for alle klasse-filer i programmet, ikke bare de som stammer fra templaten. Før vi går inn og endrer instruksjonene må imidlertid noen andre tilpasninger gjøres. Det første er å gå gjennom alle metodene og legge til en ekstra parameter. Denne parameteren vil være en peker til et såkalt TID-objekt for den aktuelle instansieringen. Dette objektet vil angi hvilken instans den er i og dermed sørge for at typene

blir typet riktig. En nærmere beskrivelse av hvordan vi bruker dette objektet vil komme strakts.

Det andre er å gi alle klassene definert i templater et nytt ekstra peker-attributt av type Object. Dette har vi har valgt å kalle `my$Real$Class`. Hva dette skal brukes til blir forklart nedenfor.

4.4.2 Datastrukturene i runtime-systemet

I innledningen av kapittel 4 satte vi opp en del begrensninger på PT-språket som vi ville anta under vår implementasjon av PT. Vi skal her fremstille runtime-systemet slik vi utformer det med disse begrensningene. Av begrensningene er de følgende to av størst betydning for organiseringen av runtime-systemet: ”Bare pakker kan instansiere templater” og ”templat-klasser kan ikke ha super-klasser”. Dette medfører at vi kun får to nivåer i forbindelse med templat-klasser.

1. Klassene som er definert i templater (som altså ikke har superklasser eller er adds-klasser for klasser av andre templater)
2. Adds-klasser, defineres i forbindelse med instansieringer, forekommer bare i tradisjonelle Java-pakker.

Dessuten har vi altså vanlige Java-klasser som er definert (og dermed bare brukt) på pakke-nivået, disse blir implementert på helt rett fram måte. Som eksempel for de videre framstillinger, ser vi på programmet i Figur 4.16**.**

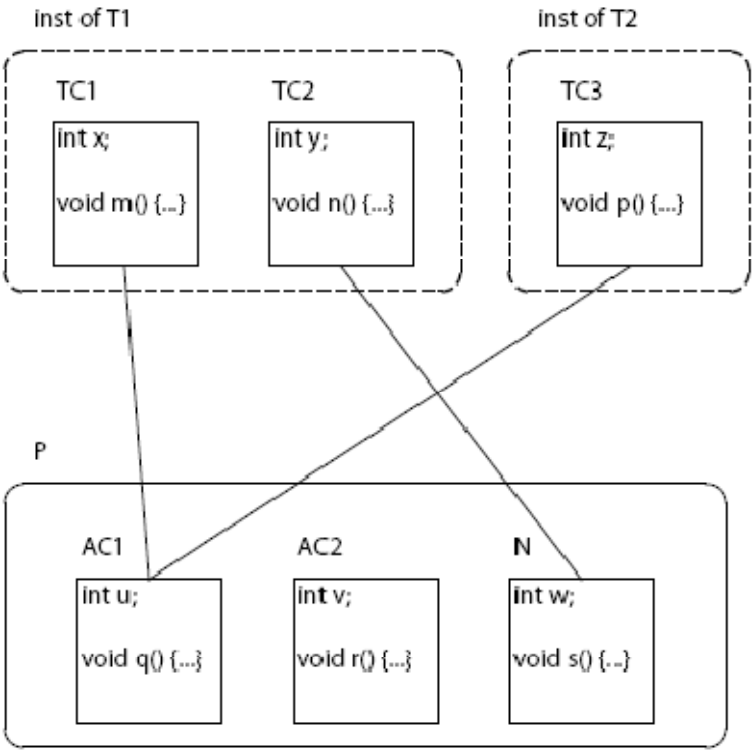
```
template T1 {

      class TC1 {
          …; int t; void m(){…}
      }
      class TC2 {
          …; int t; void n(){…}
      }

}

template T2 {

      class TC3 {
          …; int t; void p(){…}
      }

}
```

Figur 4.16: PT-program

Disse instansieres som følger:

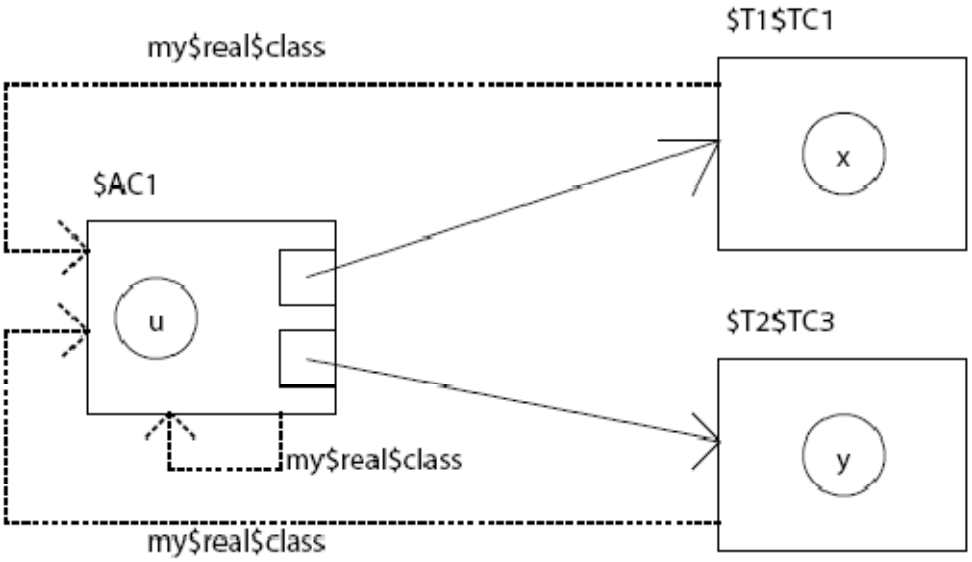
```
package P {

      inst T1 with TC1 => AC1, TC2 => AC2;
      inst T2 with TC3 => AC1;

      class AC1 adds {
            int u; void q(){…}
      }
      class AC2 adds {
            int v; void r(){…}
      }
      class N {
            int w; void s(){…}
      }

}
```

Figur 4.17: Instansiering av PT-program



Figur 4.18: Datastrukturen til instansieringen

Som vi ser i Figur 4.18 har vi to templat-instansieringer av henholdsvis `T1` og `T2`. T1-instansieringen inneholder de to klassene `TC1` og `TC2` som har variablene `x` og `y` og metodene `m()` og `n()`. T2 inneholder kun én klasse TC3, med variabelen `z` og metoden `p()`. Disse to templatene blir instansiert inne i pakken P. Her utvider klassen `AC1` klassene `TC1` og TC3, og klassen `AC2` utvider `TC2`. I tillegg til disse klassene i P har P også en vanlig Java-klasse N. I Figur 4.19 ser vi organiseringen til et AC1-objekt ved runtime.



Figur 4.19: AC1-objektet ved runtime

Et objekt av klassen `AC1` vil ved runtime se ut som angitt i figuren over. Det består altså av tre *del-objekter* som hver har sin klasse definert i byte-kode. Ett av disse (`AC1`) er *hoveddelen,* mens alle de andre stammer fra templater. Hvert del-objekt har en peker `my$Real$Class` (nevnt tidligere) som peker til hoveddelen (pekeren til hoveddelen er en peker til seg selv). Hoveddelen har pekere til de andre delene, med navn som her er `$T1$TC1` og `$T2$TC3`. Typen på disse er kjent av kompilatoren, mens typen på `my$Real$Class` er `Object.`

Når en metode i for eksempel `TC1` eksekverer, så må this-pekeren (levert med av Java som en ekstra parameter) peke til det tilsvarende del-objektet. Dermed må denne pekeren ”flyttes” etter hvor de kalte metodene er definert.

Når man kompilerer pakken `P` vil man ha full oversikt over alle klassene og hvordan de er satt sammen av objekter. Dermed kan man bare ”*dotte seg fram*” til riktig variabel eller metode. I en templat derimot har man ingen kunnskap om hvilken klasse eller pakke den enkelte klassen i templaten er en del av, men dette vil bli kompensert for med metoder i TD- og TID-klassene som vi straks skal forklare.

4.4.2.1 TD- og TID-klassene

For å holde sammen runtime-strukturen må vi for hver templat-instansiering ha et objekt som tilsvarer PIDen i Sørensens metode (kapittel 3.3.2). Klassen for dette objektet er bygget opp som følger: Vi definerer først en såkalt TD-klasse (TD = Template Descriptor) for hver templat. Så, for hver instansiering av denne templaten definerer vi en TID-klasse (TID = Template Instance Descriptor) som en subklasser av TD-klassen. Det blir så, i forbindelse med instansieringen av et program, laget et TID-objekt for alle templat-instansieringene i programmet. Hvor disse ligger vet kompilatoren.

Merk at vi i tillegg til TD- og TID-klassene også snakket om templat-klasser. Disse siste er rett og slett klasser som er definert inne i en templat, og som derved er skrevet av en JPT-programmerer. TD- og TID-klassene blir generert fullstendig av vår kompilator.

Merk også at vi i dette kapittelet forutsetter at alle metodene definert i templater hat fått en ekstra parameter som er typet med templatens TD-klasse.

Vi må i TD- og TID-klassene sette inn noen metoder som skal være ”koblingsbokser” mellom kall gjort fra template-nivå til metoder definert på pakke-nivå, og omvendt. Disse må delvis defineres i TD-klassen og delvis i TID-klassen. Husk at i TD-klassen kan vi ikke snakke om hvilke klasser de forskjellige template-klasser er blitt utvidet til under en instansiering, men det kan vi i TID-klassen, siden det lages én TID-(sub)klasse for hver instansiering av pakken.

Vi må lage slike ”kobling-metoder” for alle metoder i klasser i templaten, men det må gjøres litt forskjellige for statiske, for virtuelle objekt-metoder og for såkalte ”special”-metoder (ikke-virtuelle objekt-metoder, for eksempel konstruktører). La oss anta at vi er i en pakke med navn T, at den metoden `m` vi ser på er definert i klasse K, og leverer resultat av type U, og at den har parametere `m(P p, R r)`, der `p` representerer alle de parametrene som er typet med klasser i T, mens `r` representerer resten. P-parameterne peker til objekter med flere deler, og må derfor justeres med det som er kalt ”ut”- og ”inn”-pekere lenger ned. I den instansieringen vi ser på får vi følgende navn på adds-klassene: P blir til Q, U blir til V og K blir til L.

Vi kaller TD- og TID-klassene henholdsvis `$TD$T` og `$TID1$T` (der vi tenker oss at navnet for andre instansieringer av T er `$TID2$T`, `$TID3$T` osv.).

Om m er en statisk metode trenger vi bare å tenke på kall fra pakke-nivået til template-nivået, og vi legger følgende metode inn i `$TID1$T`:

```
V $T$K$m(Q q, R r){return (V)(K.m(this $TID$T, q.inn,
```

Dette må justeres på opplagt måte når metoden enten er av type `void`, eller er av en type som ikke er en klasse i T. Her står navnet ”inn” for de pekerne som går fra pakkenivå-delen av adds-objektene til del-objektene som tilsvarer templat-nivået. Navnet på disse er kjent av kompilatoren. Navnet ”ut” er pekeren som går den andre veien, altså ”`my$Real$Class`”. Merk at denne er typet med `Object`, slik at den alltid må type-kastes med V før bruk.

Metoden over kalles når det fra pakke-nivå gjøres et kall på den statiske metoden `m` i klassen K i T slik: `m(q, r).` Kompilatoren vet da hvilken instansiering av T det dreier seg om, og kan kalle metoden i riktig TID-objekt.

For å kunne ta oss av såkalte ”special” (virtuell) metoder som kalles fra pakke-nivå i objektet `l` (type L, adds-klassen til K) legger vi følgende inn i `$TID1$T`:

```
V $T$K$m(L l, Q q, R r)
    {return (V)(l.inn.m(this $TID$T, q.inn, r).ut);}
```

Den virker og skal kalles på samme måte som den over (bare at den har med en objekt-peker), og de tilsvarende justeringer må eventuelt gjøres angående resultattypen.

Om `m` er en virtuell metode definert i T-klassen K, så må vi tenke på at den kan være redefinert på pakke-nivå, og vi må tenke på kall både fra templat- og pakke-nivået. Det siste skal vise se at til slutt går uten forandringer.

Med tanke på kall fra template-nivået, legger vi først følgende abstrakte metode inn i klassen `$TD$T`:

```
abstract U $T$K$m($TD$T t, K k, P p, R
```

Denne skal kalles når en metode `m` i objektet `k` (av klassen K i T) kalles innefra T slik: `k.m(p, r).` I subklassen `$TID1$T` legger vi så inn en

metode som ”redefinerer” denne, og som gjør det tilsvarende kall på pakke-nivå:

```
U $T$K$m($TD$T t, K k, P p, R r)
    {return ((L)k.ut).m((Q)p.ut, r).inn);}
```

Grunnen til dette siste er at m kan være redefinert på pakke-nivå (gjerne i en subklasse av adds-klassen), og om det er tilfelle her vil kallet gjort i metoden over lede oss til den rette versjon. Vi må imidlertid også tenke på at metoden kanskje *ikke* er redefinert på pakke-nivå, og da er det versjonen i templaten som skal kalles. Måten vi ordner dette på er at vi for hver virtuelle metode `m` i templaten går inn i den tilsvarende adds-klassen `L` og ser om `m` er definert der. Om den *ikke* er det legger vi rett og slett inn følgende versjon, som kaller versjonen i K som invokespecial.

```
V m(Q q, R r){return $T$m(t, this L, q, r); }
```

Kompilatoren vil her vite hvilket TID-objekt `t` som angir riktig instansiering. Merk at Javas vanlige virtuell-mekanismse på pakke-nivå (Klassen `L` og dens subklasser) nå vil sørge for at riktig versjon blir valgt.

For å kunne utføre new-operasjoner for templat-klasser trenger vi også noen koblingsbokser. Med tanke på `new K()` gjort inne i templaten `T`, og der `K` er en klasse i `T`, legger vi inn følgende metode inne i TD-klassen for `T`:

```
abstract K $new$K$T(...)
```

I TID-klassen for en instansiering gjør vi så følgende redefinisjon av denne.

```
K $new$K$T() { new L().inn}
```

Her er inn-pekeren riktig typet så vi ikke trenger noe type-kast. Dette forutsetter at konstruktørene i klassen `L` er utvidet, slik at den også lager del-objektene på templat-nivå og kaller deres konstruktører på riktig måte (ved invokespecial).

4.4.3 Justerings gjennomløpet

I dette del-kapittelet skal vi beskrive hvilke ting vi må lete etter når vi går

igjennom og vil forandre ”rå-koden” JastAddJ har produsert for templatene og pakkene i programmet.

Mange bytekode-instruksjoner vil logisk sett referere til navn, men dette blir gjort ved at instruksjonen bare angir en indeks til klassefilens konstant-område, der det virkelige navnet ligger. I det følgende skal vi delvis abstrahere bort dette indirekte trinnet.

Siden vi ofte vil sette inn nye instruksjoner i byte-koden, må vi før endringene legges tilbake i klassefilen, rydde opp i indekseringen av instruksjonslisten. Dette har BCEL støtte for ved et kall til funksjonen `setPositions()`. Denne funksjonen vil for eksempel sørge for at hopp ved `GOTO` blir utført på riktig måte.

4.4.3.1 getfield og putfield

Instruksjonen `getfield` blir generert av kompilatoren når et felt i et objekt skal aksesseres. Verdien av feltet blir kopiert til toppen av stakken. Navnet til klassen og feltet er en konstant i konstant-området. Instruksjonen `putfield` popper toppen av stakken og setter verdien ned i feltet.

Ettersom vi som nevnt under runtime-organiseringen, alltid sørger for at this-pekeren peker på riktig del-objekt vil aksess av felter ved `getfield` og `putfield` gå som normalt så lenge man aksesserer felt definert på eget nivå. Når man fra ytterste pakke-nivå aksesserer variable i en templat-klasse må man imidlertid legge til en ekstra ”dot-aksess” med `my$Real$Class` for å komme i riktig del-objekt, og om typen er en klasse i templaten må man legge inn et type-kast for at byte-koden skal bli konsistent. Kompilatoren vet om de typer og navn som da må brukes. Felt-aksess fra templat-nivået til ytterste nivå forekommer ikke.

4.4.3.2 getstatic og putstatic

Instruksjonen `getstatic` blir generert av kompilatoren når et statisk felt i en klasse skal leses til toppen av stakken. Navnet til klassen og feltet er gitt i instruksjonen. `Putstatic` popper toppen av stakken og gir verdien til feltet.

For disse instruksjonene kreves det at vi gjør noen endringer i byte-koden, dersom det gjelder statiske variable til templat-klasser. Vi må da finne det rette TID-objektet, og ”dotte” oss inn i dette. Om man er på ytterste nivå vil kompilatoren (ut fra .pk-filen) kjenne en peker til dette objektet mens om man er i en templat-klasse vil denne pekeren være en ekstra parameter til den metoden man er i.

For `getstatic` er det derfor ikke store endringer som må til. Ved å laste templatobjektet på stakken får vi tilgang til de statiske feltene som fantes i templaten og ettersom disse nå er lagret som vanlige ikke-statiske variable kan vi utføre en getfield for å hente innholdet. Om variablen er typet med en templat-klasse må man også dotte seg langs pekeren `my$Real$Class` og legge på et type-kast til riktig objekt. Byte-koden før manipulasjon ser vi i Figur 4.20.

```
0: getstatic #27; //Field t:I
3: istore_2
```

Figur 4.20: getstatic før manipulering

Etter at byte-koden er manipulert ser den slik ut:

```
0: aload_1
1: getfield #85; //Field Graph.$Node$t:I
4: istore_2
```

Figur 4.21: getstatic etter manipulering

4.4.3.3 Metode-definisjoner

Det er lite som behøver å gjøres med parametrene etc. i de forskjellige metodedefinisjonene. I justerings-gjennomløpet for pakke-nivået er det faktisk ikke noe som må gjøres med dette, mens man på templat-nivå må la alle metoder (også statiske og konstruktører) få en ekstra parameter som er typet med TD-klassen for templaten. Det er viktig å huske at konstruktørene for adds-klasser må forandres slik at del-objekter fra templat-nivået også blir generert, og at konstruktørene også kaller disse.

4.4.3.4 invokespecial

Denne instruksjonen blir generert når en ikke-virtuell ikke-statisk metode blir kalt for et objekt. Den blir blant annet brukt for å kalle konstruktører.

Metoder definert inne i templater vil i justerings-gjennomløpet for templat-filer bli forandret slik at de forventer én parameter mer enn de er deklarert med i programteksten (i tillegg til den implisitte this-parametren) og i denne parameteren forventer en templatmetode en peker til TID-objektet for den aktuelle templat-instansieringen.

Dersom metoden kalles fra ytterste nivå er en peker til dette TID-objektet kjent av kompilatoren, og vi forandrer kallet slik som foreskrevet i kapittelet om TID-objektene og koblings-boksene. Om kallet kommer innenfra templaten vil den kallede metode selv ha fått det riktige TID-objekt som parameter. Det kan derfor greit leveres som ekstra parameter i det aktuelle kallet. Men for øvrig kan dette kallet gjøres uten å gå innom noen koblingsboks eller liknende.

4.4.3.5 invokevirtual

Denne instruksjonen blir generert av kompilatoren når en virtuell metode blir kalt. Navnet til klassen, metoden og typesignaturen er del instruksjonen.

Som vi så i kapittelet om TD- og TID-objektet er det i justerings-gjennomløpet for pakkenivået ikke nødvendig å forandre noe. Er det redefinisjoner på pakkenivået vil den riktige versjonen ut fra Javas virtuell-mekanisme bli kalt, og om det ikke er redefinisjoner vil den metoden vi satte inn lede oss til den versjonen som defineres i templaten.

For et kall gjort fra templat-nivå må vi følge anvisningen i TD- og TID-kapittelet og gjøre det om til et kall på en abstrakt metode i TD-klassen som så gjennom en definisjon i TID-klassen blir ledet til et vanlig kall på ytterste nivå.

4.4.3.6 invokestatic

Denne instruksjonen blir brukt av kompilatoren når en statisk metode blir kalt. Navnet til klassen, metoden og typesignaturen er en del av instruksjonen.

Som vi så i TD- og TID-kapittelet må man i justerings-gjennomløpet skifte ut dette kallet med et til den rette koblingsboks i TID-objektet for den aktuelle instansieringen. Om kallet gjøres innenfra en templat må vi bare passe på å gi med den ekstra TD-parametren.

4.4.3.7 new

Denne instruksjonen blir generert av kompilatoren når det skal opprettes et nytt objekt. Navnet på klassen følger med i instruksjonen.

Når et slikt kall gjøres fra pakke-nivå behøver man ikke forandre noen ting. Dette forusetter at konstruktørene for adds-objektene er blitt utvider slik at del-objektene tilsvarende templat-klassene også blir generert, og at konstruktørenene for disse kalles.

Når en templat selv sier nev på en av sine klasser må metoden `$new$K$T()` kalles slik som beskrevet i TD- og TID-kapittelet.

4.4.3.8 checkcast og instanceof

Disse instruksjonene blir generert når det skal utføres henholdsvis et type-kast eller en sjekk av klassetilhørighet. Klassen angis i instruksjonen og peker til objektet ligger på toppen av stakken.

Om en av disse instruksjonene opptrer på pakke-nivå vil de virke helt greit som de er. Dette kommer av at det aldri blir laget PT-objekter som bare består av del-objekter av templat-klasser.

Dersom en slik instruksjon kalles innenfra en templat, skal den med våre restriksjoner (og om den er gått gjennom JastAdd kompilatoren) alltid gå gjennom (bortsett fra at instanceof vil tilsvare en none-test). Byte-koden må derfor forandres ut fra det.

Om templat-klasser kunne hatt superklasser måtte vi laget et tilsvarende system som for new når den kalles innenfra en templat. Dette ville vært nokså rett fram.

4.4.4 Oppsummering kapittel 4.4

Som nevnt tidligere i dette kapittelet har vi sett oss nødt til å generere byte-kode i to omganger. Dette gjorde vi for å få nok tid til å komme noenlunde gjennom hele kompilerings-prosessen. Et alternativ hadde vært en løsning der vi manipulerer den midlertidige byte-koden JastAddJ genererer.

Vi summerer her opp det som må gjøres i forbindelse med justerings-gjennomløpet.

Av en templat:

- .pt-fil blir laget
- Ny parameter på alle metoder
- nytt attributt (`my$Real$Class` av typen object)
- diverse byte-kode endringer som vist over

Av en pakke:

- Lage .pk-filer
- Ordne med virtuelle metoder
- Spesialbehandlig av metoder i adds-delen.

For hver (virtuell) metode i templat-klassen sjekker justerings-gjennomløpet om den samme metoden (med nye klasse-navn på typene) er definert i tilleggs-klassen. Om dette er tilfellet, gjøres ingenting, men om den ikke er definert legges en slik metode inn, med kode til å kalle den tilsvarende metoden i den riktige templat-klassen. I dette kallet må de samme justeringene gjøres som ved `invokespecial`, og en ekstra parameter med inst-objektet til templaten.

Kapittel 5

Oppsummering og videre arbeid

I dette siste kapittelet vil vi summere opp hvor prosjektet står etter vårt arbeid, og antyde noen forslag til videre arbeid.

5.1 Oppsummering

Oppgaven startet altså med et ønske om å implementere den generiske mekanismen Package Templates (1) i Java. Som en del av oppgaven skulle vi selv velge hvilken måte vi ville gjøre implementasjonen på. Siden selve syntaksen og funksjonaliteten i språket langt på vei var ferdig definert, ble det viktigste i starten å sette seg inn i hvordan PT-mekanismen virker og hvilke implementasjons-alternativer som fantes. Vi startet med å se på alternativer for frontend-delen til JPT-kompilatoren. Vi studerte mulighetene for å skrive kompilatoren selv eller å bygge ut en allerede eksisterende kompilator. Etter å ha veid fordeler og ulemper opp mot hverandre fant vi ut at JastAddJ var det beste valget.

JastAddJ er et omfattende program med rundt 22.000 linjer kode fordelt på ca 130 moduler. Det har vært en omfattende jobb å sette seg inn JastAddJ og mye leting etter metoder som tilsynelatende ikke blir definert noe sted. I etterkant ser vi at JastAddJ er vel strukturert og godt bygd opp, men probleme lå heller i manglende dokumentasjon av systemet.

De nye delene vi så laget for å implementere PT-mekanismen har vi lagt i egne moduler og vi har gjort få endringer i de eksisterende modulene siden de stort sett ikke har noe med PT å gjøre.

Vi vurderte også to mulige måter å implementere backend-delen i JPT på, eksemplifisert ved Sørensens homogene og Blomfeldts heterogene implementasjon. Etter å ha sett på oppbygningen av disse og fordeler og ulemper ved begge implementasjonene tok vi valget om å bruke en homogen versjon, som også var vårt primære ønske da vi startet. Vi ville la oss inspirere av Sørensens versjon, men kunne ikke bruke prinsippene fra denne direkte da disse krever utvidelse av byte-koden.

Vår implementasjon viser at det er mulig å lage homogen implementasjon av PT som kjører på en standard JVM (i hvert fall med de begrensningene i språket vi brukte i vår implementasjon). Hvor raskt det produserte byte-kode programmet vil være under kjøring har vi dessverre ikke fått målt, og dette vil selvfølgelig være avgjørende for hvor vellykket prosjektet kan sies og være. Grovt sett vil det (med våre begrensninger på PT) bli ett, og av og til to, ekstra metode-kall for hvert metode-kall som krysser grensen mellom templat- og pakke-nivå. For variabel-aksess blir det i tilsvarende tilfeller ett ekstra indirekte steg. Dessuten blir det noe ekstra type-kasting (som alltid vil bli godkjent). Det vil bli spennende å se hvordan dette vil slå ut i praksis. Programmer som ikke bruker PT-mekanismene vil i hvert fall ikke bli påvirket. TID-objektene, som det er ett av for hver instans, tar noe ekstra plass i minnet, men dette vil normalt være svært lite i forhold til det duplikatene av runtime-koden ville kreve om vi hadde brukt en heterogen implementasjon.

Den faktiske koden vi har lagt til for å få JastAddJ til å kompilere JPT utgjør kun i underkant av et par tusen linjer. Det har med andre ord ikke vært det å skrive selve koden som har representert mesteparten av arbeidet med oppgaven, men snarere det å finne ut nøyaktig hvor vi skal sette inn denne koden og samtidig tenke ut smarte løsninger forbundet med implementasjonen.

Vi har også vist at man (i hvert fall med våre begrensninger) kan gjøre en full semantisk og syntaktisk sjekk av templaten før den instansieres. I templaten bruker vi JastAddJ sin semantiske og syntaktiske sjekk for vanlige pakker.

Vi kom altså ikke helt i mål med å lage kompilatoren ferdig i alle ledd, men det aller meste er ferdig programmert og testet. Hele frontend-delen fungerer godt, og for backend-delen er den eneste mangelen at program-delen som generer TD- og TID-klasser bygger på skjemaer som siden er noe forandret, og

det samme gjelder programet som utfører justerings-gjennomløpet. Det å få oppdatert disse program-delene burde ikke være noen ekstra stor jobb.

5.2 Mulig videre arbeid

I løpet av arbeidet med Package Templates har det oppstått mange interessante problemstillinger som vi av tidsårsaker ikke har fått utforsket ordentlig. Vi går igjennom noen av de viktigste her.

5.2.1 Komplettering av kompilatoren

Vi skal først gå gjennom de viktigste tingene som mangler for å få en komplett PT-kompilator, og diskutere kort om de vil gi noen prinsippielt nye problemer.

5.2.1.1 Type-parametere til templaten

På grunn av tid og arbeidsmengde har vi altså måttet avgrense oppgaven, og et av valgene her var å ikke implementere typeparametrisering på templat-nivå. Slik som kompilatoren framstår nå støtter den typeparametre kun på klassenivå (som i vanlig Java). Implementasjon av dette på templat-nivå ser ut til å gå greit både i frontenden og backenden. For frontenden kan man la JastAdd tro at ”templat-pakken” også har klasser og grensesnitt som tilsvarer parameter-begrensningene. Med tanke på runtime kan man legge koblingsbokser til de formelle parameter-klassene i TD-klassen, og for hver instansiering lage redefinerende metoder i TID-klassen som kobler disse til de aktuelle parametrene. Om dette vil løse alle detaljer må imidlertid studeres nærmere.

5.2.1.2 Instansieringer inne i templater

Ved å tillate instansieringer inne i templater vil vi ikke bare få to ”nivåer” i implementasjonen, men vilkårlig mange. Dette kan antakeligvis løses nokså rett fram ved å legge inn koblingsbokser mellom hver av nivåene, slik som med TD- og TID-klassene. Dette kan imidlertid virke negativt inn på eksekverings-hastigheten, da det kan bli ofte og lange skift mellom nivåene som krever mange ekstra metode-kall og indirekte variabel-aksesser. Her vil det være viktig å lete etter gode generelle optimaliseringer.

5.2.1.3 Bedre feilmeldinger ved runtime-feil

Det kunne være interessant å se mer på hvordan man kan få koden fra JPT-kompilatoren til å gi gode feilmeldinger under utførelsen. Det er da viktig at navnene som blir generert under instansieringen inneholder nok informasjon til at man kan finne ut hvor feilen har oppstått. Dette gjelder i særs grad feilmeldinger som stammer fra instansierte templater.

En løsning kan være å overstyre JVMs feilhåndtering for de generiske pakkene, parsere feilmeldingen selv, og gjøre den om til en feilmelding som er forståelig på PT-nivå, uten at man må vite noe om implementasjonen. Dette kan for eksempel gjøres ved at kompilatoren i start-metoden i programmet legger inn en mekanisme som håndterer unntak. Det er i så fall viktig at denne mekanismen er skrevet slik at den skiller mellom vanlige Java-unntak og unntak som stammer fra de feilsituasjonene vi vil forandre meldingene fra.

5.2.2 Optimalisering

Dette er den delen av oppgaven vi skulle ønske vi fikk mer tid til. Det er mange måter å implementere JPT på. Slik kompilatoren fremstår nå er JPT implementert på en rett fram måte uten spesiell tanke på effektiviteten av den produserte koden. Vi er sikre på at vi kunne fått den til å bli en god del raskere hvis vi hadde hatt mer tid til å se på dette. Som eksempler på hva man kunne vurdere kan vi nevne følgende:

5.2.2.1 Under kompileringen

Når det gjelder hastigheten av kompileringsprosessen mener vi det er mulig å generere byte-kode for JPT direkte fra (en mer omarbeidet) JastAddJ. Dette krever imidlertid at man setter seg dypere inn i hvordan JastAddJ genererer byte-kode, og at man derved kan ta i bruk flere av funksjonene JastAddJ benytter seg av. Med en slik løsning vil man slippe mange innlastinger av klassefilene med BCEL og mange oppslag etter objekter og metoder, dermed vil man spare tid under kompileringen.

5.2.2.2 Runtime

For å få så kode som mulig bør man generelt bytte ut den/de instruksjonene

som JastAddJ har generert med så få nye som mulig og legge vekt på at de instruksjonene som brukes mye, slik som invokevirtual, går raskt og heller la det gå på bekostningen av at de instruksjonene som brukes sjeldnere. Man kan også se på om man i de tilfeller man ved instansieringer bare har enkle utvidelser av templat-klasser. Kanskje kan flere av del-objektene da slåes sammen til ett objekt?

Referanser

1. **Krogdahl, Stein, Birger, Møller-Pedersen og Sørensen, Fredrik.** *Exploring the use of Package Templates for flexible re-use of Collections of related Classes.* Oslo : Department of Informatics, University of Oslo, Norway, 2008.

2. **Krogdahl, Stein.** *Generic Packages and Expandable Classes, Research Report no. 298.* 2001.

3. **Sørensen, Fredrik.** *Generic Packages and expandable Classes in Java.* 2005.

4. **Blomfeldt, Eskil Abrahamsen.** *En heterogen implementasjon av generiske pakker og ekspanderbare klasser i Java.* 2005.

5. **Sun Microsystems, Inc.** JDK 5.0 Java Programming Language-related APIs & Developer Guides. *Java Programming Language.* [Internett] 2004. http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html.

6. **Jikes.** http://jikes.sourceforge.net/. [Internett]

7. **Görel Hedin, Eva Magnusson.** *JastAdd - an aspect-oriented compiler.* Lund, Sverige : s.n., 2003.

8. **Ekman, Torbjørn og Hedin, Görel.** The JastAdd Extensible Java Compiler. *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion.* 2007.

9. **Foundation, Apache Software.** BCEL - Manual. *BCEL.* [Internett] Apache Software Foundation, 3 June 2006. [Sitert: 14 04 2009.] http://jakarta.apache.org/bcel/index.html.

10. **Bracha, Gilad.** *Generics in the Java Programming Language.* 2004.

11. **Ellis, Margaret A. og Bjarne, Stroustrup.** *The Annotated C++ Reference Manual.* s.l. : Addison-Wesley Professional, January 1990.