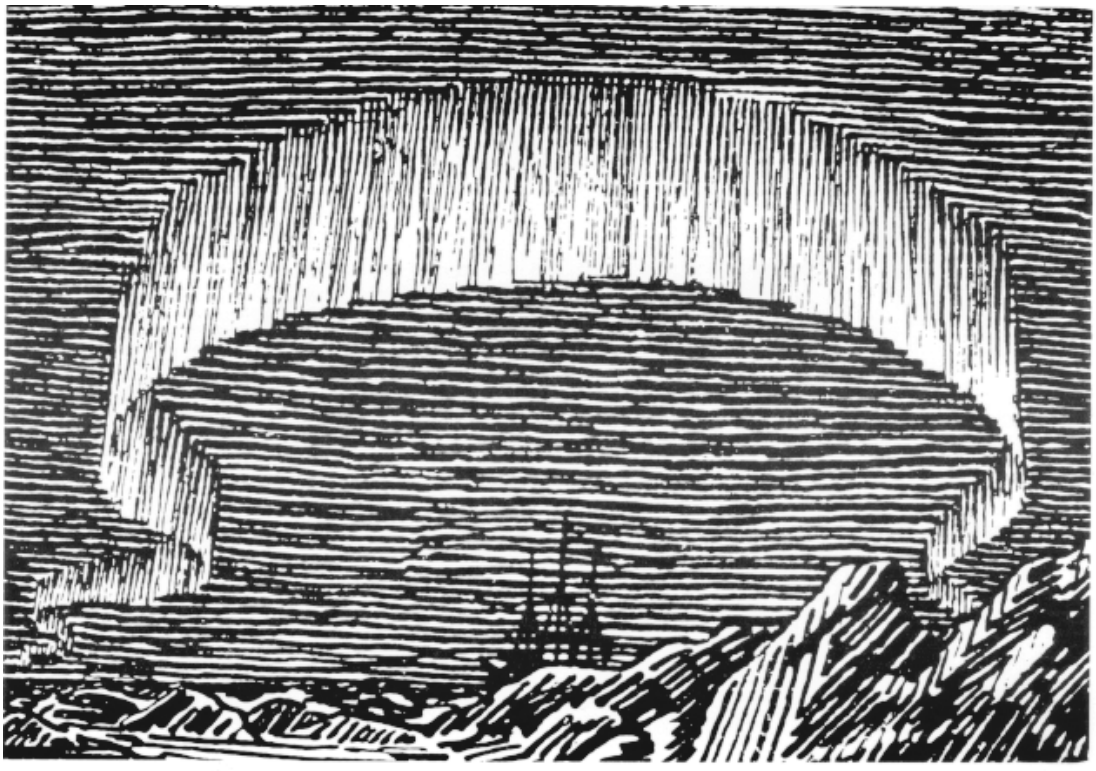
HOVEDOPPGAVE I INFORMATIKK

Objekt-adapter og programmeringsgrensesnitt for ANSAware applikasjoner

Øyvind Hanssen

1994



Original by F. Nansen

INSTITUTT FOR MATEMATISKE REALFAG
Seksjon for Informatikk
Universitetet i Tromsø, N-9037, Telefon 77 64 40 41, Telefax 77 64 45 80

Sammendrag

I denne hovedfagsavhandlinga undersøkes avbildningsmekanismer mellom ANSAware applikasjoner og en føderativ omgivelse. Dette gjøres innefor rammen av ODS-gruppas arbeid med samvirkende informasjonssystemer. For å få kunnskap om problemområdet og om hvordan avbildning effektivt kan utføres, utvikles et rammeverk for modellering, design og implementasjon og avbildningsmekanismer. Vi fokuserer spesielt på hvordan man i føderasjonen kan gi inntrykk av ANSAware objekter som persistente. Vi baserer oss på en persistensmodell som innebærer at vi stiller ulike krav til objektidentitet. Vi trenger bare permanent identitet for noen få objekter. For disse er det behov for mekanismer for transparent forvaltning (aktivisering/passivisering).

Vi undersøker to logiske komponenter som samarbeider om avbildning: Objekt-adapter som har ansvaret for forvaltning og objekt-identitet og språkbindinger som representerer programmeringsgrensesnitt for den aktuelle klient-omgivelse og realiserer aksess-transparens ved hjelp av stubs. Vi innfører begrepet proxy-objekt som representerer identifikasjon av objekter i objekt-adapter og presenterer en konseptuell modell for interaksjon med klient.

Et gjenbrukbart objekt-orientert rammeverk er realisert. Dette representerer design og delvis implementasjon av objekt-adapter. Denne påbygges med applikasjonsspesifikk software for å bli komplett. En notasjon for definisjon av forvaltning av permanente objekter blir utviklet.

Egenskaper ved språkbindinger blir undersøkt. Her gjøres et skille mellom direkte binding hvor klient er i samme prosess og binding via eksplisistt grensesnitt (kanonisk språk). Ei språkbinding til C++ er realisert, og det er skissert et rammeverk for binding til FRIL som er et funksjonelt og objektorientert språk for integrasjon og samvirke mellom ulike informasjonssystemer.

Forord

Denne Cand. Scient avhandlinga er initiert og veiledet av Frank Eliassen. Oppgaven er definert innenfor rammen av IIS-prosjektet (interoperable information systems) ved ODS gruppa ved Seksjon for Informatikk ved Universitetet i Tromsø.

Jeg vil takkel Frank Eliassen for kyndig veiledning, og ellers til alle som har vært behjelpelig med faglig og moralsk støtte. Her vil jeg spesielt nevne Ahmed Khalaila for nyttige kommentarer og råd.

Jeg vil spesielt takke min samboer Anne-Line Karlsen for moralsk støtte og forståelse til tross for at innspurten i hovedfagsarbeidet til tider har gått ut over familien. Arbeidet er tilgegnet min datter Ellen Miranda som fyller ett år i disse dager.

Tromsø
1. november 1994

Øyvind Hanssen

Innholdsfortegnelse:

Kapittel 1
Innledning

Behovet for samvirke mellom informasjonssystemer er stadig økende. Større organisasjoner har ofte sine datasystemer og informasjonsressurser spredt ut over ulike datamaskiner, databaser og applikasjoner. Organisasjoner kan gjerne være geografisk spredt, bestå av mer eller mindre autonome enheter. Disse vil ofte ha nytte av å kople sammen sine informasjonssystemer og kunne se helheten i den tilgjengelige informasjon. Organisasjoner vil også ofte ha interesse av samarbeid og deling av sine data med andre organisasjoner.

Informasjonssystemene er som oftest sterkt preget av heterogenitet. Det kan være ulike typer maskinutstyr og programvare fra ulike leverandører. Organisasjoner som har investert store beløp i sine informasjonssystemer ønsker også å utnytte disse så langt som mulig. Utfordringa blir da ofte å utnytte ulike eksisterende applikasjoner.

Problemet med å få til samvirke mellom autonome heterogene databasesystemer, har fått betydelig oppmerksomhet [ShLa90]. Målet omtales ofte som såkalte "føderative" databaser der hver enkelt database deltar i en føderasjon og deler sine data med andre uten å måtte oppgi for mye av sin autonomitet. I føderasjonen er man interessert i at bruker kunne anvende komponent-databaser som om det var en enkelt database (integrasjon) eller i det minste i et enkelt språk. Dette krever blant annet at det blir definert avbildninger for hvert enkelt komponent. Etterhvert har interessen også dreid seg om EDB-applikasjoner mer generelt (se f.eks. [Bertino89]), noe som er mer komplisert. Her har spesielt objekt-orientering vist seg å være nyttig for å overvinne kompleksitet.

1. 1. Problem-domenet

Dette prosjektet er definert innenfor rammen av IIS-prosjektet ved Seksjon for Informatikk ved Universitetet i Tromsø. IIS prosjektets mål er å utvikle en generisk infrastruktur med tjenester og verktøy for å muliggjøre samvirke mellom heterogene, autonome informasjonsressurser (databaser og andre applikasjoner).

Man søker å overvinne heterogenitet gjennom en kanonisk (opphøyet) datamodell. Denne modellen skal definnere de konsepter som er nødvendige for å beskrive abstrakte grensesnitt til alle mulige underliggende informasjonsbaser. På denne måten skal de kunne aksesseres på en enhetlig måte. En skal altså kunne skjule forskjeller mellom ulike spørrespråk og dataformat (datamodell-transparens) ved at brukerne kan betrakte de integrerte informasjonsbaser gjennom en kanonisk datamodell.

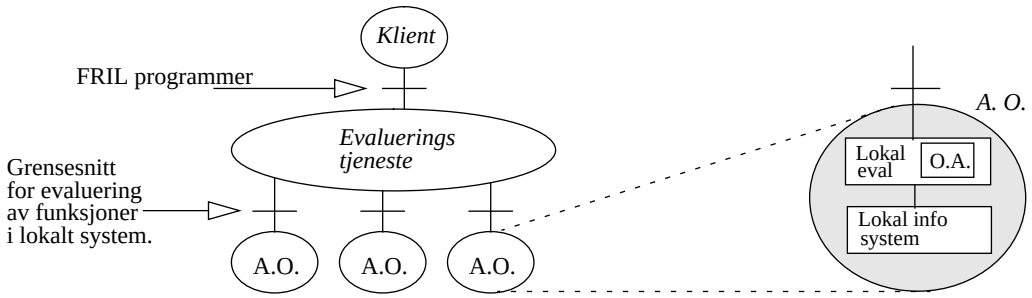
Det er valgt en kanonisk modell som kombinerer det objekt-orienterte og det funksjonelle paradigme. Her vil grensesnitt til applikasjoner beskrives som abstrakte datatyper (ADT) og applikasjonene vil aksesseres gjennom funksjoner. I funksjonene vil verdier og objekter (instanser av ADTene) kunne opptre som argumenter og resultater.

Man introduserer et programmeringsspråk som representerer en kanonisk data- og beregningsmodell. Dette språket (FRIL[1]) er både funksjonelt og objektorientert. Med FRIL kan man på den ene siden spesifi sær og eksportere informasjonstjenester (eksport skjema), og på den andre siden kan man importere og aksessere de ulike informasjonstjenestene.

1.1.1 FRIL-tjener komponenter

Vi kan skissere en forenklet modell av de komponenter som er involvert i utførelsen av FRIL-programmer: Vi har klienter, en (distribuert) evaluerings-tjeneste og et sett med applikasjonsobjekter, det vil si heterogene lokale informasjonssystemer som er "bundet" til evalueringstjenesten via avbildninger. Disse avbildningene omtales som lokal evaluering (utfører FRIL-fuksjoner ved å avbilde dem til tilsvarende operasjoner i det lokale system) og objekt-adapter (avbilder objekt-identitet og forvalter lokale objekter).

FIGUR 1. Forenklet modell av FRIL systemet



1.1.2 ANSAware

FRIL applikasjonsobjekter kan gjerne innkapsle applikasjoner som i seg sjøl er åpne og distribuerte. Her skal vi se på integrasjon av distribuerte applikasjoner bygd på ANSAware, som er en omgivelse for å utvikle distribuerte applikasjoner. ANSAware er en delvis implementasjon av ANSA-arkitekturen [AR.000.00] [TR.38.00]. I dette inngår en objekt-orientert beregningsmodell [AR.001.01], samt mekanismer som må til for at objektene skal kunne kjøre og interagere med hverandre. ANSAware tilbyr ei språklig overbygning over ANSAware arkitekturen, både for å spesifi sær grensesnitt til objekter og for å spesifi sær interaksjoner mellom grensesnitt.

ANSAware inneholder blant annet programmeringsspråk, programutviklingsverktøy, systemtjenester og systembibliotek [Netm91].

1. 2. Problemstillinga

I denne avhandlinga studerer vi mekanismer for avbildning mellom en global føderativ omgivelse og ANSAware applikasjoner. Det vil i IIS-prosjektets terminologi si at vi studerer virkemåte og oppbygning til software komponenter som representerer lokal-evaluering og objekt-adapter.

På et føderativt nivå ønsker vi å oppnå datamodell-transparens, det vil si uniform akess til underliggende systemer, der forskjeller i datamodeller og spørrespråk/interaksjonsmekanismer er skjult for brukeren. Vi skal se at datamodell-transparens må understøttes av mekanismer for blant annet følgende:

1. Functional Resource Integration Language

- Aksess-transparens som betyr å skjule forskjeller i data-representasjon og anropsmekanismer.
- Transparent forvaltning av ANSAware objekter (aktivisering/passivisering) og relokering av bindinger til deres grensesnitt.

Vi er spesielt interessert i å se på den forvaltning av ANSAware objekter og deres identifikatorer som er nødvendig for på et føderativt nivå å gi et bilde av ANSAware objekter som persistente. Det vil si vi er interessert i hvordan vi på en transparent og effektiv måte kan utnytte ANSAware applikasjoners iboende persistens.

1.2.1 Delproblemer

I forbindelse med behandlinga av denne problemstillinga, er det relevant å stille følgende spørsmål (delproblemstillinger):

- Hvordan kan persistente ANSAware-objekter identifiseres, både globalt og lokalt? Vi er interessert i å se på ulike aspekter ved objekt-identifikasjon og hvordan de forholder seg til hverandre.
- Hvordan kan vi på en effektiv måte konstruere og implementere avbildningsmekanismer (local-eval/objekt-adapter) for ANSAware applikasjoner.
- Hvordan kan vi konstruere effektive og gjenbrukbare avbildingsmekanismer? Effektivitet er et vidt begrep. Her vil vi først og fremst fokusere på aspekter som ytelse, skalerbarhet og anvendelighet.

1. 3. Mål for arbeidet

Målsettinga er å utvikle et rammeverk for modellering, design og implementasjon av avbildningsmekanismer mellom en føderativ omgivelse og ANSAware-applikasjoner. Et slikt rammeverk vil representere kunnskap om problemområdet og kunnskap om hvordan avbildningsoppgavene effektivt kan utføres.

Vi ønsker for det første å klargjøre hvilke komponenter som vil være involvert i avbildningsprosessen og hvordan disse forholder seg til hverandre. Vi ønsker å utvikle abstraksjoner som er egnet til å beskrive oppbygning og virkemåte til avbildningsmekanismene.

Vi ønsker for det andre å utvikle rammeverk og verktøy for design og implementasjon av avbildningsmekanismer som et klient program kan bruke for transparent aksess til ANSAware applikasjoner. Vi skal realisere en prototype implementasjon av dette. Denne vil bestå av programbibliotek og verktøy for å generere applikasjons-spesifikk kode. Rammeverket med tilhørende verktøy skal kunne brukes som hjelpemiddel for å realisere avbildninger (objekt-adapter, stubs og programmeringsgrensesnitt) for konkrete ANSAware applikasjoner.

Rammeverket vil ikke bare beskrive avbildning til FRIL-omgivelsen, men vil i utgangspunktet være så generelt at det kan anvendes for flere språk og interaksjonsmodeller. Vi ønsker å identifiser prinsipper som kan gjelde for ulike typer språkbindinger. Vi skal ut fra dette realisere et programmeringsgrensesnitt (språkbinding) til C++ og vi skal se spesielt hvordan vårt rammeverk også kan spesialiseres for FRIL. Grensesnittet til FRIL kan realiseres ved hjelp av ANSAware, og vi skal undersøke spesielt hvordan dette kan gjøres.

Gjennom utvikling og analyse av et slikt rammeverk, er det meninga vi ikke bare skal lære noe om avbildning av ANSAware applikasjon, men vi skal kunne lære noe om konstruksjon av objekt-adaptere/språkbindinger generelt. Vi vil se i avhandlinga at mange begreper er gyldig for et mer bredt spekter av lokale informasjonssystemer. ANSAware blir dermed på mange måter et spesielt case.

1.3.1 Avgrensinger

Det er ikke plass til å gi en skikkelig behandling av alle sider ved avbildning innenfor rammen av et Cand. Scient arbeide. Derfor er det slik at noen del-spørsmål blir behandlet mer grundig enn andre og noen problemstillinger vil vi se bort fra i denne sammenheng:

- Vi undersøker ikke avbildning mellom lokale og globale skjema, og spørsmål rundt skjema-arkitektur og skjema integrasjon. Vi er derimot interessert i hvordan vi kan defi nær software for avbilding av objekter og anrop, gitt at vi har opplysninger om en bestemt applikasjon. Her er lokale skjema og kjennskap til hvordan applikasjonen virker relevant.
- Vi fokuserer spesielt på persistens og transparens mht. forvaltning av objekter (aktivisering og passivisering). Dette har sammenheng med objekt-identitet som vi også ser på som viktig.
- Når vi evaluerer rammeverket, ser vi på noen viktige sider ved ytelse, skalerbarhet, anvendelighet og generalitet. Vi har derimot ikke sett spesielt på feiltoleranse.

1. 4. Oversikt over relatert arbeid

Historikk, bakgrunn og begrepsmessig rammeverk vil bli grundig behandlet i avsnitt 2. Her skal vi kort oppsummere relaterte arbeider og da framstår Comandos prosjektets operasjonelle avbildning [Bertino89] og Zoo_{IFI}-prosjektets integrasjonsrammeverk [HaeDitt93] som sentrale. Begge prosjektene har fremhevet objekt-orientering som ei viktig tilnærming til integrasjon og samvirke med heterogene informasjonssystemer. Ellers er det verdt å nevne de viktigste arbeidene med følgende:

- Persistens og objekt-identitet: Vi har basert oss på en global persistensmodell [Elia93a, Banc87, KhVa90]. Når det gjelder ANSAware spesielt, kan vi nevne [Blair92, Olsen92] som har behandlet infrastruktur-støtte for persistens og migrering. Ei klassifi sering av objekt-identititet er gjort i [ElKa91b, HaeDitt93].
- Arkitektur: Foruten ANSA [AW-APM, AR 001.01] og ODP [RM-ODP.1], har vi studert CORBA [OMG 91.12.1]. Her har beskrivelsen av BOA (Basic Object Adapter) betydd mye for forståelsen av hva en objekt-adapter gjør og hvordan den kan konstrueres.
- Programmeringsgrensesnitt: [OMG 93.9.2] beskriver språkbinding til C++ (for CORBA). Når det gjelder binding til FRIL, blir det gjort et parallelt Cand. Scient arbeide som ser på en evalueringstjeneste for interaktiv FRIL programmering, med grensesnitt for eksterne funksjoner [Sundsfjord94].

1. 5. Oversikt over rapporten

- Kapittel 2 introduserer bakgrunn og begrepsmessig rammeverk for resten av avhandlinga. Her er behandles historikk og en del relatert arbeid, og her gis det en introduksjon til ANSAware.
- Kapittel 3 behandler hvordan en på et føderativt nivå skal oppnå transparent interoperasjon med ANSAware applikasjoner. Fokus er på persistens og vi ser på hvordan ANSAware passer inn i den globale persistensmodellen, vi ser på betydninga av objekt-identifi kasjon og på forvaltning (aktivisering/passivisering).
- Kapittel 4 presenterer overordnede prinsipper for avbildningene mellom en føderativ omgivelse og et lokalt informasjonssystem. Vi identifi serer to samarbeidende komponenter, nemlig objekt-adapter og språkbinding (programmeringsgrensesnittet). Vi innfører begrepet "proxy-objekt" som er en form for representant (i objekt-adapter) for objekter i de lokale informasjonssystem. Vi presenterer også en modell for interaksjon med klienter.

- Kapittel 5 presenterer design av et objekt-orientert rammeverk for implementasjon av objekt-adaptere. Her identifiseres generiske deler og ANSAware-spesifikke deler Vi beskriver både statiske egenskaper (klassehierarki) og dynamiske egenskaper (hvordan objektadapter oppfører seg i ulike situasjoner).
- Kapittel 6 presenterer et rammeverk for definisjon av objekt-adaptere, gitt at vi har en spesifikk applikasjon. Her identifiseres de sentrale parametre, samt at det presenteres en notasjon for spesifikasjon av permanente objekter.
- Kapittel 7 identifiserer og diskuterer prinsipper for språkbindinger. Vi ser nærmere på hva det vil si å definere språkbindinger, for tilfeller der klient og objekt-adapter kjører i samme prosess og tilfeller der de er logisk atskilte og kommuniserer gjennom en protokoll. Ei språkbinding til C++ er realisert og den presenteres her.
- Kapittel 8 diskuterer en del sentrale aspekter ved binding til FRIL og presenterer et rammeverk for realisering av FRIL-språkbindinger.
- Kapittel 9 presenterer prototype-implementasjoner av rammeverket og diskuterer dette med hensyn til kvalitetsparametre som ytelse, skalering, anvendelighet og generalitet.

Kapittel 2
Bakgrunn og begrepsmessig rammeverk

I dette kapitlet introduseres det begrepsmessige rammeverk for resten av avhandlinga. Her skal vi presentere en del aspekter ved den abstrakte objekt modellen som ligger til grunn for integrasjon av ulike informasjonssystemer med FRIL, vi skal definere begrepet persistens og hva det innebærer integrasjon av ANSA applikasjoner, samt at vi skal definere arkitektur-komponenter som er relevant for diskusjonen videre. Vi skal også ha en mer grundig presentasjon av ANSAware. Men først skal vi presentere en del relevant relatert arbeid, både med hensyn til multidatabaser og åpen distribuert databehandling.

2. 1. Historisk bakgrunn

2.1.1 Multidatabaser og føderative databaser

Et føderativt databasesystem (FDBS) er i følge [ShLa90] en samling av samarbeidende, men autonome komponent-databasesystemer. Disse kan være integrert i ulik grad[1]. Et viktig aspekt med en FDBS er at komponent-databaser beholder sin autonomi, samtidig som den deltar i en føderasjon. [ShLa90] setter opp tre ortogonale dimensjoner for karakterisering av systemer som består av flere samarbeidende databasesystemer:

- Distribusjon - Hvordan data er distribuert (eventuelt replisert) på flere databaser. For FDBS er mye av distribusjonen bestemt ut fra at komponent-databaser eksisterer før en FDBS blir etablert.
- Heterogenitet - I databasesystemer kan vi klassifisere heterogenitet som (1) forskjeller i DBMS og datamodell, deriblant syntaktisk heterogenitet (forskjeller i hvordan data er representert og strukturert), og (2) semantisk heterogenitet som angår betydninga av data.
- Autonomi - Eiere av en komponent-database skal i minst mulig grad behøve å gi fra seg kontroll over databasen. En skal kunne stå fritt i valg av design (data, datamodell, navngiving, semantisk betydning av data, integritetsregler mm..), og i valg av kommunikasjonspartnere. Dessuten bør ikke direkte lokale operasjoner forstyrres av eksterne operasjoner, og en skal kunne stå fritt mht. hvorvidt og i hvilken grad ressurser skal deles med andre.

Taksonomi

[Litwin90] skiller mellom to noe ulike tilnærminger til multidatabase problematikken. Den første kalles "mulitidatabase" (jfr. f.eks. [LitAb86]) og beskriver systemer som er løst koplet i den forstand at en ikke har global integrasjon og globale skjema. Dette først og fremst for å bevare full autonomi for komponent-databasene. Den andre tilnærminga kalles "føderative databaser" og kan sies å være et kompromiss mellom ønsket om integrasjon av data og ønsket om autonomi [HeLo85]. Komponentene i føderasjonen kontrollerer her interaksjonene ved hjelp av eksport-skjema og import-

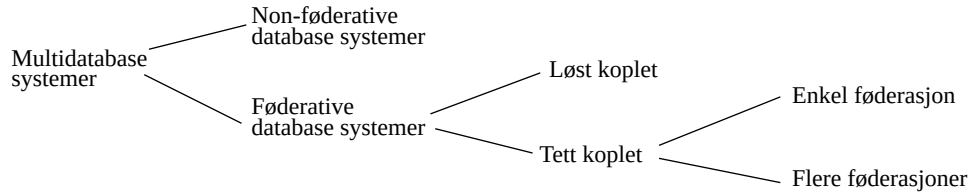
1. Med integrasjon menes at flere skjema (fra ulike komponent-databaser) integreres til ett enkelt skjema.

skjema. Et eksport-skjema definerer hvilken informasjon en komponent er villig til å dele med andre og et import skjema definerer hvilken ekstern informasjon en komponent ønsker å aksessere. En arkitektur er definert, med mekanismer for blant annet forhandling mellom komponenter om kombinering av informasjon og koordinering av aktiviteter.

[ShLa90] opererer med en videre definisjon av ordet "multidatabase". Her klassifiseres multidatabasesystemer som enten non-føderative (integrasjon av komponenter som ikke er autonome) eller føderative (autonome, samarbeidende komponenter uten sentralisert kontroll og med ulike grader av integrasjon).

Føderative systemer (som er fokusert på her) kan enten være løst koplet (det er brukeres ansvar å opprette og vedlikeholde føderasjoner og definere hvordan komponenter skal være integrert) eller tett koplet (føderasjonen og dens administratorer har ansvar for føderasjoner og kontroll av aksess til komponentdatabaser). Løst koplete systemer vil kunne ha flere føderasjoner, mens tett koplete systemer vil kunne klassifiseres i de som har en enkelt føderasjon (bare et føderativt skjema) og de som har flere. Figuren nedenfor oppsummerer denne taksonomien.

FIGUR 2. Taksonomi for multidatabase systemer



Det har blitt utviklet mange multidatabase-arkitekturer og prototype implementasjoner som kan plasseres ulike steder i denne taksonomien. Vi skal ikke gå mer i detalj med disse her, men kort nevne arbeidet til [ElVe88] som skisserer en klient-tjener arkitektur der man har en eksplisitt kontrakt mellom en klient og flere tjenere for informasjonsutveksling. Utveksling skjer gjennom predefinerte transaksjoner. Denne arkitekturen (som er forløper til FRIL), kan klassifiseres som tett koplet med flere føderasjoner.

Referanse arkitektur

Som et hjelpemiddel for å klargjøre og diskutere problemstilllinger rundt databasesystemer, presenterer [ShLa90] en referanse arkitektur. Ulike konkrete arkitekturer kan beskrives ved hjelp av denne. Referanse-arkitekturen definerer et sett med systemkomponenter som et databasesystem består av. Disse er som følger:

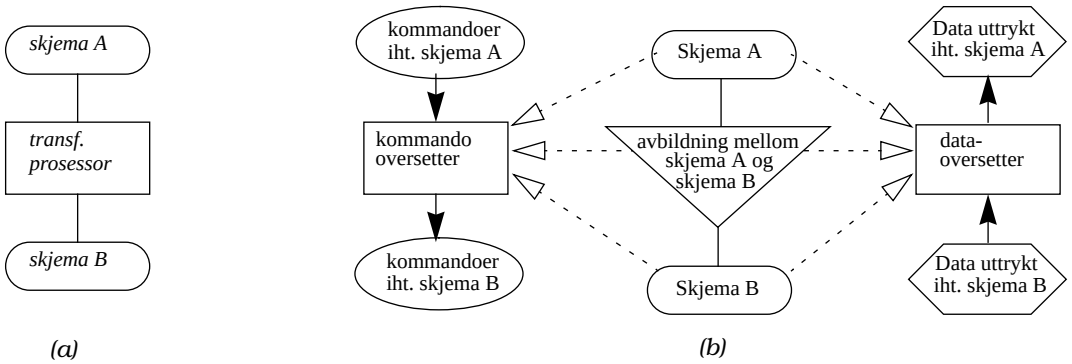
- Data - Informasjonen som forvaltes av et databasesystem.
- Database - Samling av data som er strukturert i henhold til en datamodell.
- Kommandoer - Forespørsler etter bestemte aksjoner. Disse genereres enten av brukere eller av prosessorer.
- Prosessorer - software moduler som manipulerer kommandoer og data.
- Skjema - Beskrivelser av data.
- Avbildninger - Funksjoner som korrelerer skjema-objekter i et skjema med skjema-objekter i et annet.

Videre definerers fire ulike typer prosessorer. Disse er som følger:

- Transformasjons-prosessor - Entitet som oversetter kommandoer[2] fra et språk til et annet, eller oversetter data fra et format til et annet (kommandoer og data i henhold til to skjema). Transformasjonsprosessor understøtter datamodell transparens, det vil si at datastrukturer og kommandoer som brukt av en prosessor er skjult for andre.
- Filter-prosessor - Entitet som avgrenser hvilke kommandoer og assosierte data som kan gis til en annen prosessor.
- Konstruksjons-prosessor - Entitet som partisjonerer og/eller repliserer en operasjon gitt av en enkelt prosessor, i operasjoner som kan aksepteres av to eller flere andre prosessorer (vil slå sammen returnerte data). Denne kan understøtte lokasjons-, distribusjons- og replikasjonstransparenser, og den kan utføre skjema-integrasjon, forhandling, dekomponering og optimalisering av spørringer og global transaksjon-forvaltning.
- Aksess prosessor - Software entitet som aksepterer kommandoer og produserer data ved å utføre kommandoene mot en database.

En transformasjonsprosessor er avhengig av avbildninger mellom objekter i hvert skjema. Ved oversetting av skjema (f.eks. fra en lokal til en global datamodell) vil en også kunne generere avbildninger som korrelerer skjema-objekter i de to ulike skjema. En transformasjonsprosessor vil bruke slike avbildninger, som da enten kan være kodet inn i transformasjonsprosessoren eller være lagret i en separat datastruktur som blir aksessert av transformasjonsprosessor. Det er mulig å generere transformasjons-prosessor automatisk fra slik avbildningsinformasjon. Figuren nedenfor (fra [ShLa90]) viser hvordan en transformasjonsprosessor kan instansieres og definieres ut fra skjema og skjema-avbildning.

FIGUR 3. Abstrakt transformasjonsprosessor (a) og par av samhørende transf. pros. (b)



Fem-nivå skjema arkitektur

En tradisjonell tre-nivå arkitektur (jfr. f.eks. [ElNa89]) må utvides for å understøtte distribusjon, heterogenitet og autonomi. Litteraturen har presentert flere slike arkitekturer. I [ShLa90] blir disse sammenfattet i en fem-nivå arkitektur med følgende skjema:

- Lokalt skjema (spesifisert i komponentdatabasens egen datamodell).
- Komponent skjema - Det lokale skjema oversatt til en kanonisk (felles) datamodell. Ved hjelp av denne kan ulike lokale skjema beskrives med en enkelt form for representasjon, og semantikk som mangler i det lokale skjema kan adderes. Dette for å forenkle integrasjon, forhandling, spesifikasjon av "views" og globale spørringer Overgangen mellom lokalt skjema og komponent skjema understøttes av en transformasjonsprosessor som bruker avbildninger generert ved oversetting mellom de to skjema.

2. Flyt av kommandoer og data her tilsvarer operasjonsanrop og returverdier i operasjonelle grensesnitt.

- Eksport skjema - Et subsett av komponent skjema som definerer hva som skal gjøres tilgjengelig for brukere i føderasjonen. Overgangen mellom komponent skjema og eksport skjema understøttes av en filterprosessor.
- Føderativt skjema - Integrasjon av flere eksport skjema til et (eller flere) føderative skjema. Overgangen mellom eksport skjema og føderativt skjema understøttes av konstruksjonsprosessor.
- Eksternt skjema - Et subsett av føderativt skjema som definerer hva som skal være tilgjengelig for en bestemt bruker eller en klasse av brukere (dette understøttes av en filterprosessor).

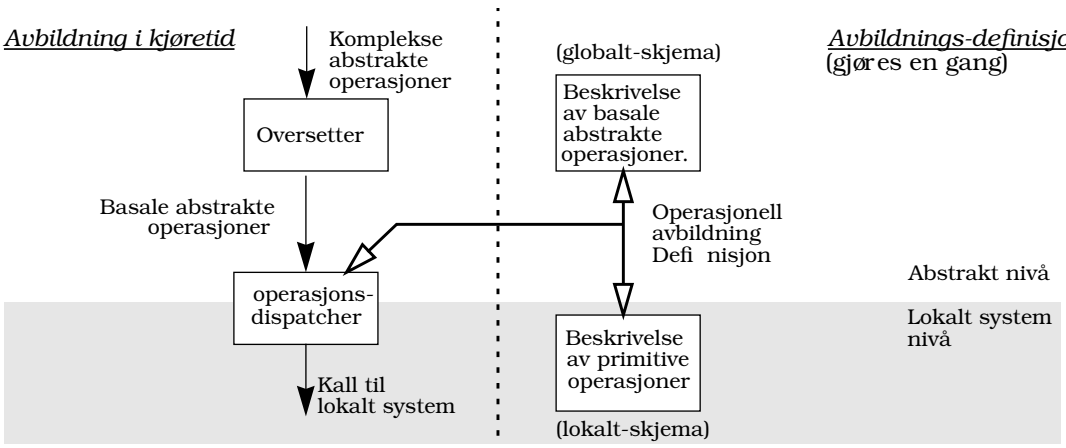
2.1.2 Operasjonell avbildning og objekt-orientering

Comandos-prosjektet [Bertino89] presenterer ei tilnærming til integrasjon av heterogene databaser, basert på operasjonell avbildning og objekt-orientering. Tradisjonell strukturell avbildning er ikke i stand til å håndtere alle former for databaser. Spesielt ser dette ut til å gjelde grafiske applikasjoner og tekstlige databaser hvor semantikken til data er sterkt avhengig av hvordan programmer manipulerer data og hvor skjema gjerne mangler.

Operasjonell avbildning går ut på å definere korrespondanse mellom operasjoner på ulike nivå i stedet for data-elementer på ulike nivå (som er tilfelle ved strukturell avbildning). Dette har en direkte relasjon til objektorientering (sjøl om objektorientering ikke er noen forutsetning) og er spesielt godt egnet der en opererer med en objektorientert global datamodell.

Man definerer såkalte abstrakte "view" over de lokale informasjonssystem som skal integreres. Operasjonell avbildning går da ut på å implementere abstrakte operasjoner ved hjelp av (primitive) operasjoner til det underliggende systemet. Man tenker seg to nivå i denne avbildninga: For det første avbildninger mellom basale abstrakte operasjoner og kall til lokalt system. Dette er enkle "navigerende" operasjoner. Man kan videre tenke seg at basale operasjoner kombineres til komplekse abstrakte operasjoner som kan utføre asossiative spørringer. Figuren nedenfor illustrerer dette. Avbildning av basale operasjoner utføres av en operasjonsdispatcher som definerers fra avbildning mellom lokale skjema og globale skjema (tilsvarende definisjon av transformasjonsprosessor i avsnittet ovenfor).

FIGUR 4. Operasjonell avbildning



Comandos prosjektet presenterer en generell arkitektur. Dette er en klient/tjener modell der en tjener representerer en abstraksjon av (deler av) en eksisterende applikasjonsomgivelse. En baserer seg også på en objektorientert datamodell som forsøker å kombinere konsepter fra objektorienterte databaser med objektorienterte programme-

ringsspråk. I [Bertino89] fokuseres spesielt på en arkitektur for tjenere og problemstillinger rundt denne. En tjener består her av følgende komponenter:

- Implementation Class Module (ICM) - implementasjon av operasjoner i en gitt abstrakt klasse. Det er typisk flere ICMer pr. tjener.
- Object Data Manager (ODM) - Forvalter objekter. Hovedproblem for denne modul er identifikasjon av objekter. Den kalles av ICM ved opprettelse og aktivisering av objekter.
- Dictionary Manager - Forvalter kataloger med abstrakte klassedefinisjoner
- Query Processor - Generisk oversetter. Oversetter komplekse spørringer til basale abstrakte operasjoner.

Objekt forvaltning og identifikasjon

En klient vil oppfatte en objekt-identifikator som en unik, permanent og utvetydig referanse til et objekt i et lokalt system. Ikke alle lokale system tilbyr objekt-identifikasjon med slike egenskaper, derfor tenker man seg at en ODM simulerer eksistensen av objekt-identifikatorer over verdi-baserte systemer, men dette er ikke mulig under alle omstendigheter og derfor innføres en restriksjon, nemlig at en objekt-identifikator som returneres til en klient bare er gyldig innenfor en sesjon. Det innebærer at disse ikke kan lagres i filer eller utveksles mellom klienter

I en tjener identifiseres et objekt i en lokal database med en såkalt DBID som er en datastruktur som representerer den lokale form for identifikasjon av objektet. Denne inngår i en større datastruktur som kalles PODP (private object data part) som allokeres av ODM når et objekt blir aktivisert (gjort tilgjengelig for klient). PODPer settes inn i en tabell og adresse til et felt i denne tabellen fungerer som objekt-identifikator

2.1.3 Objektorienterte integrasjonsrammeverk

Et prosjekt kalt ZOO_{IFI} [HaeDitt93] bygger på operasjonell avbildning (jfr. [Bertino89]) og utvikler konseptet videre. Man adopterer det såkalte rammeverk-konseptet (se f.eks. [Wirf90a, Wirf90b]) og skisserer et integrasjonsrammeverk som inneholder komplett design for og betydelige deler av implementasjon av den softwaren som er nødvendig for å etablere et såkalt homogeniseringslag over ulike autonome databasesystemer (Dette er for øvrig en ide som jeg bygger videre på i avsnitt 5. 1).

Denne tilnærminga reduserer kompleksiteten til den software som er nødvendig for å "plugge inn" en ny komponent i systemet. Det eneste en da trenger å gjøre er å implementere abstrakte metoder i passende subklasser.

Skjema arkitekturen er relativt enkel. De lokale skjema tenkes avbildet til komponentskjema som inngår i et føderativt skjema. Komponent- og føderative skjema er i henhold til en global (kanonisk) objekt-orientert datamodell. Det føderative skjema kan også "berikes" med typer som ikke inngår i noen komponent skjema. Sett i lys av 5-nivå arkitekturen i avsnitt 2.1.1 har man ingen eksportskjema og eksterne skjema.

Integrasjonsrammeverket har en intern database for å lagre instanser av de ekstra typene, og for å lagre nødvendig metainformasjon. Denne databasen kan oppfattes som en komponentdatabase i seg sjøl.

Objekt identitet

[HaeDitt93] diskuterer problemstillinger rundt objekt-identitet. Ikke alle CDBSer kan tilby sterk objekt-identifikasjon (jfrf.eks. [ElKa91a]), men det skal ikke være nødvendig å kreve sikker identitet fra alle CDBSer (og dermed utelukke noen), eller sette generelle begrensinger på hvor sterk identitet man opererer med på det globale nivå slik det

gjøres i [Bertino89]. En kan heller sette begrensinger på bruk av de enkelte objekter, det vil si hvilke typer operasjoner som er tillatt. Man tenker seg at slik brukbarhet spesifiseres som en egenskap ved klasser i komponentskjema. Tre alternativer blir skissert:

- Permanent identitet - Identifikasjon av et objekt er garantert å ikke endres så lenge objektet eksisterer. Det betyr at global referanse til slike objekter kan lagres og utveksles mellom klienter, samt at slike objekter kan delta i assosiasjoner definert på det globale nivå.
- Temporær identitet - HDBS[3] kan bare opprettholde relasjon mellom globalt og lokalt objekt så lenge det er aktivisert, det vil si mens det blir brukt av en klient på det globale nivå. Dette tilsvarer sesjons-OID i [ElKa91a] og er altså den eneste form for objekt-identitet i [Bertino89]. Slike objekter kan ikke lagres, utveksles eller delta i globale asossiasjoner.
- Imaginær identitet - HDBS kan ikke identifisere en lokal dataenhet for et gitt globalt objekt. Dermed kan oppdatering og opprettelse av nye objekter ikke tillates. I så tilfelle kreves temporær eller permanent identitet.

Permanent identitet kan i noen tilfeller oppnås sjøl om CDBS har en svakere form for identitet. For eksempel hvis en lokal verdi-basert nøkkel er garantert å aldri endres, eller hvis HDBS holdes løpende orientert om endringer. Det siste innebærer en viss svekkelse av autonomi (jfr. [ElKa91a])

2.1.4 Åpne distribuerte systemer og standardisering

Det er verdt å nevne et par viktige aktiviteter som fokuserer på arkitekturer for åpne distribuerte systemer og som har som mål å utvikle standarder (eller rammeverk for seinere standardisering) av begrepsapparat og infrastruktur for å understøtte distribusjon og åpenhet. Aspekter ved åpenhet er interoperabilitet og portabilitet. Disse aktivitetene har en sterk fokus på objektorientering, da dette paradigmet understøtter distribusjon og åpenhet på en naturlig måte med sin abstraksjon, innkapsling og polymorfi (Se f.eks. [Nicol93] eller [Blair91]). ANSA-prosjektet (som vi skal komme tilbake til) begynte tidlig med arbeid med sin arkitektur, som etter hvert har hatt innflytelse på flere standardiseringsarbeider på dette feltet, spesielt ISO/IECs aktivitet med RM-ODP, men har også vært aktiv i forbindelse med OMGs arbeid med en arkitektur for såkalte "object request brokers".

ODP-rammeverket

RM-ODP (Basic Reference Model of Open Distributed Processing) er et rammeverk (som det arbedes med innenfor rammen av ISO), for standardisering av modeller, funksjoner, grensesnitt og protokoller for å understøtte sentrale egenskaper ved åpne distribuerte systemer, nemlig distribusjon, interoperabilitet og portabilitet. RM-ODP er ment å gi et helhetlig og sammenhengende bilde av hva slags enkelt komponenter en har i et ODP system, men skal ikke gå så langt som å standardisere de enkelte komponenter eller påvirke valg av teknologi.

ANSA-prosjektet har hatt en sterk innflytelse på standardiseringsarbeidet rundt ODP og en finner mange av de samme konseptene i ODP-rammeverket og ANSA-arkitekturen. Vi skal komme inn på ANSA og ANSAware (og dermed også en del sider ved RM-ODP) i mer detalj i avsnitt 2. 7, og vi skal derfor bare kort oppsummere RM-ODP :

---

3. Heterogenous Database System - som iflg. [HaeDitt93] inneholder en føderasjon av komponent-databaser.

- Det blir definert 5 ulike "viewpoints" som fokuserer på ulike aspekter ved et åpent distribuert system. Med hvert av disse er det et assosiert språk som kan brukes til å beskrive et ODP system slik det ser ut fra den aktuelle "viewpoint".
- Som ei felles tilnærming til modellering i de ulike "viewpoints" har man valgt det objekt-orienterte paradigme. Man kaller dette objekt-modellering. Objekt orientering passer godt til ideer om modularitet og data-abstraksjon, ideen om tjenestetilbud (interaksjon), samt egenskaper ved system-komponenter som separasjon, isolasjon og autonomi.
- Det blir identifisert en samling funksjoner som byggeblokker for konstruksjon av systemer. Disse kan beskrives som objekter med definerte roller, informasjons-skjema, grensesnitt og atferd.

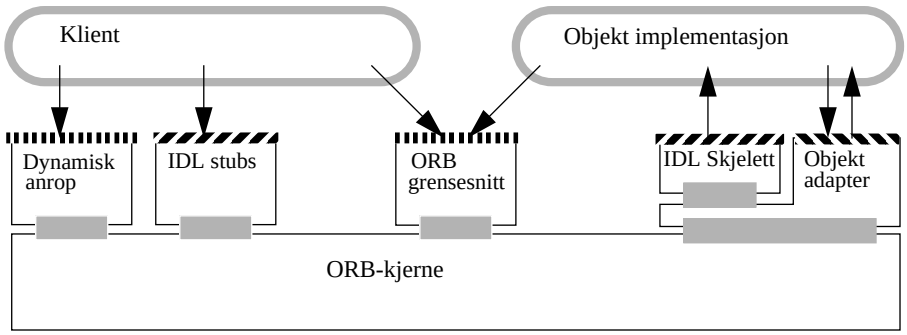
OMG og CORBA

Object Management Group (OMG) er et industrikonsortium som har som oppgave å produsere spesifikasjoner for kommersielt tilgjengelige objektorienterte omgivelser OMG definerer en arkitektur (CORBA[4]) for en såkalt "Object Request Broker" (ORB). Denne skal tilby mekanismer for å oppnå interoperabilitet mellom applikasjoner på ulike maskiner i heterogene omgivelser og skal på en sømløs måte kople sammen flere objekt-systemer (jfr. [OMG.90.9.1]). Hovedelementene i arkitekturen er som følger:

- En objekt-modell, det vil si et begrepsapparat for å definere objekter og deres interaksjon. IDL (Interface Definition Language) er et språk for å definere grensesnitt til objekter.
- Definisjon av struktur til en ORB. Det vil si beskrivelse av komponentene som en ORB består av og hvordan disse samarbeider.

Figuren nedenfor illustrerer strukturen til en ORB, det vil si hvilke komponenter som inngår og hvilke grensesnitt det er mellom dem. Klient og objekt-implementasjon er komponenter som anvender ORB. (Klienten anvender altså ORB for å anrope objekter som eksisterer i en objekt-implementasjon).

FIGUR 5. CORBA



Følgende komponenter er viktige i CORBA. (Stubs og skjelett vil måtte genereres for hver språk-avbildning og for hver objekt-type spesifisert i IDL):

- ORB-kjerne - Komponent som tilbyr basal representasjon av objekter og kommunikasjon av anrop.
- Klienter - Brukere av ORB som har aksess til objekt-referanser for objekter og anroper operasjoner på objekter. En klient ser på objekter og ORB-grensesnittet gjennom språk-avbildninger. En ORB skal kunne tilby avbildning til et eller flere (klient) programmeringsspråk. Et klient program skal være mest mulig portabel over

4. Common Object Request Broker Architecture [OMG.91.12.1].

ulike ORBer, det si den bør fungere uten videre på alle ORBer som tilbyr avbildning for det aktuelle språket. Det betyr at et konsept bør være likt representert i alle instanser av språk-avbildning for et gitt språk.

- Objekt-implementasjoner - Brukere av ORB som tilbyr semantikken til objekter ved hjelp av data for objekt-instanser og kode for metoder.
- Stubs - For ei gitt språkbinding vil det finnes et programmeringsgrensesnitt til stub'er for hver grensesnitt-type. Hver stub vil gi aksess til IDL-definerte operasjoner. En programmerer skal lett kunne forutsi representasjonen av en gitt IDL-spesifikasjon, gitt ei bestemt språk-avbildning. Stub'er gjør kall til kjernen av ORB på vegne av klienten for å få utført den aktuelle operasjon, gjerne på en måte som er optimalisert for den aktuelle ORB-kjerna.
- Dynamiske anrop - Et grensesnitt for dynamisk konstruksjon av anrop til objekter.
- Skjelett - For ei gitt språkavbildning, og muligens avhengig av objektadapter, vil det finnes et grensesnitt til metodene som implementerer hver objekt-type. Man tenker seg at objekt-implementasjonen tilbyr rutiner som kalles fra ORB gjennom skjelettet.
- Objektadapter - Komponent som har ansvaret for avbildning mellom objekt-referanser og objektenes representasjon i implementasjoner, aktivisering og passivisering av objekter og implementasjoner, generering og tolking av objekt-referanser, metodeanrop, sikkerhet mm. Man tenker seg at det vil finnes et sett med ulike objekt-adaptere, der hver av dem har grensesnitt som er egnet for spesifikke klasser av objekter.
- ORB-grensesnitt - Grensensitt for ORB-tjenester som er uavhengig av spesifikke grensesnitt eller objektadaptere.

En ORB vil også ha felles tjenester for å understøtte lagring av informasjon som er nyttig for komponentene . Vi har "interface repository" som inneholder IDL-informasjon på en form som er tilgjengelig i kjøretid, og vi har "implementation repository" som inneholder informasjon som ORB trenger for å lokalisere og aktivisere objekt-implementasjoner.

2. 2. Abstrakt objekt-modell

Vi skal nå forsøke å klargjøre hva som menes med begrepet 'objekt'. Her skal vi gjøre et klart skille mellom objekter og verdier. Det sentrale her er at objekter har en eksistens og identitet, uavhengig av hva slags informasjon objektet er bærer av. Verdier representerer informasjonen i seg sjøl. Vi skal først definere verdier:

2.2.1 Verdier

Verdier brukes for å beskrive ting, det kan være ting i den virkelige verden eller det kan være objekter som eksisterer i et EDB-system. Når man beskriver noe, bruker man symboler som representerer verdier. Verdier kan oppfattes som referanser til begreper eller objekter[5], som eksisterer i eller utenfor EDB-systemet. For eksempel verdien 'fredag' refererer til begrepet fredag. Alt som behandles i datamaskinberegninger er i bunn og grunn verdier. Vi kan si at alt som kan bli evaluert, lagret, inkorporert i datastrukturer, brukt som argument og resultat i funksjoner osv. er verdier i følge [Watt90].

5. [Beeri90] Beskriver verdier som en form for objekter (det vil si de har en identitet, hvert objekt er forskjellig fra alle andre objekter og de kan ikke endres). Verdier er videre objekter som ikke kan presenteres direkte, siden det er abstraksjoner over egenskaper. Skriver man da f.eks. tallet '7' er det da ikke noe annet enn et navn på objektet 7 som det bare finnes ett av. Man skriver aldri objektet sjøl, bare dets navn.

Det er nyttig å kunne gruppere verdier i typer. Man kan si at en type er en mengde verdier hvor et felles sett av operasjoner gir mening. For eksempel har noen operasjoner (som f.eks. aritmetiske operatorer som + - * /) mening på heltall, men ikke på sannhetsverdier.

Vi må videre skille mellom primitive typer og sammensatte typer. Primitive typer er atomiske verdier (ikke oppdelbar i mindre deler). Typiske slike er *bool, int, char* osv. Det er mulig å konstruere nye typer, hvor verdier er komponert eller strukturert fra enklere verdier. Velkjente eksempel på dette er f.eks. array, lister, tupler osv. Det er mange måter å konstruere sammensatte typer i ulike programmeringsspråk, men det er mulig å klassifisere disse i et lite antall fundamentale konsepter [Watt90]. Vi skal ikke gå nærmere inn i typeteorien, men kort oppsummere relevante strukturerings-konsepter:

1. Kartesiske produkt (tupler)
2. Disjunkte unioner
3. Mengder
4. Avbildninger (funksjoner)
5. Rekursive typer (lister).

FRIL har innebygd de atomiske verdi-typene *Bool, Int, Real, String* og *OID*. I tillegg har språket innebygd konstruktører for de sammensatte typene *lister*, *tupler* og *mengder*. Notasjonen vi bruker er slik::

- **T* uttrykker lister av verdier av typen *T*.
- *[T_1, T_2, ... , T_n]* uttrykker kartesiske produkt (tupler) av typene $T_1..T_n$.
- *{T}* uttrykker mengder av verdier av typen *T*.

2.2.2 Objekter

Et abstrakt objekt har den fundementale egenskapen at de har en uforanderlig identitet samtidig som det har en beskrivelse (tilstand) som kan forandres. Vi innfører følgende definisjon av et objekt:

Et objekt kan beskrives som et trippel: *(OID, Type, Tilstand)*.

Vi skal se på hva disse tre aspektene ved et objekt egentlig er:

Identitet

OID står for objektets identitet. Identitet er uavhengig av objektets tilstand. Det skal for eksempel være mulig for to forskjellige objekter å ha samme tilstand og likevel skal vi kunne skille mellom dem. Vi skal også kunne endre tilstanden til et objekt og fremdeles kunne si at det dreier seg om samme objekt (jfr. [KhosCop86]). Objekt-identitet kan representeres med objekt-identifikatører, som er verdier. De fungerer som unike referanser til objekter. Dermed understøtter modellen behovet for deling av objekter (referanse-deling) og objekter (representert ved identifikatørene) kan inngå i beregningsuttrykk på lik linje med andre verdier.

For å klargjøre betydninga av identitet kan vi innføre følgende definisjoner:

1. To objekter o_1 og o_2 er *identiske* (o_1 == o_2) hvis de har samme OID.
2. To objekter o_1 og o_2 er *like* (o_1 = o_2) hvis de har samme tilstand.

Type

Objekter har type. Vi kan si (som for verdier) at en type er en mengde objekter hvor et felles sett av operasjoner har mening. Dette settet av operasjoner definerer en felles abstrakt atferd for de objekter som hører til typen. Det settet operasjoner som kan utføres på et objekt kan vi også si utgjør objektets grensesnitt. Brukere kan kun aksessere objekter gjennom deres grensesnitt. Alt annet er skjult (prinsippet om innkapsling). Man kan gjerne si at grensesnittet har type (bestemt ut fra operasjonene og deres semantikk), eller man kan si at objektets type er bestemt ut fra dets grensesnitt.

En type kan altså spesifiseres som et sett med operasjons-signaturer[6] og eventuelt et sett med likninger som beskriver operasjonenes semantikk, slik den kan observeres fra en klient.

Objekt-tilstand

Et objekt har en tilstand som er en verdi[7]. Man kan si at verdien byttes ut med en ny når tilstanden til objektet endres. Det er ofte en kompleks sammensatt verdi som utgjør tilstanden og vi kan (siden vi opererer på abstrakte objekt-typer) bare observere denne tilstanden gjennom ekstraktør-operasjoner mot objektet. Vi kan ikke observere tilstanden som en verdi direkte.

Det er mulig å spesifisere et sett med operasjoner i en gitt objekt-type som definerer "alt som kan bli observert" av objektets tilstand. Disse operasjonene utgjør typens observasjons-basis [Elia93a, Dahl92]. Disse operasjonene behøver ikke nødvendigvis være en del av grensesnittet til det lokale systems obekt-type, eller være tilgjengelig for klient programmet, men er nyttige i formell spesifikasjon av semantikken til objekttypens operasjoner, og til på en abstrakt måte å beskrive tilstanden til et objekt.

2. 3. Strukturell komposisjon

Et objekt-orientert informasjonssystem vil bestå av et helt nettverk av objekter som er relatert til hverande på ulike måter. Her skal vi se på hvordan man kan beskrive hvordan strukturen til objektene og de relasjoner de har mellom seg.

2.3.1 Tilstand og objekt-graf

Objektets tilstand (som beskrevet i avsnitt 2.2.2) kan bestå av alle slags verdier, også objekt-identifikatorer. Disse må oppfattes som referanser til andre objekter. Hvis man videre evaluerer tilstanden til disse objektene (dvs. evaluerer deres observasjons basis) rekursivt, vil man etter hvert få et helt nettverk av objekt-referanser. Eller rettere sagt: En graf. Det er altså et objekts tilstand, evaluert på denne måten som utgjør objekt grafen. Her tar vi med en kort definisjon og et eksempel. Merk at utgående kanter fra i_3, og i_5 ikke er tatt med i figuren. (Se for øvrig [Elia93a], for en grundigere definisjon og eksempel).

- Et objekt er en node i grafen, og skrives med symbolet "●". Noden merkes også med objektidentifikator og navn på type. Slike har en utgående kant for hver ekstraktør funksjon i observasjonsbasis. Disse merkes med funksjonens navn.

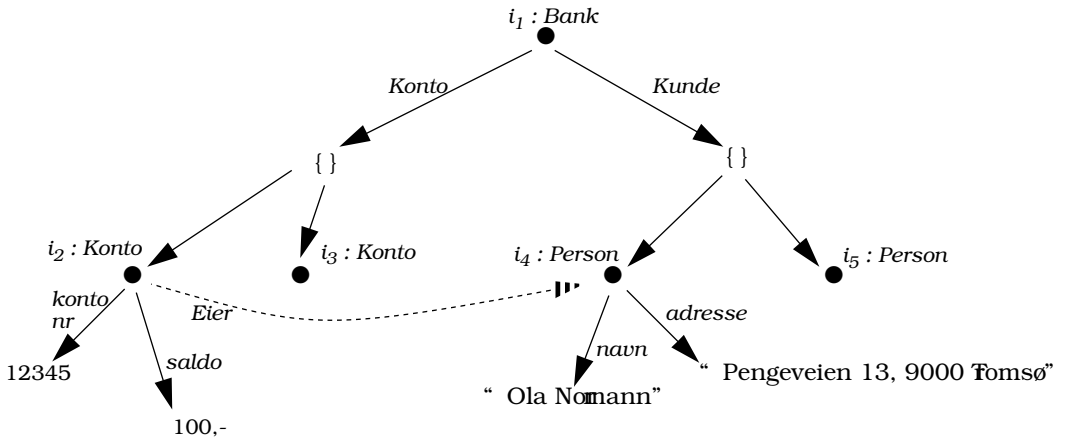
---

6. Definerer operasjonens navn, hvilke typer verdier operasjonen kan ta som argumenter og hvilken type verdi den returnerer.

7. Eller man kan like gjerne si "flere verdier". Det er ekvivalent med å si "et tuppel med verdier". O2 [Lecluse88] for eksempel, definerer objektets tilstand som et tuppel med verdier. FRIL opererer med et sett av observasjons-basis funksjoner, der hver av disse definerer en delverdi av tilstanden...

- En verdi er en node i grafen. Atomiske verdier skrives med verdien sjøl og er bladnoder. Sammensatte verdier skrives med " {}" for mengder og " []" for tupler. Sammensatte verdier har en utgående kant for hver delverdi de inneholder.
- Det finnes to slags kanter. En for verdier som er aggregert (hel linje), og en for verdier som er assosiert (stiplet linje).

FIGUR 6 Eksempel på objekt graf



2.3.2 Assosiering versus aggregering

Aggregering innebærer avhengighet. Et aggregert objekt kan ikke eksistere uten det objektet det er aggregert (eller inneholdt) i. Hvis et aggregat-objekt slettes, vil alle aggregerte objekter transitivt bli slettet. For eksempel hvis vi i figur 6 sletter banken i_1, vil alle kontoene og personene også bli slettet. Vi kan si at aggregeringsrelasjonen innebærer eksistens-avhengighet. Et objekt kan ikke bli aggregert i flere objekt samtidig.

Assosiasjon er kun referanser og de assosierte objekter eksisterer uavhengig av de objekter som har assosiasjoner til dem. Objekter kan gjennom assosiasjon også deles mellom flere. Hvis en da i figur 6 f.eks. sletter konto i_2, vil dette ikke ha noen betydning for person i_4.

2. 4. Persistens

Begrepet persistens (betyr varighet) angår objekters levetid. Man kan definere et objekts persistens som den tida det varer, altså et mål på tid. Men vi kan også spørre oss hva skal til for at vi skal kalle et objekt for varig (eller persistent). Litteraturen anvender ofte begrepet på denne måten at den skiller mellom varige og ikke-varige data. I [KhVa90] finner vi følgende utsagn om persistens:

> *"The intended meaning is that an object will persist after the termination of the program that manipulates it."*

Og slik kan vi dermed definere skillet mellom persistente og transiente objekter:

1. Et objekt som eksisterer (og er er tilgjengelig) ut over ei enkelt aktivisering av det programmet det blir manipulert av, er persistent.
2. Et objekt som bare eksisterer innenfor en enkelt aktivisering, er transient.

Persistens er en egenskap ved objekt-instanser og ikke ved typer. Alle typer kan tillates persistens. Alle typer har samme rett til persistens og så lenge et objekt varer, gjør dens type også det [AtBu87]. Dette er hva det velkjente prinsippet om 'ortogonal persistens' dreier seg om: Varighet er ortogonal til objektets type!

2.4.1 Persistens gjennom nåbarhet

Det har etter hvert blitt vanlig at konseptuelle objekt-modeller definerer hvilke objekter som skal oppfattes som persistente ut fra prinsippet om nåbarhet fra en eller annen database-rot. Alle objekter som er nåbare fra databaserota er persistente. De andre objektene er transiente.

I vår kontekst skal vi definere nåbarhetspersistens slik (lett omskriving fra [Elia93a]):

> *Ethvert objekt som er nåbart fra ei persistensrot gjennom navigering eller assosiative spørringer er persistent.*
>
> *Navigasjon er i essens å evaluere ekstraktør-funksjoner som returnerer objekter. Spørringer er avledete funksjoner, fra enklere (navigerende) funksjoner, som f.eks. returnerer en mengde objekter ut fra visse kriterier.*

Ut fra definisjonen av persistens ovenfor skulle dette bety at utfører en den samme sekvens av ekstraktør-operasjoner i to ulike program-aktiviseringer, med utgangspunkt i et felles kjent rot-objekt, skal man være garantert å få returnert referanse det samme objektet (forutsatt at det ikke har opphørt å eksistere i mellomtida).

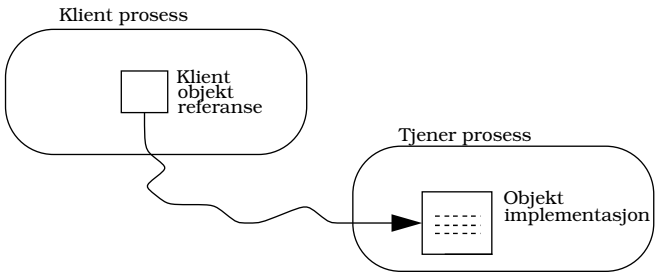
2.4.2 Aktiviseringer av objekter

Et objekt er aktivt når det befinner seg i internlager og er i en slik tilstand at det kan motta og utføre operasjons anrop[8]. Et objekt som bare befinner seg i eksternlager er ikke i stand til å utføre operasjoner direkte, men må aktiviseres først. Vi kaller det da passivt. Aktivisering av et persistent objekt innebærer da å lese data inn i internlager[9] og gjøre det tilgjengelig for bruk. Aktivisering av objekter forutsetter (og henger gjerne direkte sammen med) at en prosess startes opp (minne og prosessortid for eksekvering av programtråd allokeres).

Men i en (åpen) distribuert omgivelse som ANSAware, befinner objektenes implementasjon seg som regel ikke i samme adresserom (prosess) som programtråden(e) som bruker dem og bruker (klient) kan ikke gjøre noen forutsetninger om hvordan objektet er implementert.

Figuren nedenfor illustrerer hva som er typisk for (åpne) distribuerte applikasjoner. Man har et tjener program som implementerer objektet og et klient program som har en eller annen form for referanse til dette objektet (den informasjon som er nødvendig for å kunne finne fram til objektets lokasjon ved operasjonsanrop).

FIGUR 7. Klient og tjener aktiviseringer



Vi kan si aktivisering skjer på to nivå: I tjener-prosess der objektets implementasjon gjøres tilgjengelig for anrop utenfra og i klient-prosess som gjøres i stand til å anrope operasjoner og hvor referansen til objektet skaffes til veie. Tjener-objekt kan godt være

8. Dette er en vidt akseptert definisjon. Se f.eks. RM-ODP.
9. Persistens slik vi har definert det forutsetter at data kan lagres i persistent lager.

aktivt uten at det er noen klient-aktiviseringer og det kan godt eksistere flere klient-aktiviseringer på samme tid for samme tjener-objekt.

Dette skarpe skillet mellom klient- og tjenerobjekt åpner for en todelt tolkning av utsagnet fra et avsnitt tidligere (fra [KhVa90]) som var slik:

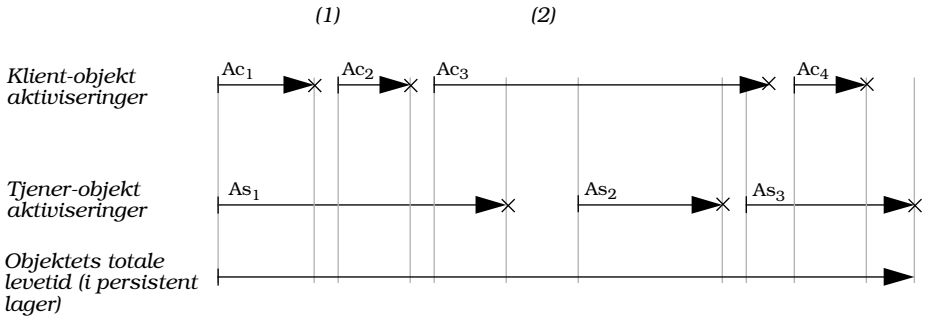
> *"... an object will persist after the termination of the program that manipulates it."*

Vi kan altså omskrive det slik for henholdsvis klient og tjener-objekt:

1. "*... an object will persist after the termination of the client program that use it.*"
2. "*... an object will persist after the termination of the server program that implements it."*

Figuren nedenfor skulle illustrere betydninga av dette skillet og hvor viktig det er å ta dette i betraktning når man skal definere hva persistens betyr.

FIGUR 8. Eksempel på tidsdiagram for aktiviseringer



Vi kan for det første (1) se et eksempel på flere etterfølgende klient-program aktiviseringer (Ac_1 og Ac_2) som begge aksesserer samme tjener-objekt aktivisering (As_1). Persistens i den første betydninga betyr at objektet overlever ulike klient-aktiviseringer.

For det andre (2) kan vi se eksempel på flere etterfølgende tjener-objekt aktiviseringer (As_1, As_2 og As_3), hvor en og samme klient (Ac_3) er aktiv. For denne klienten vil det ha stor betydning at objektet overlever flere tjener-aktiviseringer. Klienten bør altså ha inntrykk av å aksessere samme objekt, sjøl om tjener-aktiviseringa terminerer og etterfølges av en ny.

2. 5. Permanent versus temporær identitet

De lokale informasjonssystemer kan tilby ulike former for objekt-identitet til det globale objektrom. I denne sammenhengen er det relevant å skille mellom to former for identitet, nemlig permanent og temporær identitet[10].

2.5.1 Permanent identitet

Vi har permanent identitet når det lokale systemet tilbyr objekt-identifikasjon som er gyldig, og som ikke endres i løpet av det korresponderende objektets totale levetid. Slik identitet skal altså gjelde over flere aktiviseringer. Det er dette som i [ElKa91a] kalles immutabel identitet.

10. I litteraturen opereres det gjerne med tre former for identitet. Det jeg ikke har nevnt her er det som i [HaeDitt93] kalles imaginær identitet, eller det som i [ElKa91a] kalles verdi-basert identitet.

Permanent identitet er nødvendig når identifikatorer for objekter kan lagres ut over flere aktiviseringer av klient-programmet. Sett i forhold til figur 8 kan en si at samme identifikator kan brukes for samme objekt i aktiviseringene Ac_1, Ac_2, Ac_3 og Ac_4. Dette setter da også de krav til applikasjonen at identifikasjonen både overlever flere klient-aktiviseringer og at de overlever flere tjener-aktiviseringer. I denne sammenhengen er det da nødvendig med permanent identitet for persistensrøtter (eller "entry-points"). Andre objekter kan regnes som persistente, sjøl om systemet ikke oppfyller kravene til permanent oid, hvis de er nåbare fra ei rot. (jfr. nåbarhetspersistens avsnitt 2.4.1).

2.5.2 Temporær identitet

Vi har temporær identitet når det lokale systemet tilbyr objekt-identifikasjon som er gyldig, og som ikke endres, i løpet av ei aktivisering (eller klient-sesjon), men som derimot ikke er gyldig ut over ei slik aktivisering. Det er dette som i [ElKa91a] kalles for sesjons-identitet. Identiteten til et objekt kan skifte fra en sesjon til en annen, men innenfor en sesjon er man sikker på at en identifikator referer til et og samme objekt. Sett i forhold til figur 8 betyr dette at et objekt kan være representert med ulike identifikatorer i hver av aktiviseringene Ac_1, Ac_2, Ac_3 og Ac_4.

Globale objekt-identifikatorer som opprettes dynamisk som resultat av operasjonsanrop trenger ikke være permanente. Vi kan derimot oppfatte dem som temporære. Så lenge vi har nåbarhets-persistens, vil vi kunne nå det samme objektet i ulike sesjoner, gitt at vi tar utgangspunkt i oid for samme rot-objekt og utfører den samme sekvens av operasjoner (eller en ekvivalent sekvens av operasjoner). Man kan som en hovedregel si at objekter som opprettes som resultat av operasjonsanrop, har temporær identitet, mens persistens-rot objekter har permanent identitet.

2.5.3 Tjener-aktiviseringer og temporær identitet

La oss gå tilbake til figur 8 igjen og se på klient-aktiviseringa Ac_3. I løpet av den aktiviseringa har systemet to tjener-aktiviseringer for objektet (As_1 og As_2). Vi har sagt at permanent identitet skal overleve både klient- og tjener-aktiviseringer. I prinsippet er det slik at temporær identitet skal være gyldig så lenge klient-aktiviseringa varer. Dette betyr at temporær identitet må kunne overleve flere tjener-aktiviseringer.

Av hensyn til effektivitet og kompleksitet skal vi i praksis lette på dette kravet og si at temporær identitet hverken overlever klient- eller tjener-aktiviseringer. Hvis en tjener terminerer under en klient-sesjon, vil da dette bli oppfattet som en feil og klienten vil også terminere. Det vil som regel være dette som vil være tilfellet i praksis, nemlig at terminering av tjener-objekter ikke normalt skjer mens en klient anvender det. På denne måte trenger det lokale systemet bare å tilby permanent identitet (med den kompleksitet det medfører) for noen få objekter, nemlig de som skal være persistensrøtter.

2. 6. FRIL arkiktekturen

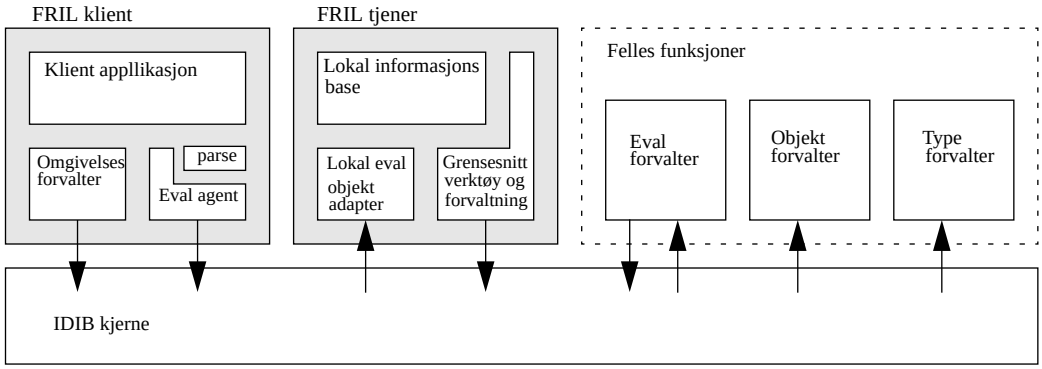
IIS-prosjektets mål er å utvikle en generisk infrastruktur som understøtter samvirke (interoperabilitet) og integrasjon av heterogene informasjonsbaser. Dette omfatter tjenester og verktøy og man tar sikte på å spesifisere og prototype disse. Infrastrukturen vil fungere som et rammeverk for resonnering og som en testbenk for funksjoner og verktøy som skal realiseres. I infrastrukturen ligger også realisering av datamodell transparens.

Prosjektet har valgt en språklig tilnærming [Elia89]. FRIL (Functional Resource Integration Language) består av to språk-komponenter: FIOL (Functional Inter Operation Language) er et funksjonelt språk. Det brukes for å spesifisere globale beregninger som kombinasjon av funksjoner. SESSL (Server Export Schema Specification Language)

brukes for å spesifi sær eksport skjema for lokale informasjonsbaser. Grensesnitt til objekter beskrives her som abstrakte datatyper (ADT).

En arkitektur for programmeringssystemet FRIL er skissert. Denne er hovedsaklig inspirert av ANSAware og CORBA. Figuren nedenfor illusterer arkitekturen. Her representerer modulen "Felles funksjoner", ei logisk samling av funksjoner og grensesnitt som er felles for alle brukere av infrastrukturen og kan betraktes som en del av denne. Arkitekturen er ment å være så generell at den kan instansieres og konfi gurres på mange ulike måter. Felles funksjoner som eval-forvalter vil f.eks. kunne realiseres som en distribuert tjeneste.

FIGUR 9. Skisse av arktitektur for FRIL



Av de logiske modulene som inngår i denne arkitekturen har vi for det første IDIB Kjerne som hovedsaklig understøtter interaksjon mellom de logiske komponentene i arkitekturen. Dette vil i essens være en protokoll for fjern evaluering av (FRIL)-programmer. Denne vil kunne kombineres med en transaksjonsprotokoll. IDIB kjerna er tenkt realisert ved hjelp av ANSAware og grensesnitt til denne vil typisk være ANSAware-grensesnitt.

For det andre har vi FRIL klienten som genererer FRIL programmer for utførelse av eval-forvalter. FRIL klient importerer typespesifi kasjoner og referanser til persistens-rot objekter (entry-punkter til informasjon) fra henholdsvis typeforvalter og objektforvalter. De logiske komponenter som utgjør FRIL-klienter er:

- Klient-applikasjon - En entitet som bruker FRIL som verktøy for å aksessere objekter. Her realiseres (delvis) integrasjon av ulike eksport skjema.
- Omgivelses-forvalter - En modul som vedlikeholder programmeringsomgivelsen til en FRIL programmerer. Omgivelsen består av et sett med eksterne abstrakte typer (importert fra type forvalter) og et sett med objekt-referanser (importert fra objekt-forvalter). I tillegg vil den kunne inneholde temporære objekt-referanser og lokale defi nisjoner av funksjoner og verdier.
- Parser - Typesjekker og oversetter FIOL-programmer til en intern form egnet for effektiv interpretering.
- Eval. agent - Modul som på vegne av klient styrer evaluering av FIOL-programmer (på intern form), ved hjelp av eval forvalter.

For det tredje har vi FRIL-tjenere som har i oppgave å realisere en abstraksjon av en informasjonsbase, slik at denne kan inngå i FRIL-omgivelsen. De logiske komponenter som utgjør FRIL-tjenere er:

- Lokal informasjonsbase - Entitet som representerer en eksisterende informasjonsbase og kan for eksempel være en DBMS eller en applikasjon der semantikken er fullstendig innbakt i programkoden til applikasjonen.

- Lokal eval og objekt-adapter - Modul som oversetter mellom den kanoniske datamodellen og tilsvarende for den lokale informasjonsbasen. Den vil kunne utføre FIOL programmer som den får fra eval-forvalter ved hjelp av kall til den lokale informasjonsbasen. En viktig del av dette er avbildning mellom global objekt-identitet og lokal form for identifikasjon av objekter (objekt-adapter). Denne modulen kan oppfattes som en transformasjonsprosessor (jfr. avsnitt 2.1.1) som oversetter mellom et kanonisk språk (FRIL) og det lokale språk.
- Grensesnitt-verktøy og forvalter - Verktøy for å konstruere grensesnitt-adapter for den aktuelle lokale informasjonsbasen, samt til å eksportere skjema og persistensrot objekter til føderasjonen.

For det fjerde har vi felles funksjoner for å understøtte distribuert evaluering, samt eksport og import av skjema og objekter:

- Eval-forvalter - Distribuert evaluering av FIOL-programmer.
- Objekt-forvalter - FRIL-tjenere bruker denne for å registrere objekter som klienter siden kan bruke for å få aksess til disse objektene. En tjener vil eksportere minst et slikt objekt og klienter skal kunne bruke dette til å få (temporær) aksess til andre objekter, gjennom funksjonsevaluering. Objekt-forvalter representerer en form for trading (jfr. avsnitt 2.7.1).
- Type-forvalter - Her registreres eksport skjema. Klienter kan siden importere disse.

2. 7. ANSAware

Denne avhandlinga handler om samvirke med ANSAware applikasjoner spesielt, og vi vil i den videre framstillinga forutsette en grunnleggende kjennskap til ANSA/ANSAware etter hvert som vi går dypere inn i materien. Derfor skal vi i dette avsnittet gi en mer grundig presentasjon av de mest sentrale konsepter og verktøy som vi finner i ANSA-arkitekturen og ANSAware. Vi bygger stort sett på ANSAware 4.0 Application Programmers Manual. [AW-APM] Kapittel 2.

2.7.1 ANSA arkitekturen

ANSAware tilbyr en implementasjon av en del av de konseptene som er definert i ANSA-arkitekturen. En arkitektur er i ikke et design av et bestemt produkt, men et sett med hjelpemidler for å gjøre design. ANSA definerer (ifl g. [AR.000.00]) en arkitektur til å bestå av følgende:

- Et sett av komponenter, som utgjør de basale byggeblokker og verktøy i arkitekturen.
- Set sett av regler, som styrer hvordan disse komponentene kan kombineres på en måte som er konform med arkitekturen.
- Et sett av fremgangsmåter som gir råd for hvordan basale komponenter kan kombineres ved hjelp av verktøyene, for å lage subsystemer med bestemte egenskaper.
- En sett av retningslinjer, som hjelper designere å gjøre valg, hvis deres egne preferanser ikke utgjør tilstrekkelig basis for slike valg.

"Viewpoints"

Under utvikling av arkitekturen, som i helhet er svært kompleks, studerte ANSA-teamet gjeldende praksis og forskning i distribuert databehandling og system-design teknikker. Det var tydelig at eksperter på distribuert databehandling har ulike syn på hva som er de sentrale problemstillinger innenfor domenet. De fokuserer på ulike aspekter ved distribuert databehandling.

Det viste seg at 5 synspunkt var dominerende. Både ANSA-arkitekturen og RM-ODP baserer seg på disse synspunktene. Disse kalles henholdsvis for 'Enterprise', 'Information', 'Computation', 'Engineering' og 'Technology'. Beskrivelsen av ANSA er strukturert som et sett av projeksjoner av arkitekturen til modeller som representerer hvert av disse 5 synspunktene. De 5 modellene er som følger:

- En "enterprise"-modell for å uttrykke system-grenser, policies og hensikt (Eller man kan si hvilken rolle informasjonssystemet har i organisasjonen). Eksempel på begreper her er agenter, ressurser, roller, administratorer, føderasjoner mm...
- En "information"-modell for å uttrykke semantikken til informasjon og semantikken til informasjons-prosesserings aktiviteter innenfor et system.
- En "computational"-modell for å uttrykke funksjonell dekomponering i distribuerbare enheter. Eksempel på begreper her er objekter, grensesnitt, operasjoner, binding, operasjonsanrop mm...
- En "engineering"-modell for å beskrive komponenter og strukturer som trenges for å understøtte distribusjon. Eksempel på begreper her er aktivisering, passivisering, migrering, protokoller, stubs mm...,
- En "technology"-modell for å beskrive valg av teknologi for å realisere et system. Her er fokus på implementasjon.

2.7.2 Beregningsmodellen

ANSAware er en implementasjon av "engineering" modellen, som igjen er et idealisert design av en støtteomgivelse for beregnings- ("computational") modellen. Vi skal her forklare konsepter i ANSA beregningsmodell som er direkte relatert til tilsvarende konsepter i ANSAware, før vi går inn på enkelte av "engineering"-konseptene.

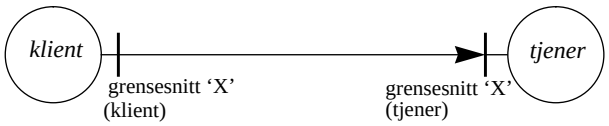
Tjenester, grensesnitt og objekter

En tjeneste er en informasjons-behandlings funksjon. Informasjonsbehandling kan være prosessering, lagring og overføring. En tjeneste en basal byggeblokk i ANSAware. I en åpen distribuert omgivelse vil vi ha komponenter som bruker tjenester og komponenter som tilbyr tjenester. Disse kalles henholdsvis klienter og tjenere.

En tjeneste tilbys ved hjelp av et grensesnitt, som er enheten for tjeneste-tilbud i ANSAware. Et grensesnitt er spesifisert som et sett av operasjoner som klienter kan anrope. En tjener kan tilby flere grensesnitt og en klient kan bruke flere grensesnitt på samme tid. Grensesnitt kan instansieres og det er fullt mulig å ha flere instanser av et grensesnitt på samme tid.

Figuren nedenfor illustrerer konfigurasjon av klienter, tjenere og grensesnitt. I denne grafiske notasjonen (fra RM-ODP) er grensesnitt synlig både på klient side (klient-grensesnitt) og tjener side (tjener-grensesnitt).

FIGUR 10. Notasjon



Et komponent eller objekt som er beskrevet alene ut fra måten den tilbyr eller bruker tjenester, kalles et beregnings objekt (computational object). Et beregningsobjekt har tilstand som manipuleres gjennom objektets grensesnitt (jfr. avsnitt 2.2.2). Et objekt kan ha flere grensesnitt på samme tid.

Fra ANSA-prosjektets rapport om beregningsmodellen [AR.001.01], finner vi bl.a. følgende definisjoner. Disse skulle oppsummere og klargjøre betydninga av de viktigste konseptene:

- Objekt: En enhet for program-modularitet som har tilstand og operasjoner for initialisering, aksessering og oppdatering av denne tilstand. Objekt-tilstand kan inneholde referanser til grensesnitt, både til objektet sjøl og til andre objekter.
- Grensesnitt: Et betegnelse av et objekt som en abstrakt tjeneste. Et grensesnitt er spesifisert som et sett av operasjoner, sammen med restriksjoner på bruk av operasjoner med hensyn til synkronisering og ordning.
- Grensesnitt-type: Et skjema for et grensesnitt som spesifiserer signaturer til operasjoner i grensesnitt av denne typen.

Transparenser

En transparens skjuler et bestemt aspekt ved kompleksiteten i distribuert programmering. I ANSA og i RM-ODP har man identifisert et sett med transparenser som en applikasjon kan tilby. Det er valgfritt for en applikasjon hvilke transparenser som skal gjelde til enhver tid. ANSAware implementerer følgende transparenser som er svært viktige i et åpent distribuert system:

- Aksess transparens: En uniform måte for objekter å interagere med hverandre, uavhengig av hvordan de er konstruert, hvilke omgivelse de befinner seg i, og uavhengig av om de er lokal eller fjern.
- Lokasjons transparens: Objekter kan interagere med hverandre uten kjennskap til fysisk lokasjon.

ANSA og RM-ODP har spesifisert en rekke andre mulige transparenser. Av disse kan vi nevne repliserings-transparens (skjuler flere instanser av et grenesnitt som skal behandles som ett), migrerings-transparens (skjuler flytting av objekter), ressurs-transparens (skjuler passivisering og reaktivisering) og transaksjons-transparens (skjuler samtidighetskontroll og støtte for atomisitet).

Arkitekturelle tjenester

Det er aktuelt for de fleste applikasjoner å bruke tjenester for å finne tjenester, for navngiving, for aksess-kontroll og for forvaltning. ANSA ser det derfor slik at det innenfor arkitekturen må finnes enkelte tjenester, som gir aksess til slik funksjonalitet. En sentral arkitekturell tjeneste er trading som går ut på at klienter kan finne fram til tjener-grensesnitt ut fra krav til hvilke egenskaper som er ønsket av grensesnittet. Trading-tjenesten tilbys av et objekt kalt trader.

Grensesnittreferanser

I beregningsmodellen kommuniserer objekter ved å utveksle grensesnittreferanser med hverandre. Dette er ganske enkelt entiteter som refererer til instanser av grensesnitt. En klient som har en grensesnittreferanse kan anrope operasjoner i grensesnittet. Grensesnittreferanser kan fritt utveksles mellom objekter.

2.7.3 Engineering modellen

Engineering modellen brukes til å beskrive hvordan distribusjons-transparensene kan realiseres ved hjelp av tjenester som er definert der. Modellen definerer en del konsepter for å beskrive slike transparens funksjoner, og vi skal her nevne det som er mest relevant for oss, nemlig begrepene node, kapsel, engineering-objekter og de viktigste transparens-tjenester.

Node

En node er (i følge RM-ODP) en enheten for lokalisering. En node har et sett av funksjoner for prosessering, lagring og kommunikasjon. Et typisk eksempel på hva en node kan være i praksis, er en enkelt datamaskin eller en arbeidsstasjon. Dette svarer til den vanlige oppfatninga av lokasjon i distribuerte system. En node kan også være andre ting. For eksempel er det naturlig å betegne et nettverk av datamaskiner som kjører med et distribuert operativsystem, som en node.

På en node finner vi også kjerne (eller "nucleus"). Ei kjerne er (ifl g. [RM-ODP.3]) et objekt som koordinerer prosessering, lagring og kommunikasjonsfunksjoner for bruk av andre "engineering" objekter innenfor en node. Vi kan altså si at begrepet kjerne er noe som tilbyr ressursforvaltnings-funksjoner på en node. Et typisk eksempel på kjerne er operativsystemet.

Kapsel

En kapsel er (ifl g. [RM-ODP.3]) enhet for innkapsling av prosessering og lager. Programmerere lager et sett av kommuniserende kapsler og hver av disse representerer et separat adresserom. Hver kapsel har tilgang til tjenester i kjerna. (eller man kan si at det finnes en instans av grensesnitt til kjerna i hver kapsel). I UNIX for eksempel svarer en kapsel ganske enkelt til en prosess. Objekter eksisterer innenfor kapsler og bruker kjerne-tjenestene for å interagere med hverandre.

Objekter

Ved implementasjon av tjenester, vil en programmerer definere et antall beregningsobjekter. Programmerer vil anvende et språk som representerer beregningsmodellens konsepter . Når slike beskrivelser av beregningsobjekter (som er potensielt distribuerbar) kompileres, vil beregnings-konsepter bli oversatt til engineering-konsepter. Operasjonsanrop vil bli oversatt til kall til den lokale kjerne. Et kompilert beregningsobjekt kalles et engineering-objekt.

Et engineering-objekt er den minste enhet i ANSAware som kan bli distribuert, aktivisert, passivisert og migrert. Et engineering-objekt kan i følge [AW-APM] komponeres av et eller flere beregningsobjekter, som er bundet sammen under kompilering og som interagerer internt via lokale prosedyrekall. Dette minner om RM-ODP begrepet cluster som der er enhet for aktivisering og passivisering (ressurs-allokering).

Flere engineering-objekter kan linkes sammen til en kapsel-template[11]. Programmerer sørger for å lokalisere de ulike objekter til kapsler ut fra hensyn til beskyttelse og feiltoleranse. Kapsel-templates tilsvarer program-filer i ANSAware under UNIX.

Transparens tjenester

Engineering objekter kommuniserer med hverandre gjennom kjerna. For å oppnå de ønskede transparenser er det nødvendig å ha transparens-tjenester som en del av hver kapsel. En transparens-tjeneste forvalter ressurser som tilbys fra kjerna i en kapsel og kommuniserer med tilsvarende entiteter i andre kapsler for å realisere den ønskede transparens.

Aksess-transparens tjenester skjuler forskjeller i representasjon av data og operasjoner. Dette realiseres ved hjelp av stubs for fjerne operasjoner og marshalling for å over-

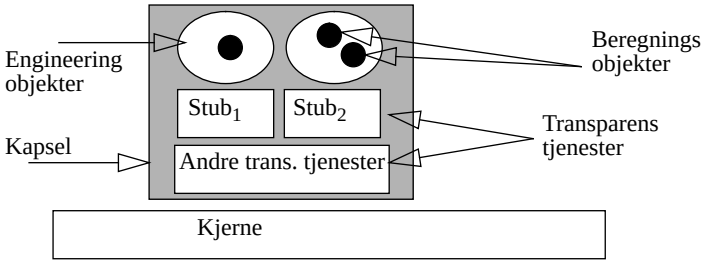
11. En "template" er en "mal" som definerer felles egenskaper for ei samling objekter. Vi kan ha templates for ulike komponenter i arkitekturen. F.eks. objekter, grensesnitt og kapsler. Objekter kan bli generert fra en template gjennom instansiering. (jfr. [RM-ODP.1])

sette mellom lokalt data-representasjons format og et format som lar seg kommunisere serielt over nettverket. Stub-funksjonene kaller marshalling-funksjoner for å omforme argumenter og resultater til operasjons-anrop.

Lokasjons-transparens tjenester utfører slike ting som avbildning mellom grensesnitt-referanser og nettverks-adresser.

Figuren nedenfor illustrerer forholdet mellom beregnings-objekter, engineering-objekter, kapsel og transparens-objekter i ANSAware.

FIGUR 11. Kapsel, objekter og transparens-tjenester



2.7.4 IDL og stub-kompilatorer

En grensesnitt-spesifikasjon definerer hvordan interaksjon med en bestemt tjeneste kan foregå, altså hvilke interaksjoner som kan foregå over et grensesnitt, og hvilke data som kan utveksles i disse interaksjonene. En kan si at i en slik spesifikasjon definerer en protokollen for interaksjon.

All interaksjon mellom ANSAware komponenter er basert på grensesnittspesifikasjoner for bestemte tjenester, og dette er også den eneste informasjonen som interaksjonene er basert på. Dette betyr at alle detaljer som angår interaksjoner basert på et bestemt grensesnitt, må kunne uttrykkes i et grensesnitt-spesifikasjons-språk, IDL (Interface Definition Language).

Slike grensesnitt-spesifikasjoner tilsvarer spesifikasjon av abstrakte datatyper, men man spesifiserer bare interaksjonsmåten. IDL representerer stort sett bare de syntaktiske aspekter ved grensesnittet, det vil si operasjonssignaturer og data-typer (jfr. abstrakt syntaks i OSI-presentasjonslag [HeSh90]). For spesifikasjon av grensesnitt i IDL kan en spesifisere noe semantisk informasjon. Slik informasjon faller innenfor to kategorier:

- Egenskaper ved operasjons-anrop. En kan spesifisere om interaksjonen er synkron i den forstand at klient blokkerer til svarmelding har kommet fra tjener, eller om den er asynkron. Dette angår valg av underliggende protokoller for å understøtte grensesnittet.
- Transparens-krav til interaksjoner[12].

Definisjon av grensesnitt

ANSAware tilbyr et grensesnitt-spesifikasjons språk (IDL) samt verktøy for å avbilde fra IDL til transparenstjenester og bruk av underliggende interaksjonsmekanismer. En grensesnitt-spesifikasjon i IDL består av følgende deler:

- Heading.
- Angivelse av relasjoner til andre grensesnitt.

12. Ikke uttrykkbart i ANSAware IDL foreløpig.

- Grensesnittets kropp som består av et sett av type-definisjoner og et sett av operasjonssignaturer.

FIGUR 12. Struktur til en grensesnitt-spesifikasjon

```
<TypeName> : INTERFACE =
<relasjon til andre grensesnitt>
BEGIN
    <grensesnitt-spesifikke type-definisjoner>
    <operasjons-signaturer>
END.
```

I heading definerers grensesnittets navn. Dette navnet kan brukes til å referere til grensesnittet i andre grensesnittspesifikasjoner eller det kan brukes i klient- og tjenerprogrammer ved deklarasjon av grensesnittreferanser. En grensesnittreferanse for gresesnitt med navn '*XXX*' vil i ANSAware være av en type med navn '*XXXRef*'. Vi kan si at en type blir implisitt definert i en grensesnittspesifikasjon, nemlig typen for referanser til grensesnittet.

Definisjon av verdi-typer

Første del av blokka mellom *BEGIN* og *END*, er et sett med verdi-type definisjoner. Det man gjør er egentlig å definere nye typer ved hjelp av typekonstruktører og navn på eksisterende typer. Man kan også oppfatte det som at man definerer alias for typer. For eksempel kan man definere en ny type *Ident* til å være et heltall slik:

```
Ident : TYPE = INTEGER;
```

Man har følgende predefinerte basale typer i ANSAware. Dette er atomiske typer som kan brukes direkte eller til å bygge mer komplekse sammensatte typer:

- *BOOLEAN*. Kan ha to verdier: *ansa_TRUE* og *ansa_FALSE*.
- Positive heltall: Vi har *SHORT CARDINAL* som har 16 bits presisjon, *CARDINAL* som har 32 bits presisjon og *LONG CARDINAL* som har ekstra presisjon (ikke standardisert antall bits)
- Heltall: Vi har *SHORT INTEGER* som har 16 bits presisjon, *INTEGER* som har 32 bits presisjon og *LONG INTEGER* som har ekstra presisjon (ikke standardisert antall bits).
- Flyttall: Vi har *REAL* som har 32 bits presisjon og *LONG REAL* som har 64 bits.
- *OCTET* (8 bits enhet) og *CHAR* (7 bit ASCII tegn).
- *STRING* som er en tekststreng med variabel lengde.

For definisjon av mer komplekse typer har vi følgende konstruktører:

- Oppramstyper.
- ARRAY (sekvens med fast lengde) og SEQUENCE (sekvens med variabel lengde).
- RECORD (gruppering eller tupler av elementer av ulike typer) eller CHOICE (uniontype der en verdi kan være av en av flere alternative typer).
- Grensesnittreferanse av en bestemt type.

Figuren nedenfor viser eksempler på bruk av disse konstruktørene.

FIGUR 13. Type-konstruktører i IDL

```
Color : TYPE = {Red, Blue, Green};     -- Eksempel på oppramstype
Address : TYPE = ARRAY 16 OF OCTET;    -- Eksempel på array
Vector : TYPE = SEQUENCE OF Real;      -- Eksempel på sekvens
HostEntry : TYPE = RECORD [            -- Eksempel på record
       H_Name : STRING,
       H_Type : INTEGER,
       H_Address : Address
];
Result : TYPE = CHOICE Color OF {      -- Eksempel på choice
       Red => INTEGER,
       Green => STRING,
       Blue => REAL
];
MyRef : INTERFACEREF OFTYPE IfTypeName; -- Grensesnittreferanse
```

Definisjon av operasjons-signaturer

Hver operasjon i et grensesnitt har et navn, ei argumentliste (som kan være tom) og ei resultatliste (som kan være tom). Argument- og resultatlister i operasjonssignaturer definerer hvor mange og hvilke typer verdier som kan inngå i et anrop og i ei terminering av et anrop. Argumenter må gis formelle parameternavn, mens for resultater er dette ikke nødvendig (valgfritt).

To typer operasjoner (med hensyn til anrops-semantikk) kan definerers: *ANNOUCEMENT OPERATION* som er asynkront (en venter ikke på noe svar) og *INTERROGATION OPERATION* som er synkront. Hvis en bare skriver *OPERATION* betegner dette en synkron operasjon.

FIGUR 14. Eksempel på operasjons-signatur

```
Sum : OPERATION [x, y : REAL] RETURNS [REAL];
```

Arv og inklusjon

To typer relasjoner til andre grensesnitt kan definerers: NEEDS som bare inkluderer navn på typer som er definert i det navngitte grensesnittet, IS COMPATIBLE WITH som også innebærer at operasjoner fra det navngitte grensesnittet arves. Dette er en måte å spesifisere subtyper i IDL (at et grensesnitt er kompatibelt med et annet). [13]

FIGUR 15. Eksempel på grensesnitt med inklusjon og arv fra andre grensesnitt

```
ExtendedBankAccount : INTERFACE =
NEEDS Bank;
IS COMPATIBLE WITH Account;
BEGIN
    TransferTo : OPERATION [amount: REAL; BankRef bank, AccountRef acc]
                 RETURNS [Status];
END.
```

Stub-kompilator

Grensesnitt-spesifikasjoner brukes til å generere stubs for den aktuelle språkbindinga, slik at applikasjonen kan utføre operasjonsanrop på en transparent måte. I essens genereres dette av ANSAwares stub-kompilator. Den genererer i hovedsak dette:

13. Det er også mulig å si “IMPLEMENTATION IS COMPATIBLE WITH”. Da antar man at også implementasjonen til operasjonene er arvet (se for øvrig [AW-APM] avsnitt. 3.2.4)

- Typedefinisjoner som korresponderer til typene som ble definert i IDL spesifikasjonen, til bruk i språk-binding. Dette inkluderer en type for grensesnittreferanser til det aktuelle grensesnittet. I tillegg kommer funksjons-deklarasjoner som korresponderer til operasjonene i grensesnittet.
- Implementasjon av klient-stubs og tjener-stubs.
- Implementasjon av marshalling funksjoner for klient og tjener.

Språkbindinger

For at applikasjonsprogrammerer skal kunne anvende ANSAwares muligheter, må konsepter representeres i det språk som brukes. I ANSAware skrives program-kode for å tilby og bruke tjenester ved hjelp av PREPC-setninger som er innbakt i C kode[14]. PREPC brukes til å uttrykke følgende:

- Deklarasjon av grensesnitt-typer og binding av variabler som representerer grensesnittreferanser.
- Dynamisk opprettelse og sletting av grensesnitt-instanser.
- Anrop av operasjoner i et eller flere grensesnitt-instanser.

Figuren nedenfor viser et eksempel på anrop av grensesnitt fra en klient (ANSA_Real er en C-type som svarer til IDL typen REAL:

FIGUR 16. Eksempel på anrop i PREPC

```
! USE Calculator;            /*  Deklarasjon av grensesnitt-type i PREPC  */
SummerRef Summer;            /*  Variabel for grensesnittreferanse i C  */
...                          /*  Vi antar at det her blir gjort binding til riktig -tjener grensesnitt */
{
   ansa_Real x, y, z;
   x = 3;
   y = 5;
!  {z}<- summer$Sum(x,y)     /*  Operasjonsanropet (resultat bindes til variabel z) */
   ...
}
```

14. Man tenker seg også bindinger til andre språk. F.eks. C++, eller at det blir utviklet et eget distribuert programmeringsspråk (DPL) hvor alle beregningsmodell-konseptene er representert.

Kapittel 3
Transparent interoperasjon med ANSAware

Aksess transparens er i følge [RM-ODP.3] å skjule forskjeller i data-representasjon og anropsmekanismer. Dette muliggjør interoperasjon over heterogene datamaskin-arkitekturer og programmeringsspråk. Dette kan sammenliknes med datamodell transparens i FDBS (jfr. avsnitt 2.1.1), men for distribuerte systemer vil dette i tillegg til aksesstransparens (slik RM-ODP definerer det) og tilhørende språkbinding til et data-manipuleringsspråk, kreve lokasjonstransparens, relokeringstransparens og eventuelt andre transparenser alt etter hvilke krav som settes for den aktuelle føderasjonen.

ANSAware tilbyr aksesstransparens ved hjelp av stubs, marshalling og språkbinding til C/PREPC. Denne avhandlinga ser i den forbindelse på alternative bindinger, til språk for anvendelse på et føderativt nivå, sammen med mekanismer for objekt-forvaltning (aktivisering og passivisering) og relokering[1].

I dette avsnittet fokuserer vi på persistens. Vi undersøker hvordan vi på en transparent og effektiv måte kan utnytte ANSAware applikasjoners iboende persistens. Det vil si at vi ønsker å gi inntrykk av persistente objekter på det globale nivå. Vi kan utdype problemstillinga i form av følgende spørsmål:

- Hva som kreves av ANSAware objekter for at vi skal kunne oppfatte dem som persistente. Dette gjelder både objekter som opptrer med permanent og med temporær identet (jfr. avsnitt 2. 5).
- Hvordan ANSAware applikasjoner svarer til prinsippet om nåbarhetspersistens (jfr. avsnitt 2.4.1). Vi spør her både hvorvidt vi kan oppfatte nåbare objekter som persistente og hvorvidt objekter kan bli persistente ved å asossiere dem med andre persistente objekter.
- Hvordan persistente ANSAware objekter identifiseres, både globalt og lokalt.
- Hvilke forvaltningsoppgaver som ligger i oppnåelse av transparens med hensyn til aktivisering, passivisering og migrering. Her kan vi spørre hvordan passivisering og migrering kan oppdages og hvordan systemet reaktiviserer (eventuelt finner ei eksisterende aktivisering. Det er relevant å spørre seg om kostnaden ved slik forvaltning og hvordan dette kan effektiviseres.

3. 1. Relaterte arbeider med persistens-transparens

En rekke arbeider fokuserer på infrastruktur støtte for persistens i (åpne) distribuerte omgivelser. Vi kan f.eks. nevne *Emerald* [Black87], *Arjuna* [Dixon89] eller *COMET* [MooVer92]. Vi skal kort presentere et par arbeider som baserer seg på ANSA-arkitekturen og forsøker å utvide den med støtte for persistens og migrering.

Zenith prosjektet ved universitetet i Lancaster [Blair92] arbeider med omgivelser for distribuert multimedia design. I forbindelse med realisering av sin objekt-modell, bygger de på ANSAware plattformen og adderer persistens- og migrerings-transparens. En

1. Vi forutsetter en omgivelse som tilbyr lokasjonstransparens, noe ANSAware gjør. ANSAware har også støtte for relokeringstransparens, noe det er opp til den enkelte applikasjon å benytte seg av.

objekt-identifikator vil her inneholde både et adressehint til et objekt, adresse til en ressursforvalter og adresse til en lokasjonsforvalter. Hvis et operasjonsanrop fra en klient feiler (den første adressen), vil ressursforvalter bli kontaktet (den andre adressen). Når objekter passiviseres, blir tilstand registrert i ressursforvalter. Denne har ansvar for reaktivisering og oppdatering av adressehint. Ved migrering registreres tilstand i en ressursforvalter på en annen node og adressen dit blir registrert i lokasjonsforvalter. Hvis anrop på en ressursforvalter feiler, vil lokasjonsforvalter bli kontaktet. Denne gir klienten adresse til den riktige ressursforvalter.

[Olsen92] beskriver en prototype av en infrastruktur for persistente objekter i heterogene distribuerte systemer. Denne understøtter transparent aktivisering/passivisering og migrering av objekter over ANSAware. Følgende komponenter blir beskrevet:

- En infrastruktur for såkalte "snapshots" det vil si maskinuavhengige representasjoner av objekter (tilstand og grensesnitt-instanser). Kode for å produsere og installere snapshots kan genereres automatisk ved hjelp av stub-kompilator og preprosessor.
- En snapshot database tjeneste som tilbyr et persistent lager for snapshots.
- En lokasjons-tjeneste som tilbyr en avbildning fra ugyldige til gyldige objekt-referanser. En objekt-referanse inneholder både grensesnittreferanse til objektet og referanse til en lokator tjener.
- En infrastruktur for transparent passivisering når et objekt ikke trengs og aktivisering når dets tjenester trengs. Aktivisering kan være "lazy" (gjøres etter at anrop feiler) eller "eager" (gjøres før anrop). Et problem som er nevnt er mulighet for "samtidige" aktiviseringer. Dette løses ved å sette lås på oppslag i lokator-tjenesten.
- En infrastruktur for migrering.

Infrastrukturer kan genereres automatisk ved hjelp av preprosessor. PREPC er utvidet med konstruksjoner for å spesifisere hva som er med i en snapshot, passivisering, aktivisering og migrering.

I RM-ODP skisseres noe liknende. Her passiviseres et objekt ved at det skrives et såkalt sjekkpunkt til stabilt lager. Deretter fjernes objektets cluster (representasjon av aktivt objekt). Et sjekkpunkt fungerer som en template for et cluster. Denne template er avledet fra tilstand og struktur til det bestemte clusteret og inneholder tilstrekkelig informasjon for å gjenopprette clusteret med samme tilstand. Aktivisering av objekt gjøres ved at det instansieres et cluster fra sjekkpunktet som tidligere har blitt skrevet til stabilt lager (dette kalles også "kloning" av cluster). En lokator tjener er også involvert i RM-ODP for at klienter skal kunne lokalisere nye aktiviseringer. En lokator kan også koples til aktiviseringstjeneste, det vil si at lokator kan starte aktivisering av et objekt når klient gjør oppslag (hvis det ikke er aktivt fra før).

3. 2. Persistente ANSAware objekter

Problemet her blir noe annet enn hva som er behandlet i avsnitt 3. 1. Vi skal se på transparent aksess til eksisterende ANSAware objekter. Utfordringa er å identifisere og utnytte iboende persistens i ulike applikasjoner. Her kan infrastruktur-støtte som nevnt i avsnitt 3. 1 bli anvendt i varierende grad, eller ikke i det hele tatt.

Brukere på det føderative nivået skal ha et bilde av objekter som oppfører seg i henhold til en global persistensmodell basert på nåbarhetspersistens (jfr. avsnitt 3. 3). Dette betyr at alle persistente objekter er nåbare fra et, eller flere rot-objekter. For rot-objekter må identifikasjon være permanent, det vil si den skal gjelde så lenge objektet eksisterer.

3.2.1 Permanente objekter

Vi skal først se på objekter som skal opptre med permanent identitet. Følgende krav må tilfredsstilles dersom objekter skal kunne opptre som permanente:

- Objekter må kunne identifiseres med en immutabel identifikator. Det vil si at objektets identifikator er den samme sjøl om ANSAware objektet passiviseres og aktiviseres igjen. Den må hele tida vise til samme informasjonsinnhold.
- Infrastrukturen (ANSA og/eller avbildnings-software) må være i stand til å reaktivisere ANSA-objekter som opptrer med permanent identitet hvis disse terminerer, og det må sørges for at de får samme tilstand som de hadde da de terminerte.
- ANSA-objektene må lagre informasjon på et stabilt medium hvor informasjonen overlever termineringer.

3.2.2 Temporære persistente objekter

For objekter som skal opptre med temporær identitet må følgende krav være tilfredsstilt, for at de skal kunne oppfattes som persistente:

- Operasjoner som returnerer grensesnittreferanse til et persistent objekt må aktivisere et eksisterende objekt, eller asossiere et nytt objekt med den persistente objektgrafen.
- Endringer som blir gjort på et objekt skal være synlig i andre (senere) aktiviseringer av objektet.
- Endringer lagres på stabilt medium, slik at de overlever terminering av tjener.

Vi skal si litt om dette i de neste avsnitt. Først skal vi se på operasjoner som returnerer grensesnittreferanser.

3. 3. Nåbarhetspersistens i ANSAware applikasjoner

I forbindelse med kravene beskrevet ovenfor, kan vi stille to spørsmål: Hvordan oppnår vi at et ANSAware objekt kan opptre som permanent, og hvordan avgjør vi om en gitt operasjon som returnerer en gitt grensesnitt-type, tilfredsstiller persistensmodellen. Vi har sett at ved integrasjon av en ANSAware-applikasjon kan vi stå overfor operasjoner som returnerer objekter, og vi har sett at det er mulig å gi et inntrykk av permanente ANSAware objekter. I så måte melder det seg noen spørsmål:

- Er alle nåbare objekter persistente, så lenge "ønt" objekt er det?
- Kan et objekt bli persistent gjennom ei beregning?

3.3.1 Komponent objekter

Svar på det første spørsmålet avhenger en del hva vi legger i begrepet 'nåbarhet'. Når et objekt opprettes (nytt) og returneres som resultat av en operasjon, kan en si det er nåbart gjennom denne operasjonen, men det kan i utgangspunktet være transient. Et nåbart objekt som er persistent, eksisterer fra før og har visse egenskaper. Operasjoner som returnerer objekter kan være en av følgende:

- Konstruktører: Operasjoner som oppretter nye objekter[2].
- Ekstraktører: Operasjoner som finner et eksisterende objekt og returnerer dette. Dette innebærer også aktivisering (se avsnitt 3. 4).

2. Vi forutsetter at en ren konstruktør, kun oppretter et objekt, uten å assosiere det med andre objekter. I praksis vil en ofte ha operasjoner som er kombinasjoner av konstruktør og mutator. Det vil si at samtidig som et objekt opprettes, assosieres dette med et annet objekt.

Nåbare objekter er persistente, dersom de returneres fra en ekstraktør-operasjon tilhørende et persistent "foreldre" objekt (jfr attributt-relasjoner i [ElKa91b]). For at en operasjon skal kunne kalles ekstraktør må den oppfylle kravene i avsnitt 3.2.2.

Et av kravene er at endringer på ekstraherte objekter skal være synlig i andre etterfølgende aktiviseringer. Hvis en operasjon returnerer et eksisterende objekt, og det ikke har denne egenskapen, kan vi oppfatte operasjonen som en konstruktør som oppretter et nytt transient objekt, som er en kopi av det egentlige objektet.

Ved avbildning av ANSAware skjema (IDL) til en global datamodell basert på nåbarhetspersistens, vil en del av jobben være å klassifisere operasjonene i konstruktører og ekstraktører, ut fra om kravene til persistens blir oppfylt. Dersom vi med nåbarhet bare mener nåbarhet gjennom ekstraktører (referanser til eksisterende objekter), kan vi si at nåbare objekter er persistente. Et IDL-skjema gir ikke tilstrekkelig informasjon om semantikken til operasjoner, så vi kan ikke klassifisere operasjoner ut fra slike skjema alene. Vi må forsøke å analysere hva applikasjonen gjør og trekke ut nødvendig informasjon.

3.3.2 Eksempel

Som eksempel på applikasjon med persistente objekter, har vi et informasjonssystem for en bank med opplysninger om konti og kunder. Her er kunder og kontoer aggregert i bankdatabasen, og kontoer og kunder er asossiert med hverandre (konto eies av kunde). Systemet har tre objekt-typer som er representert med hver sin ANSAware grensesnitt-type: *Bank, Kunde* og *Konto*.

I figuren nedenfor (Bank-grensesnitt) returnerer alle operasjonene grensesnittreferanser til enten *Kunde-* eller *Konto*-objekter. Vi vet ut fra kunnskap om applikasjonen at to av disse er konstruktører (*ny_Konto* og *ny_Kunde*). De to andre returnerer objekter som allerede finnes (*finn_Konto* og *finn_Kunde*). Disse er ekstraktører.

FIGUR 17. Eksempel på Bank grensesnitt

```
Bank: INTERFACE =
NEEDS Konto;
BEGIN
    ny_Konto   : OPERATION [k: KundeRef]
                   RETURNS [KontoRef, KontoNr];
    ny_Kunde   : OPERATION [navn: STRING; adresse: STRING]
                   RETURNS [KundeRef];
    finn_Konto : OPERATION [KontoNr]
                   RETURNS [KontoRef];
    finn_Kunde : OPERATION [navn: STRING; adresse: STRING]
                   RETURNS [KundeRef];
END.
```

konstruktører

Ekstraktører

I *Konto*-grensesnittet nedenfor har vi en operasjon *eier* som returnerer et *Kunde*-objekt. Hvis vi tenker oss at konto-grensesnittet er implementert slik at *Kunde*-objekter som returneres her, ikke har den egenskapen at endringer på disse blir persistente (synlig i andre aktiviseringer), kan vi regne operasjonen som en konstruktør som returnerer en (transient) kopi av et persistent objekt.

FIGUR 18. Eksempel på Konto grensesnitt

```
Konto : INTERFACE =
BEGIN
    KontoNr : TYPE = LONG INTEGER;

    eier       : OPERATION [] RETURNS [KundeRef];
    saldo      : OPERATION [] RETURNS [Real];
    credit     : ...
    debit      : ...
END.
```

Konstruktør (eller ekstraktør ....)

Ekstraktør (av verdi)

3.3.3 Hvordan nye objekter kan bli persistente

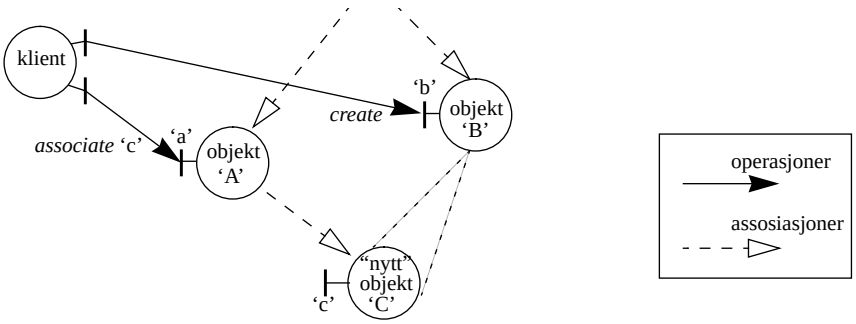
I følge persistensmodellen kan vi anta at alle objekter som ikke er nåbare fra ei persistensrot er transiente, og dersom et objekt ble assosiert med et persistent objekt, slik at det kunne nås fra dette, ville det bety at dette ble persistent. I objekt-orienterte databasesystemer som baserer seg på nåbarhetspersistens (jfr. f.eks. FAD [Banc87, KhVa90]) tenker man seg at et objekt blir gjort persistent (varig), ved å assosisere det med den persistente objekt-grafen[3].

I åpne distribuerte systemer som ANSAware er dette mer problematisk. Disse er bygd på autonome og heterogene komponenter. En komponent har ikke tilgang til og kan ikke gjøre forutsetninger om intern representasjon og implementasjon i en annen komponent. Hvis vi tar for oss ANSAware operasjoner som tar andre objekter som argument, kan dette i noen tilfeller være operasjoner som oppretter assosiasjoner mellom objekter. Det som er mest interessant her er objekter som i utgangspunktet ikke er persistente og som i følge persistensmodellen skulle bli det. I så fall må følgende gjelde:

- Hvis det "nye" objektet ikke allerede lagres på stabilt medum (disk) må assosiasjonen føre til at det blir det.
- Det objektet som det "nye" objektet blir assosiert til må ha en måte å reaktivisere dette, eller finne ei gyldig aktivisering. Objekts implementasjon må da enten kjenne det "nye" objektets interne representasjon og sjøl kunne reaktivisere dette fra disk (det innebærer at de er samlokalisert til samme kapsel), eller det må vite hvordan det ved hjelp av andre grensesnitt i systemet skal kunne reaktivisere det.

Figuren nedenfor illustrerer assosiasjon av objekter. Her opprettes et nytt objekt 'C' ved hjelp av grensesnitt 'b'. Dette assosieres deretter med objekt 'A' ved hjelp av en operasjon i dennes grensesnitt.

FIGUR 19. Illustrasjon av instansiering og assosiering



Så lenge vi kan anta at 'C' er aktivt, er det tilstrekkelig for 'A' å ta vare på grensesnittreferansen til 'c', men hvis 'C' passiviseres må 'A' ha mer informasjon. Problemstillinger her er hvordan 'A' kan vite at 'C' har blitt passivisert og hvordan 'A' kan reaktivisere 'C'.

Vi kan anta at ANSAware objekter kan ha operasjoner som lager assosiasjoner til, eller aggregerer andre objekter. Når ANSAware applikasjoner skal delta i føderasjoner, må en da vurdere semantikken til operasjoner og se om disse lager persistente asossiasjoner mellom objekter. Det er for øvrig verdt å merke seg at i arbeider med samvirkende informasjonssystemer som f.eks. ZOO_{IFI} [HaeDitt93], kreves det permanent objektidentitet av objekter som skal inngå i assosiasjoner som opprettes på globalt nivå. Tilsvarende problematikk skulle gjelde ANSAware-omgivelser.

I praksis vil det sannsynligvis være mest vanlig at persistent assosiasjon mellom objekter skjer innenfor samme kapsel og at instansiering og assosiering/aggregering skjer i

3. Dette gjelder i det minste på et konseptuelt nivå. I implementasjon er det vanlig at persistens til et objekt er bestemt ut fra om det bli plassert i et persistent eller temporært lager.

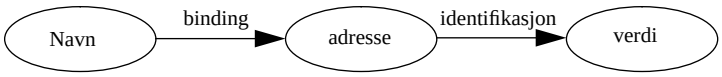
samme operasjon. Et nytt objekt assosieres med foreldre-objektet. La oss se litt på eksemplet i figur 17. Operasjonene *ny_Konto* og *ny_Kunde* oppretter nye objekter, men det vil i praksis ikke være naturlig at disse er transiente, men automatisk blir nåbare fra bank-grensesnittet og satt inn i den underliggende databasen.

Også her vil ei tilnærming være å klassifiser operasjoner i grensesnitt (se også avsnitt 3.3.1). Vi kan skille mellom de som er persistens-etablerende (oppretter en persistent assosiasjon), eventuelt persistens-terminerende (opphever en persistent asossiasjon), og de som ikke er det, ut fra hvorvidt kravene til persistente assosiasjoner blir oppfylt. For å gjøre slik klassifisering kreves kunnskap om semantikken til operasjonene.

3. 4. Transparens og objekt-identifikasjon

Identifikasjonsprosessen i tradisjonelle programmeringsspråk kan ifl g. [Renesse89] beskrives slik: Navn (det vil si variabelnavn i program-teksten) bindes til en adresse som identifiserer objektets verdi. Bindinga kan være statisk (gjøres av kompilator) eller dynamisk (gjøres i kjøretid). Figuren skulle illustrere dette.

FIGUR 20. Tradisjonelt syn på objekt-lokalisering



I en ANSAware-omgivelse blir prosessen litt mer komplisert, men kan forklares ut i fra det samme prinsippet. Grensesnittreferansen brukes til å identifisere grensesnitt. Dette innebærer både lokalisering av node, lokalisering av tjener-prosess, (ANSA-kapsel) og lokalisering av tilstand og operasjoner internt i kapselen (Grensesnittreferanser er altså bærer av adresseinformasjon). Denne identifikasjonsprosessen er transparent for brukere (lokasjonstransparens).

ANSAwares grensesnittreferanser er ikke uten videre en permanent form for identifikasjon. De overlever klient-aktiviseringer, men ikke tjener-aktiviseringer, siden hver grensesnittreferanse svarer til en bestemt tjener-objekt-aktivisering. Ny aktivisering innebærer at ny grensesnittreferanse blir generert[4]. Dermed må en simulere global permanent identitet over lokal identitet. Det vil si at ei avbildning er nødvendig mellom global objektreferanse og lokal identifikator som må oppdateres etter hvert som objekter aktiviseres og passiviseres. Utfordringa vil dermed være transparent bruk av aktiviserings- og passiviseringsmekanismer, samt å tilby en permanent global objekt-identitet. Endringer i lokal identifikasjon skal være skjult (transparent) for globale brukere. Sett i forhold til definisjonen av identitet i avsnitt 2. 5, vil grensesnittreferanser svare til temporær objekt-identitet (jfr. avsnitt 2.5.2). Grensesnittreferanser er en tilstrekkelig sterk identifikasjon, når vi i globale beregninger får dem returnert som resultat av operasjonsanrop og bare trenger dem innenfor en sesjon.

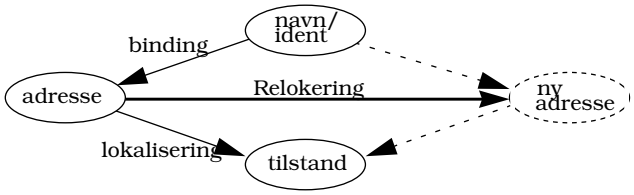
3.4.1 Objekt-identifikasjon på det globale nivå

Problemet her er at en adresse gjelder midlertidig. I ANSAware gjelder grensesnittreferansen så lenge et grensesnitt er aktivt. Vi vet at informasjonen som grensesnittet skal gi aksess til, kan vare lenger. Samme informasjon skal kunne være representert ved flere aktiviseringer gjennom sin levetid, og permanent objekt-identifikasjon på det globale nivå skal gjelde så lenge informasjonen eksisterer. Skal man i et klient-program ha et persistent bilde av informasjonen må vi sikre at bindinga mellom navn og informasjon er varig. Det gjøres gjennom relokering det vil si at avbildninga mellom navn og

4. ANSAwares “rot-objekt”, traderen er et unntak i så måte. Den er implementert slik at den oppretter en fast “hardkoda” grensesnittreferanse. Adresseinformasjon som internett-adresse, TCP/UDP port nummer vil være fast og definert på forhånd. Dette gir inntrykk av permanent identitet.

en adresse som ikke lenger er gyldig, skiftes ut. Dette forutsetter vi gjort transparent for brukeren. Figuren nedenfor illustrerer relokering.

FIGUR 21. Relokering

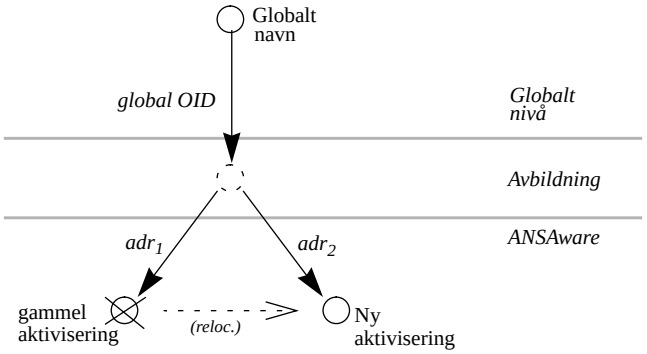


ANSAware understøtter relokering av grensesnittreferanser. Dersom et operasjonsanrop feiler, vil ANSAware automatisk forsøke å relokere med en ny grensesnittreferanse og forsøke anropet igjen. Grensesnittreferanser kan inneholde informasjon om eventuelle lokator-tjenere hvor man kan henvende seg for å få tak i nyere grensesnittreferanser (og eventuelt initiere reaktivisering), dersom den aktuelle grensesnittreferansen har blitt ugyldig.

Når ANSA-objekter skal gjøres tilgjengelig i en føderasjon, kan vi tenke oss et ekstra ledd i den identifikasjonsprosessen som jeg har nevnt ovenfor, nemlig at man på det globale nivå operer med en surrogat-basert-objektidentifikator (se f.eks. [KhosCop86] og [ElKa91a]). Denne spiller egentlig samme rollen overfor ANSA-objektene som variabelnavnet ovenfor. For permanente objekter er det nødvendig med en slik identifikator.

Det er avbildninga til den til enhver tid gjeldende grensesnittreferanse som må endres. Dette ser vi av figuren nedenfor. Global OID vil være den samme mens avbildning til grensesnittreferanser kan endres ved relokering.

FIGUR 22. Relokering og global OID



3.4.2 Identifikasjon av tilstand

Vi kan kalle objektets informasjonsinnhold for *tilstand* (se for øvrig avsnitt 2.3.1). Så lenge objektet er aktivt, tilbyr ANSA-infrastrukturen mekanismer for avbildning fra grensenitt til aktiv tilstand, men når objektet passiviseres og reaktiviseres, må vi ha en metode for å gjenopprette den aktive tilstanden.

For å gjenopprette tilstanden til objekter ved aktivisering, må en tjener få tak i informasjonen. Denne informasjonen kan enten bli gitt direkte til tjener, eller tjener kan få referanse til informasjonen, slik at denne kan finnes. Det vil egentlig være mange forskjellige måter å referere til passiv informasjon. Hvis vi forutsetter at man skal integrere eksisterende applikasjoner kan man ikke gjøre noen forutsetninger i så måte (I ANSAware-arkitekturen er ikke dette standardisert). Informasjonen kan for eksempel være:

- Tilstanden sjøl, på en implementasjonsuavhengig form.
- Filnavn (Et objekt lagrer tilstanden sin i ei navngitt fil).
- Indeks for post i fil (hvis man tenker seg at flere objekter av samme type deler ei fil).
- Relasjonsnavn i DBMS.
- Primær-nøkkel for post i DBMS.

En kan gjerne tenke seg kombinasjoner av slike referanser. For eksempel er det naturlig å kombinere filnavn og post-indeks i fil.

Tjenere kan i grove trekk få oppgitt slike tilstands-referanser på to måter. Den første er at den oppgis ved aktivisering av objekt. Enten som (kommandolinje-) argumenter ved oppstart av prosess (kapsel), som argument til operasjon som aktiviserer objekt. Den andre måten er at programmet finner automatisk fram til tilstanden. Det er ikke uvanlig å ha ei fast fil som inneholder nødvendig informasjon. Navnet på denne fila vil da være "hardkoda" i programmet. Denne metoden setter begrensinger på hvor mange instanser man kan ha av en type.

3.4.3 Klassifisering

For å oppsummere diskusjonen av objekt-identifikasjon så langt, kan vi klassifisere de nevnte aspekter ved identifikasjon av et objekt med hensyn til om de er permanent eller temporær og om de er global eller lokal. Figuren nedenfor illustrerer dette.

FIGUR 23. Klassifisering av objekt-identitet for et gitt (permanent) objekt

	Temporær	Permanent
Globalt (føderativt) nivå		Objekt identifikator
Lokalt nivå	Objekt-aktivisering (grensesnittref)	Objekt i persistent lager (disk)

3. 5. Transparens og forvaltning

Ønsket er altså at objekter framstår overfor føderasjoner som persistente og at underliggende passivisering, aktivisering er transparent for brukere. Migrering kan (sett fra føderasjonens side) oppfattes som passivisering og aktivisering på en annen lokasjon. Føderasjonen skal ikke styre passivisering og migrering, men skjule dette for brukere. Vi skal her se på hvilke forvaltningsoppgaver som ligger i oppnåelse av slik transparens. Dette involverer både bruk av mekanismer for detektering av passivisert objekt og lokalisering av nytt objekt (eventuelt initiering av aktivisering).

3.5.1 Hvordan oppdage passivisering og migrering

Objekter kan passiviseres som resultat av feil, eller for å frigjøre ressurser i perioder når objekter ikke anvendes. Vi skal ikke gå nærmere inn på hvordan og hvorfor objekter passiviseres, men stille spørsmål om hva systemet gjør for å oppdage at objekter er passivisert, slik at de kan reaktiviseres og relokeres når det er behov for det.

Når et objekt-passiviseres, vil alle dets klienter ha ugyldige referanser. Disse må relokeres for nye aktiviseringer. Et ønske er at reaktivisering og relokering, eventuelt spesiel feilhåndtering, skal kunne gjøres automatisk og mest mulig transparent for brukeren. Vi kan identifisere tre hovedstrategier for dette:

- Management by exception[5]. I utgangspunktet antar klienter at objektene er aktive og forsøker å utføre operasjonskall. Hvis objektet har blitt passivisert, feiler operasjonskallet. Feilhåndteringsrutinene i klient-kapsel vil forsøke å initiere nødvendig forvaltning for å finne nyere aktiviseringer eller aktivisere objektet igjen.
- Notifikasjon. Systemet varsler interesserte parter hvis tjenerobjekt passiviseres, slik at disse kan starte eventuell reaktivisering. Klienter som bruker notifikasjon oppretter et eget notifikasjonsgrensesnitt som gis til systemet.
- Monitorering, det vil si periodisk test om en tjeneste er i live. Hvert objekt kan testes med jevne mellomrom ved å anrope en operasjon i objektets grensesnitt. Hvis operasjonen feiler kan en anta at objektet er passivisert eller migrert. Alle ANSAware grensesnitt har en operasjon '*ping*' som kan brukes til dette formålet. Monitorering kan med fordel kombineres med notifikasjon.

"Management by exception" har den fordel at forvaltning bare initieres når absolutt nødvendig, det vil si når operasjonsanrop feiler. Ulempen er at hvis objekter er passive, er det en tidkrevende oppgave å oppdage dette. Feildeteksjon vil i verste fall være basert på at tjener ikke gir noe svar og det avgjøres med timeout. Notifikasjon har den fordel at klient får beskjed tidlig om at passivisering har skjedd, men ulempen er at tjener må registrere notifikasjonsgrensesnittet til hver enkelt klient. Dette skalerer dårlig med hensyn til antall klienter og antall objekter. Ved monitorering kan passivisering oppdages før operasjonsanrop, men det vil ikke alltid skje. Løsninga skalerer dårlig med hensyn til antall objektreferanser hos klienter, da det må gjøres anrop til hvert objekt man er interessert i.

3.5.2 Hvordan aktivisere eller finne aktivisering?

Når klientens infrastruktur vet at en grensesnittreferanse er ugyldig kan den forsøke å lokalisere en nyere aktivisering eller forsøke å få aktivisert objektet. Det er flere alternative måter å gjøre dette på:

- Klientens infrastruktur kontakter en lokator-tjeneste som har avbildning mellom gammel og ny grensesnittreferanse. ANSAware har støtte for dette, men det er opp til tjener-programmer å anvende dette.
- Klienten kontakter trader for å finne en eksisterende aktivisering som tilfredsstiller gitte krav til egenskaper. Det er opp til applikasjonen å spesifisere egenskaper og registrere dem i trader.
- Klienten kontakter en factory-tjeneste som instansierer eller aktiviserer et objekt. ANSAware tilbyr en factory tjeneste som kan starter opp tjener-kapsler. Factory har også støtte for monitorering av kapsler og notifikasjon til interesserte klienter hvis kapsler terminerer.
- En ANSAware-kapsel som aktiviseres av factory har et eget forvaltningsgrensesnitt. Dette har operasjoner for å aktivisere objekter innenfor kapsel, samt notifikasjon.
- Egne forvaltnings-objekter på høyere nivå. ANSAware tilbyr f.eks. en såkalt nodeforvalter, som anvender factory for å monitorere og administrere aktiviseringer av objekter på en node. Nodeforvalter tilbyr også navngiving av de forvaltede objekter.

5. [Blair92] kaller sin tilnærming til forvaltning for "management by exception".

3.5.3 Diskusjon

Forvaltning som er beskrevet her er ikke kostnadsfritt. Det vil si at oppdagelse av passivisering og reaktivisering tar tid, og lagring av nødvendig informasjon for forvaltning i klientens infrastruktur tar plass. Vi skal her kort diskutere enkelte aspekter med dette og se at er mulig å effektivisere forvaltning og avgrense antall forvaltede objekter.

Forvaltnings "policy"

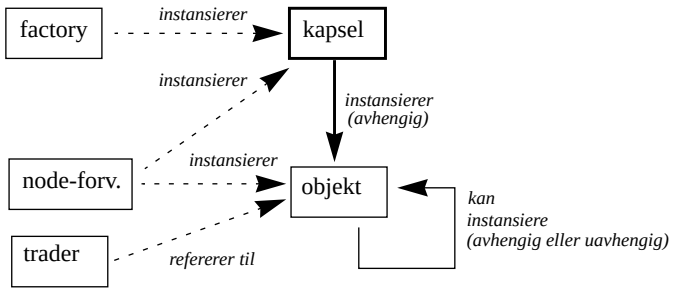
Policy for bruk av forvaltningsoperasjoner som er angitt over, kan (jfr. [Olsen92]) enten være "lat" (skjer først når klient anroper operasjoner på objekt) eller "ivrig" (skjer så tidlig som mulig). "Management by exception" forutsetter "lat" forvaltning, mens ved tidlig oppdagelse av passivisering, kan reaktivisering enten gjøres med en gang eller utsettes til anrop blir gjort. Tidlig oppdagelse er alltid en fordel, fordi man reduserer antall anrop som feiler og dermed sparer seg for den overhead det medfører. Sein forvaltning medfører overhead ved operasjonsanrop og det vil derfor kunne være en fordel om dette kunne gjøres tidligere. Problemet med tidlig forvaltning er at man ikke kan vite på forhånd når det er behov for et objekt. Dermed vil et objekt måtte være aktivt så lenge det kan bli behov for det. For noen applikasjoner kan dette være unødvendig lenge og binde opp ressurser i for stor grad.

Avhengighet

Et objekt kan være avhengig av et annet i den forstand at det ikke kan eksistere uten at det andre objektet eksisterer, eller det kan ikke være aktivt uten at det andre objektet er aktivt. Dette faktum kan utnyttes til å oppnå tidlig oppdagelse av passivisering, uten den kostnaden notifikasjon, monitorering eller "management by exception" medfører. Hvis et objekt blir passivt kan vi slutte at alle objekter som er avhengig av det også er passive. Kostnaden med denne tilnærminga blir først og fremst å statisk avgjøre avhengighetsforhold. Hvis vi ser på forvaltning innenfor ODP-rammeverket, kan for eksempel ikke et basic engineering objekt være aktivt uten at det tilhører et aktivt cluster. Et cluster kan ikke være aktivt uten at det tilhører en kapsel, etc..

I ANSAware omgivelsene er det relevant å se på begrepene grensesnitt, objekt, kapsel og node, og avhengighetene skulle være klare. Uten node kan ikke kapsel eksistere, uten kapsel kan ikke objekter eksistere, og uten objekt kan ikke grensesnitt eksistere. Det finnes egne grensesnitt med forvaltningstjenester for kapsler og objekter, så det er de som er mest relevant å ta med i diskusjon om forvaltning. Figuren nedenfor illustrerer forholdet mellom disse ulike typer objekter. Objekter kan instansieres av kapsel og det innebærer avhengighet[6]. Objekter kan instansieres av objekter og det kan enten gjøres avhengig eller uavhengig. Trader, node-forvalter og factory er spesielle objekter, som forvalter kapsler og objekter (Figuren er både en modell og en meta-modell).

FIGUR 24. Modell og meta-modell for instansiering



6. Alle objekter instansieres og er avhengig av en kapsel, men det er ikke alltid at dette skjer via en operasjon i kapsel-grensesnittet. Derfor kan det hende vi må spørre objekter om hvilken kapsel de tilhører.

Temporære objekter versus permanente objekter

Vi kan gjøre følgende avgrensing:

> *Bare objekter som fremstår med permanent identitet trenger å forvaltes slik vi har skissert det her. Hvis et temporært objekt passiviseres under en sesjon, antar vi at det er uttrykk for feil.*

For objekter med permanent identitet er slik forvaltning nødvendig (det er slik vi oppnår permanent identitet), mens for objekter med temporær identitet vil den tida en klient refererer til et slikt objekt, som også er den tida det trenger å være aktivt og tilgjengelig, vanligvis være kort.

Dermed avgrenser vi antallet objekter som behøver forvaltning til å være lite. Det vil som regel si et mindretall av de objekter en klient vil ha aksess til.

3. 6. Oppsummering

I dette kapitlet har vi behandlet transparent interoperasjon med ANSAware applikasjoner. Viktig her er aksesstransparens og persistens som forutsetter transparent forvaltning og transparent relokering,

Fokus har vært på persistens og vi har konstentrert oss om problemstillinger som angår transparent aksess til eksisterende ANSAware objekter. Utfordringa er å identifisere og utnytte applikasjoners iboende persistens. Her skiller vi mellom objekter som opptrer med permanent identitet og krav til disse, og objekter som opptrer med temporær identitet og krav til at disse skal være persistente. Vi søker å gi et inntrykk av nåbarhetspersistens, og her har vi sett på klassifisering av operasjoner i ANSAware-grensesnitt. Vi har identifisert ekstraktør-operasjoner, konstruktør-operasjoner og konstruktører som assosierer objekter med den persistente objektgrafen. Skal operasjoner oppfattes som ekstraktører eller assosierende på det globale nivå, må krav til persistens oppfylles.

Ulike aspekter ved objekt-identifikasjon ble diskutert. I dette ligger identifikasjon på globalt nivå, identifikasjon av objektets aktivisering og identifikasjon av passive objekters tilstand. Betydninga av relokering for å oppnå inntrykk av permanent identitet ble forklart.

Til slutt har vi diskutert betydninga av forvaltning. Problemstillinger her er effektiv oppdagelse av passivisering og metoder for reaktivisering (eller lokalisering av nyere aktiviseringer). Vi har kort diskutert hvordan kunnskap om avhengighet kan brukes til å effektivisere forvaltning, samt at vi har avgrenset forvaltning til å kun gjelde objekter med permanent identitet.

Kapittel 4
Objekt adapter for ANSAware applikasjoner

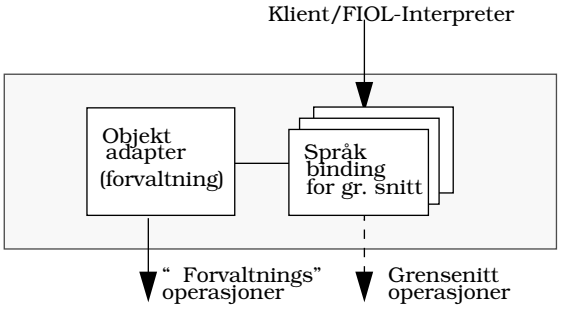
I dette kapitlet presenteres overordnede prinsipper for transformasjons-prosessor (se. avsnitt 2.1.1) mellom en føderativ omgivelse og et lokalt informasjonssystem som ANSAware. En slik modul skal oversette mellom kommandoer/operasjonsanrop som er i henhold til en global datamodell og de tilsvarende operasjoner i det lokale system, samt at den skal utføre nødvendig avbildning av objekt identitet og forvaltning av objekter (jfr. kapittel 3).

I forhold til FRIL-arkitekturen (jfr. avsnitt 2. 6) er fokus på det som omtales som "local eval and object adapter", og da først og fremst på objekt-adapter. Vi skal også se på binding til språk på det globale nivå, uten å gå inn på hvordan komplekse programmer evalueres i et språk som FRIL. Det vi konsentrerer oss om er enkel operasjonell avbildning av abstrakte operasjoner (jfr. [Bertino89]).

4. 1. Arkitektur

Ser vi nærmere på arkitektur for en slik transformasjonsprosessor, kan vi identifisere to hoved-komponenter. For det første har vi en objekt-adapter. Denne har ansvaret for å realisere transparens med hensyn til persistens og tilby sterk global identitet for objekter i lokale systemer. For det andre har vi språkbindinger (eller stub's), som har ansvaret for å realisere aksess-transparens for det språket klienten anvender.

FIGUR 25. Arkitektur for objekt-adapter



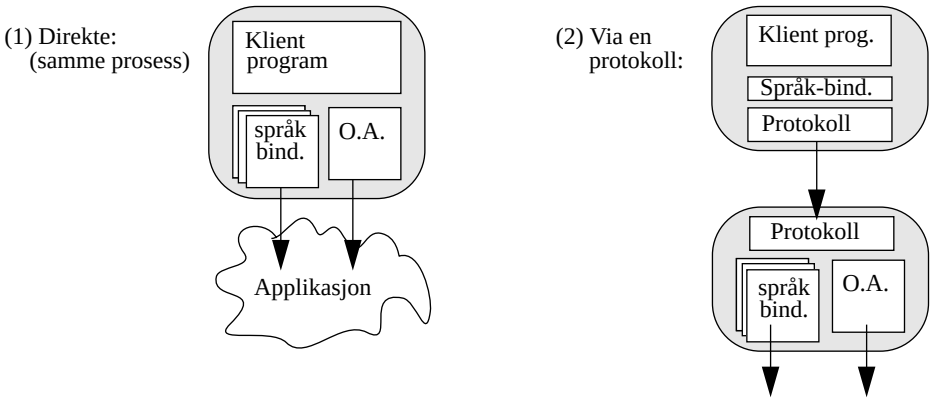
4.1.1 Klienter

En 'klient' er et program som er spesifikt for den applikasjonen som anvender objekter på det føderative nivå. Klienter kan bruke ulike programmeringsspråk og de kan interagere med transformasjonsprosessor-modul (objekt-adapter) på ulike måter:

- Direkte. Det vil si at klient-program (eller interpreter som kan utføre program) linkes sammen med kode for språkbinding og objekt-adapter og kjøres i samme prosess.
- Via en protokoll, det vil si at klient og transformasjonsprosessor er atskilte entiteter som kommuniserer ved hjelp av meldinger eller operasjoner (protokoll-elementer) som er i henhold til et sett regler som er kjent for begge parter. Sett fra transformasjonsprosessorens side kan protokollen i seg sjøl oppfattes som det språk[1] som

språkbindinga skal definerers for. Sett fra en klients side, kan en si at språkbinding består av to deler. En hos objekt-adapter (avbildning til protokoll-elementer) og en hos klient (lokal representasjon av objekt- og datatyper pluss avbildning til protokoll-elementer).

FIGUR 26. Konfigurasjons alternativer



4.1.2 Språk-binding

Språkbindinger er software-entiteter som ovenfor klienter tilbyr transparent aksess til objekt-adapter og de objekter som objekt-adapter formidler kontakt med, på en måte som er naturlig for det spesifikke språket. Språkbindinger må tilby følgende:

- Representasjon av referanser til objekter i det lokale systemet. For hver lokale objekt-type vil vi ha en assossiert objekt-referanse type i det aktuelle språket. (se avsnitt 4.4.1 for nærmere definisjon).
- Representasjon av operasjoner. For hver operasjon, i hver objekt-type, i det lokale systemet, vil vi ha noe tilsvarende (f.eks. en prosedyre) i det aktuelle språket.
- Representasjon av verdi-typer. For hver verdi-type som brukes i grensesnitt til det lokale systemet, vil det være definert en tilsvarende type i det aktuelle språket, pluss mekanismer for å avbilde.

Det er en fordel at språkbindings-komponentene skilles ut fra objektadapter slik, fordi vi oppnår at objekt-adapter kan beskrives uavhengig av hvordan klienter velger å interagere med systemet. Det er heller ikke noe i veien for at bindinger til ulike språk kan eksistere parallelt i forbindelse med en objekt-adapter.

Det skal kunne genereres språkbindinger for FRIL (og eventuelt andre språk som f.eks. C++). Dette involverer også omforming av argumenter og resultat-verdier og stub's som opptrer som lokale representanter for de "fjerne" operasjonene. Vår antakelse er at det her er mulig å generere all koden som trengs i språkbindinger fra IDL-spesifikasjoner ved hjelp av stub-kompilator (jfr. avsnitt 2.7.4).

Det vi her kaller språkbindinger svarer nært til det som i Thor-prosjektet [Liskov93, Bogle94] kalles "Veneers", det vil si software-"lag" som tilbyr program i ulike språk aksess til Thor-databaser. Aksess til Thor skjer via et språk-uavhengig grensesnitt til database-systemet. Det vi har sagt her er også i tråd med tilnærminga i [HaeDitt93] om deres integrasjons-software. Her sies det at softwaren kan deles opp i to deler: Den første delen har med innbaking av den globale datamodell og dens funksjonalitet i et bestemt programmeringsspråk. Denne softwaren kan genereres fra komponent-skjema. Den andre delen har med implementasjon av den globale datamodell over komponent-database-systemer, det vil si koplings-software.

1. En kan si vi avbilder via et "kanonisk" språk (se avsnitt 7.1.2 for ei nærmere utdyping)

4.1.3 Objekt-adapter funksjoner

Objekt-adapter-funksjonenes oppgaver er for det første å vedlikeholde referanser til objekter og assosiere dem med den informasjon som er nødvendig for at en klient skal kunne aksessere objektene. Klienter skal kunne identifisere hvert objekt gjennom slike referanser. Dette innebærer blant annet at objekt-adapter har disse oppgaver:

- Å generere nye objekt-referanser for objekter som blir kjent for objekt-adapter.
- Tolke objekt-referanser. For ANSAware vil det f.eks. si å skaffe klienten (språk-bindinga) en gyldig grensesnittreferanse for en gitt objekt-referanse.
- På vegne av objekt-implementasjonen assosiere den informasjon med objekt-referansen som er nødvendig for at objekt-adapter (på vegne av klient) skal kunne finne objektet, eller aktivisere det hvis det er nødvendig.
- Søppelsamle objekt-referanser (med tilhørende informasjon) som ikke lenger er i bruk.

For det andre skal objekters implementasjoner aktiviseres, der dette er nødvendig, og eventuelt passiviseres. Hvis objekt-adapter vedlikeholder informasjon som kan brukes for å gjenopprette tilstanden til et objekt og hvis objekt-adapter inneholder metoder for å gjøre dette, er den i stand til å gi et inntrykk av permanent objekt-identitet, for enkelte av objektene (jfr. avsnitt 2. 5).

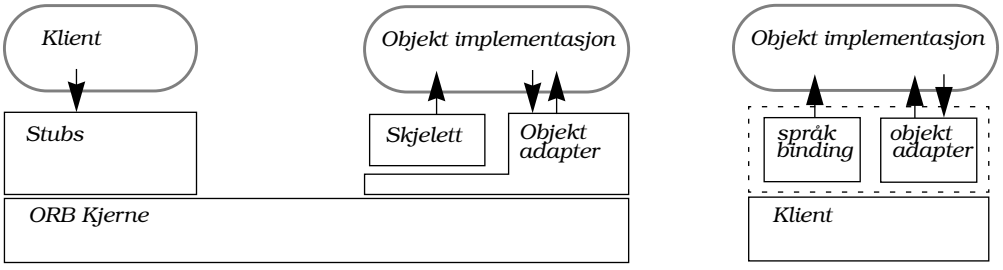
4.1.4 CORBA objekt-adapter

I CORBA, har objekt-adapter ansvaret for flere funksjoner. Slik defineres oppgavene til Basic Object Adapter [OMG.92.12.1]:

- Generering og tolkning av objekt-referanser.
- Autentisering av den som gjør anropet.
- Aktivisering og deaktivisering av implementasjon.
- Aktivisering og deaktivisering av individuelle objekter.
- Metode-anrop gjennom skjeletter.

Objekt-adapter er implisitt involvert i anrop av metoder, sjøl om det direkte grensesnittet er gjennom skjeletter. Slik kan en si at skjeletter i CORBA svarer til det vi har kalt for språkbindinger og at objekt-adapter svarer til de andre objekt-adapter funksjonene (vi ser bort fra autentisering her). Det er likevel en forskjell: Vi gjør ikke noen forutsetninger om hva som er klienter i forhold til objekt-adapter. I CORBA kommuniserer objekt-adapter alltid med klienter via ORB-kjerne. Skjelettene definerer dermed ikke binding til et bestemt språk, men til ORB-kjerna. Tilkoplet ORB-kjerna vil vi da også ha klienter, som har stub's og språkbindinger. I forhold til modellen som blir beskrevet her, kan CORBA oppfattes som et spesialtilfelle, der klient ikke interagerer direkte med objekt-adapter, men via en protokoll som er definert som en del av ORB-kjerne (se også figur 26, alternativ 2).

FIGUR 27. Sammenlikning med CORBA



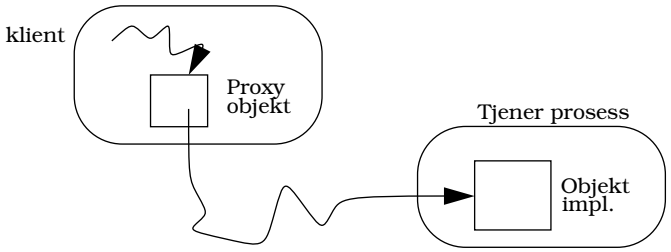
4. 2. Proxy-objektet

Vi skal her introdusere begrepet proxy-objekt. Hensikten er å kunne innkapsle informasjon om et objekt i et lokalt informasjonssystem i en logisk enhet. Dette omfatter identifikasjon av den til enhver tid gjeldende aktivisering, identifikasjon av passive objekter (for permanente objekter) samt tilhørende metoder for avbildning, aktivisering, passivisering og sammenlikning av objekter.

Begrepet proxy-objekt blir også introdusert i [ElKa91c] og hovedhensikten er å oppnå "representasjons-transparens". Det vil si at en har et uniformt format for globale objekt-identitet. Her identifiserer en oid et proxy-objekt og proxy-objektet representerer ei avbildning fra global til lokal representasjon. En proxy kan logisk sett oppfattes som en tuppel *(oid, e)* der *oid* står for global identitet og *e* står for et uttrykk som identifiserer et lokalt objekt.

En kan også si at et proxy-objekt er et objekt som overfor et klient-program opptrer på vegne av et annet objekt. Dette befinner seg gjerne i et annet adresserom (prosess) enn proxy-objektet. Proxy-objektet inneholder den nødvendige informasjon for at klient-programmet skal kunne lokalisere og interagere med objektets implementasjon (se figuren). Klienter vil referere til de egentlige objektene via proxy-objektene og må da kunne identifisere proxy-objektene. Sett fra klientenes side vil en objekt-referanse i praksis være en referanse til et proxy-objekt.

FIGUR 28. Proxy-objektet og objektets implementasjon



En alternativ tilnærmingsmåte er på det globale nivået å operere med OID'er som inneholder lokale-identifikatorer (jfr [Elia93b]). Dette innebærer heterogene OID'er. Denne heterogeniteten kan gjøres transparent for klient-programmer. Fordelen med denne tilnærminga er blant annet at en slipper å lagre avbildning fra global til lokal objekt-identitet (f.eks. som tabeller) i objekt-adapter. For applikasjoner hvor det er svært mange aktive objekter på samme tid kan dette bli et problem (se forøvrig avsnitt 8.1.1). I ANSAware må man lagre informasjon for permanente objekter, fordi lokal oid (grensesnittreferanser) kan bli ugyldig i løpet av objektets levetid (jfr. avsnitt 3. 4).

Det er altså tenkelig at proxy-objekter ikke alltid er representert ved fysiske objekter som lagres i objekt-adapter, men opptrer som rent konseptuelle konstruksjoner. I det følgende antar vi at alle proxy-objekter skal eksistere i objekt-adapter, som ei pragmatisk løsning.

4.2.1 Permanente versus temporære proxy-objekter

I diskusjon om objekt-identitet (i avsnitt 2. 5) går det fram hvor viktig det er å skille mellom permanent og temporær objekt-identitet. Vi ønsker å klassifisere proxy-objekter etter disse kriterier. Vi setter dermed et skille mellom permanente og temporære proxy-objekter. Dette vil ha sammenheng med hvordan proxy-objekter opprettes og slettes, hvordan objekt-implementasjonen aktiviseres og hva slags informasjon proxy-objektene har assosiert med seg.

Permanente proxy-objekter er i stand til å (re)aktivisere objekt-implementasjonen og har assosiert med seg den informasjonen som er nødvendig for å få til dette. Temporære proxy-objekter er ikke i stand til det. De vil være aktivisert for et allerede aktivt ANSAware objekt og vil ikke kunne brukes mer etter at ANSAware objektet er passivisert. De kan dermed søppelsamles.

4.2.2 Proxy-objektets oppgaver

Proxy-objektet vil ha følgende hoved-oppgaver i objekt-adapteren:

- Lokalisere objektets gjeldende aktivisering av implementasjon, slik at klienten kan utføre operasjoner mot det. Proxy-objektet må skaffe til veie informasjon om objektet, slik det er representert i det lokale systemet, for at klienter skal kunne utføre operasjoner mot objektet. Den lokale representasjon av objekter i ANSAware er som kjent grensesnittreferansen, og den er tilstrekkelig for å kunne utføre operasjoner. Alle ANSAware objekter identifiseres gjennom denne.
- Aktivisere (og eventuelt passivisere) objekters implementasjoner, der vi har med permanente proxy-objekter å gjøre. Aktivisering betyr altså at det blir mulig å utføre operasjoner mot objektet. Det omfatter aktivisering av objektet i det lokale systemet, og at proxy-objektet får en gyldig kopi av den lokale representasjonen (grensesnittreferansen) som nevnt i punktet ovenfor.

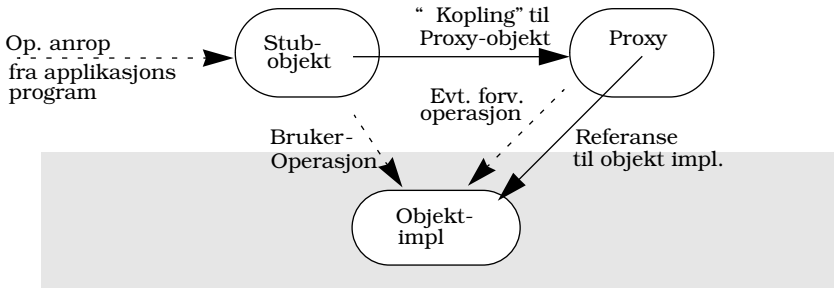
For å kunne aktivisere, må permanente proxy objekter ha tilgang på informasjon om hvordan objektet aktiviseres og de må ha informasjon som er nødvendig for å gi objekter riktig tilstand ved aktivisering. Altså en permanent form for identifikasjon av objektets tilstand (jfr. avsnitt 3.4.2).

4.2.3 Språk binding og proxy-objekt

Det følger av arkitekturen at språkbindinger ikke skal være en intern del av proxy-objektet. Vi ønsker at proxy-objekter skal kunne defineres og eksistere uavhengig av den språkbindinga som til enhver tid blir anvendt.

Språkbindinga vil måtte anvende proxy-objektet for å kunne utføre operasjonene. Vi har da to typer objekter som henger sammen og som begge er nødvendige. Vi har språkbindings-objekt (i ANSAware omgivelser er det naturlig å kalle det stub-objekt), og vi har proxy-objekt som overfor stub-objekter er representant for objektets implementasjon.

FIGUR 29. Proxy- og stub-objektenes rolle



4.2.4 Aktivisering av proxy-objekter

Det er to ulike måter et proxy-objekt kan bli opprettet (aktivisert) på:

1. Et ANSAware objekt returnerer grensesnittreferanser som resultat fra operasjonsanrop. Vi oppretter (aktiviserer) nye proxy-objekter for hver av disse. Slike objekter er typisk temporære.

2. Vi oppretter et proxy-objekt som ikke har noe tilhørende aktivt ANSAware objekt. Det er passivt i utgangspunktet. Proxy-objektet vil ha kjennskap til hvordan det tilhørende ANSAware objektet skal aktiviseres når det er behov for det. Slike objekter er permanente og kan dermed opptre som persistens-røtter.

Disse alternativene er ikke nødvendigvis gjensidig utelukkende (sjøl om det vil være det vanlige). En kan tenke seg kombinasjoner som dette:

3. Et ANSAware objekt returnerer en grensesnittreferanse som det opprettes proxy for. Denne sørger deretter for å hente inn nødvendig informasjon, slik at det tilhørende ANSAware-objekt kan passiviseres og aktiviseres igjen.
4. En proxy skal opprettes etter alternativ 1 og det viser seg at det allerede eksisterer en proxy for den aktuelle grensesnittreferansen, som er opprettet etter alternativ 2. Da kan den allerede eksisterende proxyen anvendes.

4.2.5 Sammenlikning av proxy-objekter

Det er nødvendig å kunne teste om to proxy-objekter representerer samme objekt, for å kunne eliminere duplikater av proxy-objekter. Objekt-adapter skal ha den egenskapen at to ulike proxy-objekter alltid representerer to ulike objekter i det lokale system. Når et nytt proxy-objekt skal opprettes og installeres, vil objekt-adapter måtte sjekke om det finnes et proxy-objekt fra før som representerer samme objekt. Dette involverer sammenlikning av proxy-objekter.

Aktive objekter

ANSAwares grensesnittreferanser kan sammenliknes for å avgjøre om de representerer samme grensesnitt-instans. ANSAwares infrastruktur tilbyr funksjoner for dette.

Passive objekter

Et forhold som kan komplisere sammenlikning av permanente objekter er at objektenes implementasjoner ikke nødvendigvis alltid er aktive. Passive objekter har ugyldige grensesnittreferanser og disse kan da ikke brukes til sammenlikning. Det er to måter å løse dette.

- Man kan velge å aktivisere objekter før sammenlikning av grensesnittreferanser.
- Man kan bruke annen informasjon som grunnlag for sammenlikning. Vi har tidligere nevnt at permanente proxy-objekter må ha assosiert med seg en form for permanent identifikasjon av objektet, som brukes ved aktivisering. Er denne informasjonen lik, kan man slå fast at det dreier seg om samme (passive) objekt.

4. 3. Aktivisering av objekt-implementasjon

Vi skal her klargjøre hva det vil si å aktivisere objekt-implementasjoner. Vi skal både se på hvordan dette kan foregå og på hvilke objekter som skal være gjenstand for eksplisitt aktivisering.

4.3.1 Diskusjon av CORBA BOA sine “activation policies”

I CORBA Basic Object Adapter [OMG.92.12.1] gjøres det et klart skille mellom objekter og objektenes implementasjon, med hensyn til aktivisering. Det skilles mellom to typer aktivisering, nemlig aktivisering av implementasjon, det vil si et program (eller en prosess), og aktivisering av objekter. Det kan være ulike sammenhenger mellom aktivisering av implementasjon og aktivisering av objekt, noe som gjenspeiles i de 4 kategoriene av implementasjons-aktivisering som dokumentet om BOA skisserer:

- Shared Server: Flere objekter kan bli aktivisert av det samme program, og program må aktiviseres før objekter kan bli aktivisert.
- Unshared Server. Hvert objekt er implementert med sitt eget program. Aktivisering av program innebærer også at objekt blir aktivisert.
- Server per method. Det aktiviseres et program for hvert metode-anrop til et objekt. Her vil altså programmer bli aktivisert etter objektet.
- Persistent server. Implementasjonen aktiviseres av noe utenfor BOA. Implementasjonen vil varsle BOA om at den er tilgjengelig. Ellers som shared server.

I ANSAware, skal objektets lokasjon og implementasjon kunne være transparent for den som anvender objektet. Det skal kunne være mulig å anvende objekter, uten å vite hvem som har aktivisert dem, og det skal kunne være mulig å be infrastrukturen om å aktivisere objekter, uten å vite hvilket program som implementerer objektene.

I ANSAware vil vi kunne operere med abstraksjoner over objektenes implementasjon. I "engineering viewpoint" (jfr avsnitt 2.7) behandles disse abstraksjonene (kapsler) på lik linje med andre "engineering" objekter. Kapsler representerer programmer, og disse har grensesnitt en kan bruke for å aktivisere objekter.

Vi skal innføre et noe videre "tjener" begrep, når vi ser dette i forhold til CORBA BOA. En tjener kan være det objektet som representerer implementasjonen (kapsel), eller det kan være et annet objekt, som er i stand til å aktivisere (og eventuelt passivisere) objekter. Ut fra dette, vil diskusjonen i CORBA BOA, om de fire "activation policies" ha relevans, også mht. ANSAware. Følgende to "policies" ser ut til å passe for ANSAware:

- Shared Server: En tjener kan implementere flere objekter. Objekt-adapter må aktivisere tjener før objekter kan aktiviseres, hvis denne ikke allerede er aktiv (Dette passer f.eks. på kapsler).
- Persistent Server: Tjener aktiviseres av noe annet enn objekt-adapter. ANSA-arkitekturen har enkelte tjenere som skal gå permanent. Disse kan en anta er aktive i utgangspunktet, og de kan anvendes direkte (Dette passer f.eks. på trader og factory).

4.3.2 Permanente vs. temporære objekter

Det er to ulike måter å aktivisere på, alt etter om vi har med permanente eller temporære proxy-objekter å gjøre (se også avsnitt 4.2.4):

- For temporære objekter gjøres det ved at klienten anroper operasjoner i grensesnittet. Språkbindinga anroper ANSAware-operasjoner. Disse kan aktivisere ANSAware objekter og returnere grensesnitt til disse. Språkbindinga vil da måtte be objekt-adapter om å få opprettet proxy-objekter for disse.
- For permanente objekter er det altså slik at det opprettes proxy-objekter først (typisk ved oppstart av objekt-adapter). Persistent informasjon assosieres med disse. De permanente proxy-objektene vil være i stand til å aktivisere tilhørende ANSAware objekt når det er behov for det.

Det er altså bare for permanente objekter at objekt-adapter behøver å sørge for aktivisering av ANSAware objekter (jfr. avsnitt 2.5.3). Ellers skjer det implisitt gjennom anrop av bruker-operasjoner.

4. 4. Modell av interaksjon mellom klient og objekt-adapter

Vi skal nå definere noen begreper som kan brukes til å beskrive semantikken til interaksjoner mellom klient-program og objekt-adapter. Vi skal presentere et konseptuelt grensesnitt med et sett basale (abstrakte) operasjoner. Det er konseptuelt i den forstand at i konkrete grensesnitt mellom klient og objekt-adapter, vil vi ikke nødvendigvis se de samme operasjonene, men alle konkrete grensesnitt skal kunne forklares ved hjelp av det konseptuelle[2]. De basale operasjonene vil altså være representert med ulike konsepter, alt etter hva slags språkbinding vi har med å gjøre.

4.4.1 Objekt referanser

Objekt referanser er her verdier som objekt-adapter gjennom ei gitt språkbinding tilbyr klient-programmet, som unike identifikatorer for objekter. Objekt-referanser tilbys klientprogram av språkbindinga. Disse er representanter for de egentlige objekter. I klient-programmeringsomgivelsen, vil en ha egne typer (verdi-typer) for objekt-referanser. Verdier av disse typene kan brukes som argumenter og resultater i operasjons-anrop, og de kan kopieres, slettes og tilordnes variable (uten at dette fører til tilsvarende i objekt-adapter ellers, eller i det lokale systemet). Semantisk sett likner dette på pekere.

I våre omgivelser, vil en objekt-referanse være (eller inneholde) en mer eller mindre direkte referanse til proxy-objektet, som i objekt-adapteren opptrer som representanter for lokale objekter (jfr. avsnitt 4. 2).

4.4.2 Basale operasjoner

Det er relevant å definere tre basale operasjoner i et konseptuelt grensesnitt:

- *Bind* - Definere binding mellom klient og objekt.
- *Unbind* - oppheve binding
- *Invoke* - anrope operasjon.

Dette er tilstrekkelig for å beskrive interaksjonen mellom klient og objekt-adapter. Vi har ikke tatt med operasjoner for å opprette, kopiere eller fjerne objekter i det lokale systemet. I den grad det lokale systemet tillater det, er det ønskelig å ha tilgang til slike operasjoner fra klient-programmet, men vi trenger ikke nødvendigvis å addere basale operasjoner. Slik funksjonalitet kan tilbys som egne objekt-operasjoner som kan anropes via invoke. Dette gjenspeiler bedre at ikke alle applikasjoner tilbyr dette.

Bind-operasjonen

Denne operasjonen indikerer at klient vil bruke et objekt (klienten "binder" seg til et objekt). Dette henger ofte sammen med at et variabelnavn i et program bindes til en objekt-referanse. Binding innebærer også enten en "tilkopling" til et eksisterende proxy-objekt, eller at det opprettes et nytt proxy-objekt som klient "koples" til.

Argument er:

- En identifikasjon av objektet. For eksempel et navn.

Resultat er:

---

2. Grensesnitt til OSI-tjenester (se f.eks. [HeSh90]) beskrives på tilsvarende måte ved hjelp av tjeneste-elementer som brukes for å beskrive funksjonalitet, men som ikke er ment å legge føringer på hvordan grensesnittet til OSI-tjenester realiseres. Det konseptuelle grensesnittet som behandles her likner mye på tjeneste-elementene som er definert i OSI-standarden for fjerne operasjoner (ROS), men det er ikke spesifisert noen protokoll her.

- Objekt-referanse. Dette vil være en referanse til et aktivt objekt. Aktivt i den forstand at klient kan anrope operasjoner (*invoke*)[3]. Objekt-referansen kan senere brukes i *invoke* (se nedenfor) for å identifi sær objektet.

Implisitt binding

Binding er som regel implisitt. Det vil si at ved anrop på operasjoner som returnerer objekter (grensesnittreferanser i ANSAware omgivelser), blir det implisitt gjort bindinger. For hvert objekt som returneres fra anropet, vil det bli returnert en objekt-referanse til klient-programmet.

De fleste bindingene blir gjort implisitt, det vil si objekt-referanser returneres fra operasjonsanrop. De objekter som er persistens-røtter (jfr. avsnitt 2.4.1), vil måtte bindes eksplisitt, fordi de ikke kan nås gjennom anrop på andre objekters operasjoner[4].

Unbind-operasjonen

Denne operasjonen indikerer at klient ikke lenger bruker et objekt og at den ikke lenger har behov for referanse til det. Objekt-adapter implementasjonen kan utnytte dette for å kunne søppelsamle proxy-objekter som ingen klient lenger har interesse av. Argument til denne operasjonen vil være en objekt-referanse.

Invoke-operasjon

Denne operasjonen angir et anrop til en operasjon i det lokale informasjonssystemet. Som nevnt ovenfor vil en *invoke* for en operasjon som returnerer objekt(er), også (implisitt) være en *bind*-operasjon.

Argumenter er:

- Operasjons identifi kator (navn)
- Argument til anrop. Alle verdi-typer er tillatt (inkludert objekt-referanser).

Resultat er:

- Resultat fra anrop. Alle verdi-typer er tillatt (inkludert objekt-referanser).

Feilsituasjoner som kan oppstå (må håndteres av klient) er:

- Ukjent operasjons navn.
- Typefeil i argument
- Ugyldig objekt-referanse
- Feil ved anrop (problem med kommunikasjon/nett o.l.)

4.4.3 Eksempel (språkbinding til C++)

Vi skal se på et eksempel på ei språkbinding til et populært objekt-orientert programmeringsspråk, nemlig C++ [Stroustrup91]. Vi skal se hvordan C++ konstruksjoner svarer til de abstrakte operasjonene beskrevet ovenfor.

---

3. Dette betyr ikke nødvendigvis at objektets implementasjon er aktiv, men det betyr at det finnes et aktivt proxy-objekt.

4. Noe tilsvarende blir gjort i ObjectStore [OStore.2.0]. Persistensrot objekter må her gis tekstlige navn og må eksplisitt bindes til pekere (objekt-referanser) ved bruk.

Objekt-referanser er her representert som klasser. Det er en objekt-referanse klasse for hver objekt-type. Grensesnittene til objektene er representert som metoder i klassene. Metodenavn svarer til operasjonsnavn. En '*invoke*' med et operasjonsnavn svarer til kall på metode med dette navnet. Argument- og resultat-typer er typer som i C++ svarer til argument- og resultat-typer i det lokale systemets eksport-skjema[5]. I tillegg til metoder som svarer til operasjoner i eksport skjema har vi konstruktør-metoder, destruktør-metoder og metoder for tilordning. Vi skal se hva dette betyr.

FIGUR 30. Eksempel på objekt-referanse klasse og hva metodene svarer til

```
class AccountRef public ObjectRef
{
public:
   AccountRef(AccountRef&);                    // bind
   ~AccountRef();                              // unbind
   AccountRef& operator = (BankAccountRef&);   // unbind + bind
   ...
   int Credit(Real);                           // invoke "Credit"
   int Debit (Real);                           // invoke "Debit"
   Real Saldo ();
};
```

Bind

Eksplisitt binding kan skje på tre måter i ei slik C++ språkbinding. For det første ved kall på konstruktør-metode, der objektet (persistens-rot) identifiseres med et navn eller liknende. For det andre vil binding kunne skje ved kopiering av objekt-referanser, altså ved kall på kopi-konstruktør-metode (kopi-konstruktør oppretter et objekt som er kopi av et annet). For det tredje vil binding skje ved tilordning av et objekt til en variabel. Den verdien variabelen hadde fra før vil bli kastet og ny blir tilordnet.

FIGUR 31. Eksempler på 'bind' i C++

```
{
   BankRef bank ("Fokus-bank"); // Deklarasjon av variabel som initialiseres. Binding
                                // gjøres til objekt med navn "Fokus-Bank" (1)
   BankRef xBank = bank;        // Deklarasjon av variabel som initialiseres med kopi
                                // av en annen. (2)
   xBank = yBank;               // Variabel får ny binding (3)
   ...
}
```

Unbind

To situasjoner vil føre til at '*unbind*' må utføres: For det første vil det være tilfellet når en automatisk variabel som f.eks. *bank* i figuren ovenfor går ut av skopet og dermed ikke lenger er gyldig, eller at en et dynamisk allokert objekt slettes eksplisitt. Allokert lager må da deallokeres og bindinger oppheves. Her vil destruktør-metode bli kalt og opprydningsarbeid kan implementeres der.

For det andre vil det være tilfellet ved tilordning. Den bindinga en variabel har vil da bli skiftet ut med en ny. Den "gamle" bindinga vil oppheves.

5. Det er ikke relevant å gå nærmere inn på avbildning av verdi-typer her.

4. 5. Oppsummering

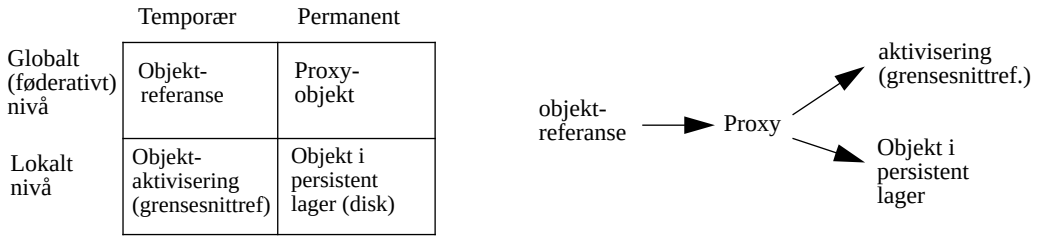
I dette kapitlet har vi presentert en del overordnede prinsipper for komponentene som skal avbilde mellom en føderativ omgivelse og et lokalt informasjonssystem. Vi har identifisert to komponenter, nemlig objekt-adapter og språkbindinger. Objekt-adapter forvalter objekter og tilbyr transparens med hensyn til persistens, og språkbindinger (stubs) tilbyr aksess-transparens.

Vi har innført begrepet "proxy-objekt". Proxy objekter vedlikeholdes av objekt-adapter og er representanter for lokale objekter i en føderativ omgivelse. De har ansvaret for identifikasjon og eventuelt aktivisering av lokale objekter. Vi har kort diskutert sentrale egenskaper ved proxy-objekter. Vi har også sett på hvordan objekters implementasjon aktiviseres. Man aktiviserer (eller finner aktiviseing) ved hjelp av andre objekter (vi kaller dem "tjenere"). Her kan vi skille mellom tjenere som forvaltes eksplisitt (f.eks. kapsler) og tjenere som vi kan anta er aktive i utgangspunktet og som forvaltes av infrastrukturen (f.eks. trader).

Til slutt har vi presentert en modell for interaksjon med klienter, noe som først og fremst angår språkbindinger. Objektets representasjon i klient-programmerings-omgivelsen kaller vi objekt-referanser. I forhold til disse har vi tre konseptuelle operasjoner: Bind, unbind og invoke, for å manipulere med binding mellom objekt-referanser og proxy-objekter og for å anrope operasjoner.

I dette kapitlet har vi diskutert objekt-referanser og proxy-objekter som to aspekter ved objekt-identifikasjon på det føderative nivå. Vi kan plassere disse i klassifiseringa fra avsnitt 3.4.3. Objekt-referanser identifiserer proxy-objekter som identifiserer lokale objekter. Figuren nedenfor illustrerer dette:

FIGUR 32. Aspekter ved objekt-identifikasjon



Kapittel 5
Et rammeverk for realisering av objekt-adapter

I dette kapitlet ser vi på hvordan en objekt-adapter som beskrevet i kapittel 4 kan realiseres. Vi skal presentere et gjenbrukbart objekt-orientert design og se på hvordan det virker. Først skal vi presentere et statisk design av klasser som objekt-adapter bygges opp av og til slutt skal vi se på en del dynamiske egenskaper ved objekt-adaptere.

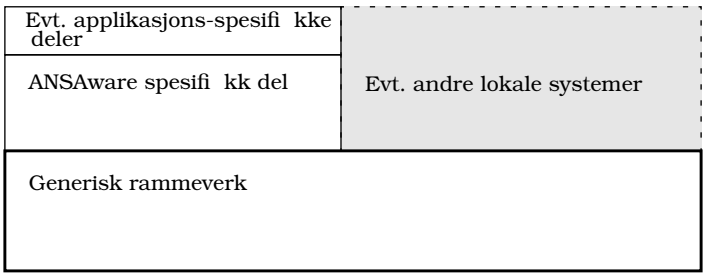
5. 1. Design av objekt-orientert rammeverk

I dette avsnittet beskrives design av et rammeverk for konstruksjon av objekt-adaptere. Vi skal se nærmere på hva som menes med begrepet rammeverk her, men først skal vi skissere noen overordnede prinsipper som bør gjelde. For vårt objekt-adapter rammeverk bør følgende gjelde i størst mulig grad:

- Det bør kreves minst mulig applikasjons spesifikk kode i en objekt-adapterDet vil si at mest mulig av jobben bør være gjort på et mer generisk nivå
- Det er også ønskelig at rammeverket er så generelt at deler av det kan brukes for andre lokale systemer enn ANSAware. Her ønsker vi mest mulig av design og implementasjon løftet opp på et generisk nivå, slik at det kreves så lite som mulig systemspesifikk kode.

Objekt-adapter tenkes konstruert etter en byggekloss-filosofi. En har i utgangspunktet et generisk rammeverk. Dette er definisjoner og programkode som kan brukes for flere lokale systemer. Dette kan bygges ut med ANSAware spesifikk del,som igjen kan bygges ut med applikasjonsspesifikke delerFør vi beskriver dette nærmere skal vi se på hvilken betydning vi skal legge i begrepet rammeverk.

FIGUR 33. Byggeklosser for objekt-adapter



5.1.1 Hva er et rammeverk?

Et objekt-orientert rammeverk ([Wirf90a], [Wirf90b]) representerer et abstrakt design for en hel applikasjon eller subsystem, innenfor et spesifikt domene. En slik "generisk applikasjon" blir raffinert til en spesifikk, ved å lage subklasser av de abstrakte klassene og implementere de abstrakte metoder som ble definert i de abstrakte klassene. Disse metodene blir først og fremst kalt fra rammeverket sjøl. På denne måten oppnår man ikke bare gjenbruk av kode, men også gjenbruk på design-nivå. Man oppnår også at mye av den globale flyt-kontroll kan plasseres i prefabrikert rammeverk-kode.

[HaeDitt93] viser hvordan rammeverk-ideen kan anvendes for programvare for integrasjon av ulike komponent-databaser i en felles global datamodell. Over de ulike komponent-databaser tenker de seg et såkalt "homogeniserings lag" implementert som et "integrasjonsrammeverk" som består av hierarkier av samarbeidende (C++) klasser.

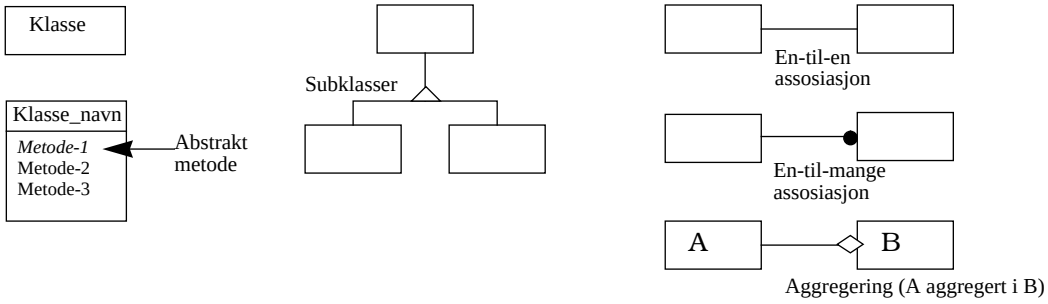
Meninga er at designet og store deler av den "koplings software" som kan gjøre en komponent-database til medlem av det globale objekt-rom, skal være produsert på forhånd. I dette designet vil det finnes en del såkalte elementære deloppgaver, som er avhengig av det lokale system (eller lokale datatyper). Disse elementære deloppgavene innkapsles i abstrakte metoder som implementeres for hver komponent-database eller type. Prefabrikert integrasjonskode vil bygge på disse elementære metodene, noe som impliserer en stor grad av gjenbrukbarhet.

5.1.2 Klassehierarkiet

Rammeverket består av et sett med samarbeidende klasser. En del av klassene er abstrakte og har subklasser hvor de abstrakte metodene blir implementert. Før vi presenterer designet, skal vi kort introdusere notasjonen som blir brukt:

Vi vil i det følgende bruke en grafisk notasjon for objekt-orientert modellering og design, sterkt inspirert av notasjonen som brukes i [Rumbaugh91]. Vi bruker notasjonen for å vise hvilke klasser som er med og hvilke relasjoner det er mellom disse klassene. Kort sagt er det slik: Klasser tegnes som bokser. Det kan være ulike relasjoner mellom klasser. Her anvendes subklassing, assosiasjoner og aggregering. Figuren nedenfor oppsummerer de viktigste notasjoner her. Det er verdt å merke seg at klasser kan ha med navn på attributter og metoder. Vi har noen steder i dette dokumentet tatt med metodenavn. Abstrakte metoder er skrevet i kursiv og konkrete metoder i vanlig tekst.

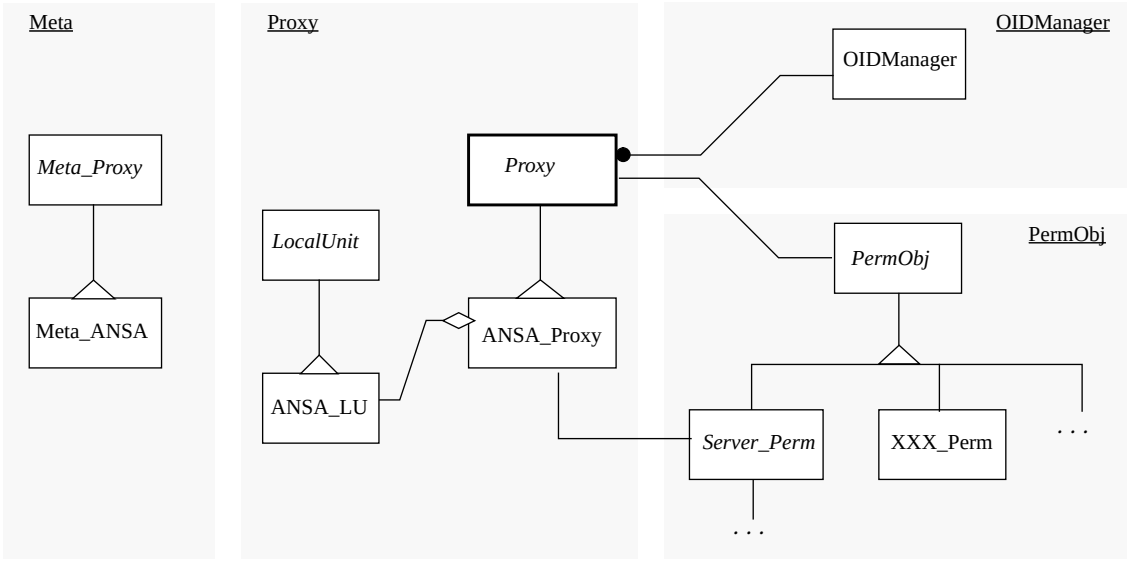
FIGUR 34. Notasjon



Designet av objekt-adapter rammeverket er slik: Sentralt i rammeverket har vi en *Proxy*-klasse. Proxy-objektet som ble beskrevet i avsnitt 4. 2, vil her være representert med en egen klasse! Vi har i utgangspunktet en abstrakt proxy-klasse som kan spesialiseres for ulike lokale systemer. Ellers har vi disse klassene:

- LocalUnit som representerer den lokale representasjonen av objekter.
- Meta_Proxy for aktivisering (oppretting og installering) av proxy-objekter.
- OIDManager - Holde oversikt over alle aktive proxy-objekter. OIDManager har metoder for å tildele nye OID'er, sjekke om en gitt proxy allerede eksisterer, installere nye proxyer, finne en proxy og eventuelt fjerne proxy'er.
- PermObj - Aktivisere passive objekter (implementasjon), holde nødvendig tilstandsinformasjon, for å kunne gjøre dette.

FIGUR 35. Klassehierarki for objekt-adapter



5.1.3 Proxy-klassen

En proxy er et objekt som opptrer som representant for et objekt i et lokalt informasjonssystem, som f.eks. ANSAware. Proxy-objekter inneholder eller har assosiert med seg informasjon (og kode) som er nødvendig for å identifisere og aksessere objektet (fra en klient) og for å aktivisere (og eventuelt passivisere) objektets implementasjon der vi har permanente proxy-objekter.

TABELL 1. Proxy-objektets metoder

Metode	Beskrivelse	Synlig	Abstr.
is_SameAs (Proxy*)	Returnerer TRUE hvis den angitte proxy representerer det samme lokale objekt (se forøvrig avsnitt 4.2.5).	Public	Nei
get_Activation ()	Returnerer *LocalUnit* som representerer (lokal) aktivisering av objektet. Hvis objektet er passivt, vil det blir forsøkt aktivisert.	Public	Nei
impl_Activate ()	Aktiviserer objektets implementasjon hvis mulig. Returnerer indikasjon på om aktivisering var vellykket.	Public	Nei
impl_Passivate ()	Passiviserer objektets implementasjon hvis mulig. Returnerer indikasjon på om passivisering var vellykket.	Public	Nei
impl_Is_Active ()	Returnerer TRUE hvis implementasjonen av objektet er aktiv?	Public	Nei
get_LU ()	Returnerer (referanse til) proxy-objektets tilhørende *LocalUnit* objekt. Kalles internt fra rammeverket (*get_Activation*)	Protected	Ja
compareLoc (Proxy*)	Representerer den angitte proxy den samme aktivisering av det lokale objekt. Metoden vil kun sammenlikne *LocalUnit* objektene. Kalles internt fra rammeverket (*is_SameAs*)	Protected	Ja

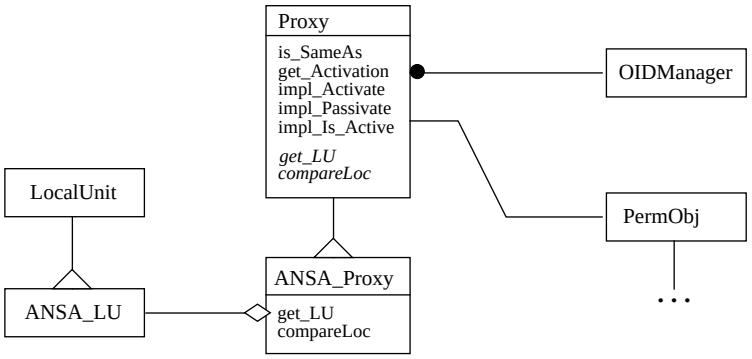
Nært koplet til proxy-objekter er språkbindinga. Språkbindinga vil ha referanser til proxy-objektene (typisk gjennom pekere) og via disse få aksess til objektenes egentlige representasjon.

Proxy-klassen er abstrakt. For de lokale informasjonssystemene vil det måtte defineres subklasser av denne hvor de abstrakte metodene blir definert. Metodene til *Proxy* er listet opp i tabellen ovenfor. Vi ser at bare to av metodene er abstrakte og gjør relativt lite. Disse vil også kun bli kalt fra rammeverket. Permanente proxy-objekter vil ha assosia-

sjon til et *PermObj* objekt. Dette vil være i stand til å foreta aktivisering og passivisering av objektets implementasjon.

For ANSAware har vi definert en spesialisering av *Proxy*, nemlig *ANSA_Proxy* som aggregerer og har kjennskap til et objekt av *ANSA_LU* klassen. Det er ikke nødvendig å definere flere enn en konkret proxy-klasse for ANSAware. Forskjeller mellom ulike typer ANSA-grensesnitt og ulike forvaltningsstrategier for ANSA-objekter, vil gjenspeiles i språk-bindinga og eventuelle *PermObj* klasser (gjenspeiler ulike måter å aktivisere/passivisere).

FIGUR 36. Proxy-klassen



5.1.4 LocalUnit klassen

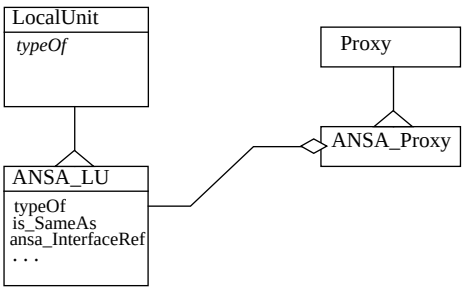
Spesialiseringer av *Proxy* vil inneholde (aggregere) et objekt som inneholder den lokale representasjonen av objektet. Dette objektet vil være en spesialisering av den abstrakte klassen *LocalUnit*. Klassen *LocalUnit* inneholder det lokale systemets representasjon av objektet, slik det er i aktivisert form. Hvert lokale system må definere (minst) en subklasse av *LocalUnit* hvor metoden *typeOf* er definert, samt at man her definerer en eller flere metoder for å aksessere den lokale representasjonen direkte.

Vi har valgt å ha med en abstrakt klasse for lokal representasjon for at man på et abstrakt nivå skal kunne referere til, og teste typen til lokale representasjonsenheter, uten å kjenne til hva de egentlig består av og hvordan de er implementert.

Spesialisering for ANSAware

I ANSAware representeres objekter med grensesnittreferanser. *ANSA_LU* er en subklasse av *LocalUnit* og inneholder en grensesnittreferanse. Her legges blant annet til en metode *is_SameAs* som implementeres ved å bruke ANSAwares funksjoner for sammenlikning av grensesnittreferanser (Se for øvrig avsnitt 4.2.5). Vi trenger også en metode for å konvertere fra *ANSA_LU* til en grensesnittreferanse som kan brukes av språkbinding til operasjonsanrop. Fordi alle aktive ANSAware objekter er representert med grensesnittreferanser, er det ikke nødvendig å definere flere *LocalUnit* klasser for ANSAware applikasjoner.

FIGUR 37. LocalUnit klassen



5.1.5 Meta proxy-klassen

For hver Proxy-klasse vil en ha behov for et meta-objekt. Dette for å kunne aktivisere proxy-objekter (det vil si opprette og installere dem), f.eks. for en gitt lokal-enhet. Vi vil, ved implementasjon av metoder på det abstrakte nivå i rammeverket, ha bruk for å instansiere proxy-objekter uten at den eksakte klassen til disse er kjent. Derfor anses det som nødvendig å innføre en abstrakt meta-klasse for proxy-klassen. Det må definieres en konkret *Meta_Proxy* klasse for hver konkrete *Proxy*-klasse. Inntil videre ser vi bare på metoder for aktivisering av proxy-objekter. Aktivisering av disse innebærer følgende:

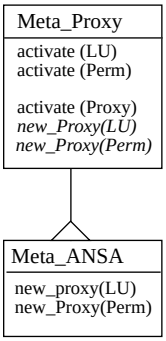
- Å finne og returnere eventuelt tidligere aktiviserte proxy-objekt for et gitt *LocalUnit* objekt eller *PermObj* objekt.
- Instansiere proxy-objekt hvis det ikke eksisterer fra før.
- Ny-instansierte proxy-objekter må assosieres med en global objekt-identifikator og registreres, slik at de kan gjenfinnes senere.

TABELL 2. Metoder for Meta_Proxy

Metode	Beskrivelse	Synlig	Abstr
activate (LocalUnit&)	Aktivisere (temporær) *Proxy* for angitt *LocalUnit*	public	Nei
activate (PermObj*)	Aktivisere (permanent) *Proxy* for angitt *PermObj*.	public	Nei
activate (Proxy*)	Returnere allerede eksisterende proxy hvis det finnes fra før. Returner den angitte proxy hvis ikke.	protect	Nei
new_Proxy(LocalUnit&)	Instansiere proxy-objekt.	protect	Ja
new_Proxy (PermObj*)	Instansiere proxy-objekt	protect	Ja

Vi ser her at vi enten kan aktivisere proxy-objekt for et gitt *LocalUnit* objekt eller for et gitt *PermObj* objekt. I avsnitt 4.2.4 skisseres to ulike måter å aktivisere proxy-objekter og det er egentlig det som gjenspeiles her. For aktivisering etter alternativ 1 (temporære objekter) oppgis et *LocalUnit* objekt (det vil inneholde en grensesnittreferanse for et allerede aktivt ANSA-objekt). For aktivisering etter alternativ 2 (permanente objekter) oppgis et *PermObj* objekt som representerer det passive objektet og hvordan det aktiviseres. Metoder for aktivisering vil være realisert på et abstrakt nivå i rammeverket. Disse vil trenge metoder som instansierer proxy-objekt, og disse vil være abstrakte. De vil måtte definieres i en klasse som er en spesialisering av *Meta_Proxy*. Vi har her *Meta_ANSA*.

FIGUR 38. Meta_Proxy



5.1.6 PermObj klassen

Denne klassen representerer den informasjon og de metoder som er nødvendig for å kunne aktivisere permanente objekter. *PermObj* klassen har også ansvaret for passivisering av objekter det blir bedt om det[1].

PermObj-klassen er abstrakt. Metoder for aktivisering og passivisering er abstrakte og vil bli definert i subklasser av *PermObj*. Det samme gjelder informasjon som identifiserer objektet i persistent lager (jfr. avsnitt 3.4.2). Siden ANSAware ikke har en uniform tjeneste for aktivisering, vil definisjonen av konkrete *PermObj*-klasser avhenge av den aktuelle applikasjonen, både mht. metoder og identifikasjon av objekter. Vi vil imidlertid som regel kunne anvende forvaltnings-tjenester som er definert i ANSAware arkitekturen. Et objekt vil (som antydet i avsnitt 3.5.2) i en gitt applikasjon kunne aktiviseres/passiviseres på en av følgende måter:

- Via factory (gjelder bare kapsel-objekter)
- Via kapsel-grensesnitt.
- Via node-forvalter.
- Via trader (ikke egentlig aktivisering, men "gjenbruk" av aktiviseringer som er gjort av andre).
- Via forvaltnings-grensesnitt. ANSA-objekter kan ha et forvaltnings-grensesnitt (kalles 'Object') som har en operasjon for passivisering av objektet.
- Via brukerdefinerte grensesnitt..

Aktivisering via factory, kapsel, node-forvalter, trader eller objekters forvaltnings-grensesnitt, skjer på måter som er definert i ANSAware arkitekturen, og dermed blir implementasjon av slik forvaltning svært enkel, fordi vi kan bruke ferdige tjenester og tilby ferdige rutiner for anrop av disse i et bibliotek. For en konkret *PermObj* klasse, vil vi som regel kunne velge fra et sett av forvaltnings-metoder.

TABELL 3. Metoder for *PermObj*

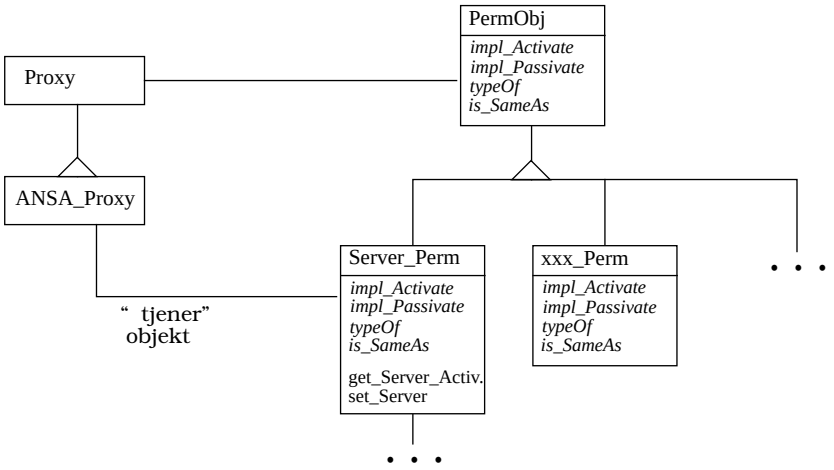
Metode	Beskrivelse	Synlig	Abstr.
impl_Activate (LUnit&)	Aktivisere objekt og oppdater angitte *LocalUnit* objekt med den nye aktivisering. Funksjonen returnerer OK hvis vellykket og ERROR hvis ikke.	public	Ja
impl_Passivate (LUnit&)	Passivisere objekt. Objektets aktivisering er angitt i form av en referanse til en *LocalUnit*. Returnerer OK eller ERROR på samme måte som *impl_Activate*.	public	Ja
typeOf ()	Brukes ved sammenlikning av *PermObj* objekter Vi må under kjøretid kunne avgjøre om de to objektene som sammenliknes er av samme konkrete klasse. Kalles internt fra rammeverket (*PermObj::is_SameAs*)	public	Ja
is_SameAs (PermObj&)	Sammenlikne to *PermObj* objekter om de representerer samme permanente objekt.	public	Ja

Diskusjonen i avsnitt 4.3.1. antyder av vi har å gjøre med to typer aktiviserings-"policies" for ANSAware applikasjoner. Shared Server prinsippet antyder at vi er avhengig av et annet objekt (et "tjener" objekt) som vi bruker for forvaltning. Det synes hensikts-

1. Man kan tenke seg at klient program har mulighet til å passivisere objekter. Man kan også tenke seg at passivisering skjer etter en bestemt "policy" som er representert ved et policy-objekt. Et slikt objekt vil ha assosiasjoner til de proxy-objekter som forvaltes etter den aktuelle "policy".

messig å definere en egen klasse (subklasse av *PermObj*) som understøtter objekter forvaltet etter Shared Server prinsippet. Dette definerer en referanse til et annet proxy-objekt (som representerer "tjener" objektet) og en metode for å få tak i aktivisering av dette. De andre metodene vil fortsatt være abstrakte. For Persistent Server prinsippet er det ikke nødvendig å definere noe nytt. Forvaltningsmetoder kan anta at "tjener" objektet som skal anvendes er tilgjengelig i utgangspunktet og permanent.

FIGUR 39. PermObj hierarkiet



5.1.7 OIDManager klassen

Alle aktive *Proxy*-objekter er registert i *OIDManager*. Den har som oppgave å forvalte objekt-identifikatorer og proxy-objekter. Ved hjelp av *OIDManager* skal vi kunne vite hvilke proxy-objekter som til enhver tid eksisterer. Vi skal kunne finne fram til et proxy-objekt, gitt at vi har en *OID*, og vi skal kunne sjekke om et proxy-objekt allerede eksisterer i *OIDManager*. *OIDManager* har følgende metoder:

FIGUR 40. Metoder for OIDManager

Metode	Beskrivelse	Synlig	Abstr
getNewOID ()[a]	Returnerer ny unik *OID*	public	Nei
install (Proxy*)	Installerer det angitte proxy-objektet. Forutsetter at dette er gitt en unik *OID*.	public	Nei
remove (OID)	Fjerner det angitte objektet. Metoden kan også kalles med argument av typen *Proxy**	public	Nei
lookup (OID)	Returnerer *Proxy** for den angitte *OID*. NULL hvis ikke installert.	public	Nei
checkActive (Proxy*)	Returnerer *Proxy** som representerer samme lokale objekt som den angitte proxy, hvis dette eksisterer i *OIDManager*. NULL hvis det ikke eksisterer.	public	Nei

a. Man kan tenke seg at OID som en opsjon kan avledes fra LocalUnit. Dette blir diskutert nærmere i kapittel 9.

Vi tenker oss her en instans av *OIDManager*, men det behøver ikke være noe i veien for å ha flere. En mulig effektivisering ville være å ha en *OIDManager*-instans for hver objekt-type. Dette vil partisjonere mengden man søkte ved oppslag etter et bestemt proxy-objekt, og dermed effektivisere oppslag.

5. 2. Dynamiske egenskaper

Vi har til nå bare beskrevet et statisk design av objekt-adapter rammeverket. I dette avsnittet skal vi se nærmere på objekt-adapterens dynamiske egenskaper, altså hvordan objekt-adapter vil oppføre seg i ulike situasjoner.

Vi skal se på hva som gjøres når en objekt-adapter starter opp, hvordan objekt-adapter etablerer bindinger mellom klient-programmets form for objekt-referanser og ANSAware objekter, vi skal se på noen scenarier for hva som skjer ved operasjonsanrop og vi skal se på hvordan objekt-referanser som ikke brukes lenger, kan søppelsamles.

5.2.1 Aktivisering av objekt adapter

Ved oppstart av en objekt-adapter, må følgende gjøres:

- Opprette en instans av *OIDManager*.
- Opprette instanser av *Meta-Proxy*. En instans for hver klasse av proxy-objekter.
- Opprette proxy-objekter for permanente objekter. Det betyr også at det må opprettes en instans av *PermObj* for hver av disse. Dersom vi ønsker at permanent objekt-identitet skal overleve flere aktiviseringer av objekt-adapter, må den dynamiske informasjonen *PermObj*-objektene trenger for å kunne aktivisere de tilhørende ANSAware objektene, lagres i persistent lager eller "hardkodes" i implementasjonen. Det siste er greit hvis informasjonen ikke endres underveis.

Figuren nedenfor viser et eksempel på hvordan definisjon og aktivisering av permanente proxy-objekter kan "hardkodes" i programkoden til objekt-adapter. Først opprettes en instans av *CapsulePerm* (subklasse av *PermObj*) for kapselen som objektene blir implementert i. Nødvendig informasjon som da oppgis er maskin-navn som kapsel skal kjøre på, sti i filsystemet hvor program befinner seg og navn på program. Det opprettes en proxy for kapselen, og i dette eksempelet blir hvert enkelt *planet*-objekt eksplisitt lokalisert til kapselen. Proxy for kapsel er argument ved instansiering av *PlanetPerm* (subklasse av *PermObj*). Andre argumenter ved instansiering kan brukes for å identifisere bestemte instanser. For eksempel ved navn (f.eks. "jupiter" eller "saturn").

FIGUR 41. Eksempel på aktivisering av permanente objekter (i C++)

```
CapsulePerm *pc1;
PlanetPerm *po1, *po2;
AnsaProxy *obj1, *obj2, *obj3;
...

pc1 = new CapsulePerm ("odslab1", "/usr/local/Planet", "planetserv");
obj1 = ansa_META.activate(perm1);

po1 = new PlanetPerm (obj1, "Jupiter", 5);
po2 = new PlanetPerm (obj1, "Saturn", 6);

obj2 = ansa_META.activate(po1);
obj3 = ansa_META.activate(po2);
```

5.2.2 Binding

Et klient-program vil eksplisitt måtte binde seg til persistens-rot objekter. Hvordan dette gjøres avhenger sjølsagt av hva slags språkbinding en har med å gjøre. Men uansett så må klientprogrammet på en eller annen måte identifisere de objekter det vil binde seg til. Det kan gjøres på ulike måter:

- Eksisterende objektreferanse (for å lage flere bindinger til samme objekt)
- Objekt-identifikator som kan identifisere (f.eks. gjennom oppslag i tabeller) riktig proxy-objekt.

- Tekstlig navn som kan avbildes til riktig proxy-objekt (Forutsetter at slik avbildning eksisterer)
- Proxy-objektet direkte (hvis det er kjent).

En kan i enkelte tilfeller bruke trading som et hjelpemiddel i bindingsprosessen. Klienten oppgir en rekke ønskede egenskaper til en trading tjeneste (se f.eks. avsnitt 2. 7) og får tilbake et eller flere objekter (objekt-identifikatorer eller objekt-referanser) som passer til de ønskede egenskaper. Dette forutsetter at en form for trading-tjeneste er med i kjøretidsomgivelsene for det aktuelle programmeringsspråk, og at objekt-adapter "eksporterer" de permanente objekter til denne trading-tjenesten.

Figuren nedenfor viser et eksempel på hvordan klienten kan bindes til et objekt ved hjelp av trading. Vi fortsetter med planet-objektene fra eksemplet over og tenker oss at objekt adapter eksporterer hver av disse til en trader (figuren til venstre). Klient-program vil kunne importere et passende persistens-rot objekt som det kunne binde seg til (figuren til høyre) Import-operasjonen vil f.eks. kunne returnere en objekt-referanse.

FIGUR 42. Eksempel på binding ved hjelp av trading (C++)

```
{
  // Eksport av proxy-er
  ...
  obj2 = META.activate(po1);
  obj3 = META.activate(po2);
  trader.export (obj2, ...);
  trader.export (obj3, ...);
  ...
}
```

```
{
  // Import av objekt-referanse
  PlanetRef world = trader.import(...);
  ...
}
```

5.2.3 Anrop av operasjoner

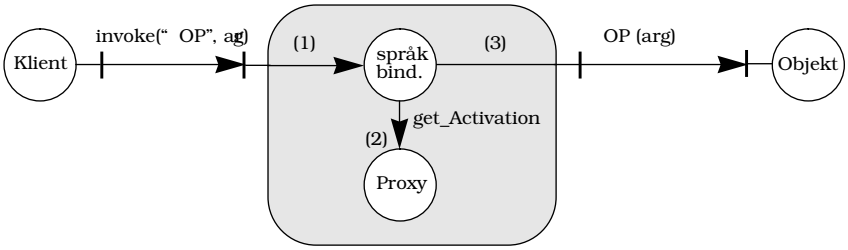
Vi skal her illustrere objekt-adapterens dynamiske struktur ved anrop av operasjoner, ved hjelp av noen scenarier. Vi skal se ulike måter systemet kan reagere på operasjonsanrop fra klient-program (invoke).

Anrop på aktivt objekt.

Anrop fra klienten på operasjonen "*OP*" (1), vil utløse følgende:

- Språkbindinga finner det riktige proxy-objektet, for eksempel ved å slå opp i *OID-Manager*[2]. Proxy-objektets metode *get_Activation* kalles (2) og den returnerer en *LocalUnit* som inneholder den informasjon som er nødvendig for at språkbindinga skal kunne anrope det egentlige objektet (det vil si ANSAware grensesnittreferanse).
- Språkbindinga anroper ANSAware objektet (3).

FIGUR 43. Operasjonsanrop på aktivt objekt



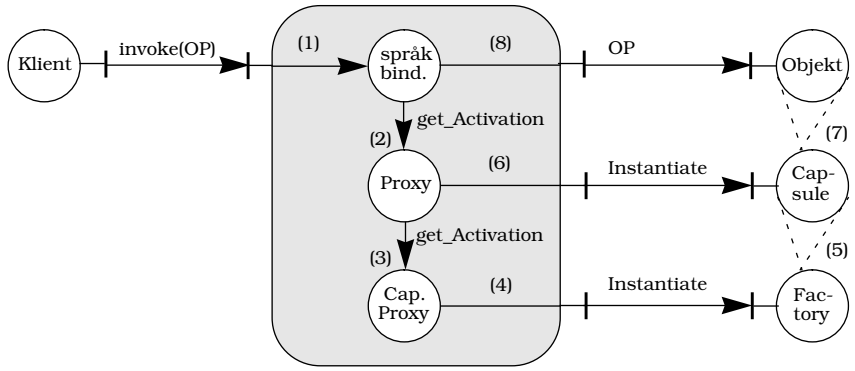
2. Dette er for enkelhets skyld ikke tatt med i figuren.

Anrop på passivt objekt

I dette eksemplet er objektet passivt. Det forvaltes av en kapsel som er representert i objekt-adapter med et eget proxy objekt. Kapsel er i utgangspunktet også passivt. Når klient anroper en operasjon i objektet, vil objekt-adapter forsøke å aktivisere objektet, før anropet utføres. Scenariet er som følger:

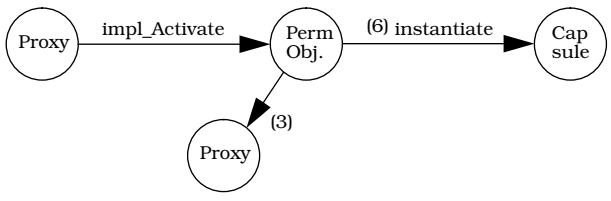
- Språkbindinga finner det riktige proxy-objekt og kaller *get_Activation* for å få tak i grensesnittreferansen (2). Siden objektet ikke er aktivt, vil proxy-objektet forsøke å aktivisere det.
- For å aktivisere objektet, må proxy anvende en kapsel. *PermObj* har referanse til en proxy for den kapsel som skal anvendes. Metoden *get_Activation* i denne kapsel-proxyen kalles for å få grensesnittreferansen (3).
- Kapsel-proxyen aktiviserer en kapsel (5) ved hjelp av *Factory* sin operasjon *instantiate* (4). *Factory* trenger ikke ha noen egen proxy, for vi regner den som en persistent tjener, som vi antar er aktiv i utgangspunktet (se avsnitt 4.3.1). For å finne riktig factory kan vi bruke ANSAware *Trader*.
- Objektets proxy kan nå anrope kapselens *instantiate* operasjon (6) for å aktivisere objektet (7). Dermed kan operasjonen "OP" anropes (8).

FIGUR 44. Anrop på passivt objekt som forvaltes av passiv kapsel



I figuren ovenfor har vi for enkelhets skyld utelatt *PermObj*. Disse er nært koplet til permanente proxy-objekter, og de skal gjøre jobben med å aktivisere passive objekter og lagre informasjon som er nødvendig for å få til dette. Figuren nedenfor viser forholdet mellom *Proxy* og *PermObj*: Proxy-objektet kaller metoden *impl_Activate* hos *PermObj* objektet, som utløser anrop på operasjonen *instantiate* (6) hos kapsel. Er kapsel passiv må altså *PermObj* henvende seg til kapselens proxy først (3).

FIGUR 45. PermObj objektenes rolle



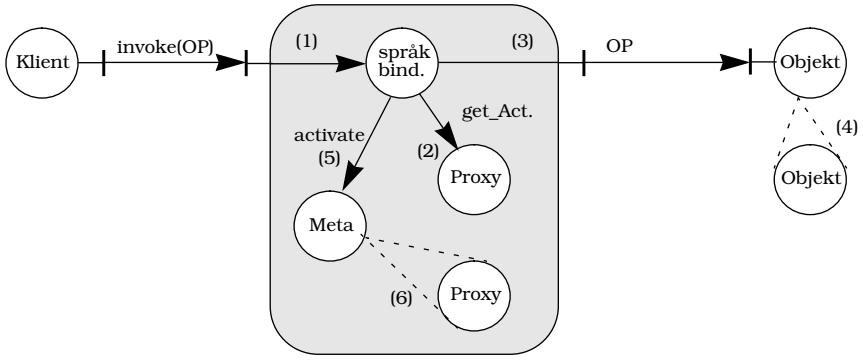
Operasjon som returnerer objekt(er)

I den abstrakte modellen for klient/objekt-adapter interaksjon kan klienten utføre *invoke* på operasjoner som returnerer objekt-referanser. Det vil si at *invoke* avbildes til en operasjon i et ANSAware-grensesnitt, som returnerer en eller flere grensesnittreferanser. Objekt-adapter sørger for at proxy-objekter blir aktivisert for disse, og at klien-

ten implisitt blir bundet til de objektene som har blitt aktivisert. (jfr. den abstrakte modellens *bind*-operasjon). Systemets respons på en *invoke* vil her være:

- Språkbinding finner riktig proxy-objekt og får tak i aktivisering for tilhørende objekt (2).
- Operasjon anropes i tilhørende ANSAware objekt (3) og denne utløser aktivisering av et nytt ANSAware objekt (4), og grensesnittreferanse til dette returneres til objekt adapter.
- Språkbinding sørger for at det nye ANSAware objektet får aktivisert et tilhørende proxy-objekt. *Meta-Proxy* objektet (for ANSAware) brukes til dette. Her kalles metoden *activate* (5) som sørger for å instansiere nytt proxy-objekt (6), gi det en unik identitet og installere det i *OIDManager*, slik at det kan gjenfinnes.

FIGUR 46. Operasjon som aktiviserer objekt (og returnerer gr.sn. referanse)



5.2.4 Søppelsamling

Et spørsmål rundt den dynamiske strukturen til objekt-adapter er hvordan proxy-objekter som ikke lenger er i bruk, kan bli deallokert, slik at ressurser kan frigjøres. Det er bare aktuelt å søppelsamle temporære proxy-objekter . Dette kan vi utlede fra definisjonen av temporær objekt-identitet (se avsnitt 2.5.2). Vi skal ikke gå i dybden i behandlinga av søppelsamlingsproblematikken i denne avhandlinga, men kort plassere en enkel strategi basert på referansetelling i forhold til rammeverket. Det er rom for diskusjon og forbedringer av denne strategien spesielt med hensyn til feiltoleranse[3].

Referanse telling

Referanse-telling passer naturlig sammen med de basale operasjonene *bind* og *unbind*. Det vil si at *bind* svarer til å øke antall referanser og *unbind* svarer til å redusere antall referanser.

Vi kan fra objekt-adapter rammeverket tilby to operasjoner som tar proxy-objekt som argument: *ref_Up* og *ref_Down* (tilsvarer *bind* og *unbind*) som henholdsvis inkrementerer og dekrementerer en referanse-teller som hvert proxy-objekt har. Hvis referanse-teller blir null for et temporært proxy-objekt, kan det fjernes.

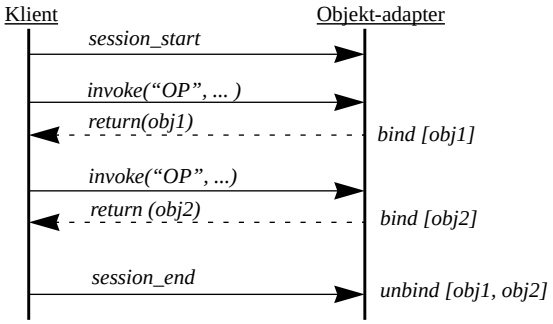
Eksempel

Bind/unbind semantikken behøver ikke nødvendigvis eksplisitt implementeres for hvert objekt i klient. En kan f.eks. tilby operasjonene: *session_start* og *session_end*. Ved kall på *session_end*, vil objekt-adapter implisitt kunne slutte at *unbind* skal utføres på alle objekter som klienten har bundet til siden *session_start*.

3. I distribuerte konfigurasjoner kan en for eksempel tenke seg at klient-entitet kræsjer uten å varsle objekt-adapter. Gode algoritmer og protokoller for søppelsamling må ta hensyn til slike tilfeller.

Figuren nedenfor viser et eksempel på sesjonsbasert interaksjon mellom klient og objekt-adapter. Innenfor sesjonen her gjør klient-programmet to operasjonsanrop som returnerer objekt-referanse. Disse bindes, det vil si referanse-teller inkrementeres (*ref_Up*). Ved *session_end*, vil objekt-adapter finne objekt-referanser som er assossiert med sesjonen og dekrementere referanse-tellere til hver av disse (*ref_Down*).

FIGUR 47. Scenario for klient-sesjon



5. 3. Oppsummering

Vi har i dette kapitlet sett på hvordan objekt-adaptere kan realiseres. Vi har presentert et objekt-orientert design av de klassene en konkret objekt-adapter bygges opp av. Her er vi inspirert av ideen om objekt-orienterte rammeverk og skisserer et rammeverk for objekt-adaptere hvor en betydelig del er spesifisert uavhengig av hvilket lokalt system man baserer seg på. Dette bygges ut med subklasser spesifikk for ANSAware.

Det sentrale her er en klasse for *Proxy*-objekter. Vi har en abstrakt klasse og en spesialisering av denne for ANSAware. Aktive lokale objekter er representert med en *LocalUnit* klasse. En spesialisering av denne for ANSAware består av grensesnittreferanse. For permanente objekter har vi en klasse *PermObj* som inneholder metoder for aktivisering og den informasjon som er nødvendig for å identifiser objekters tilstand. Spesialiseringer av denne er avhengig av applikasjon.

Vi har sett på scenarier for hvordan objekt-adaptere oppfører seg i ulike situasjoner. Spesielt har vi sett på aktivisering og binding av permanente objekter (ved oppstart av objekt-adapter og klient-program), på hva som skjer ved anrop av operasjoner. Til slutt har vi antydet at referansetelling kan være ei tilnærming til søppelsamling. Vi har sett hvordan bind/unbind semantikken (jfr. avsnitt 4. 4) svarer til referansetelling og hvordan dette kan gjenspeiles i grensesnitt mellom klient og objekt-adapter.

Kapittel 6
Defi nisjon av objekt adapter for konkret applikasjon

Det vi har beskrevet til nå er et rammeverk (design og delvis implementasjon) for objekt-adaptere. Vi må ha opplysninger om det lokale informasjonssystemet og opplysninger om den aktuelle applikasjonen (eller applikasjonene) som skal integreres, for at vi skal kunne realisere en komplett objekt-adapter. Her skal vi kort oppsummere hva som vil være grunnlag for defi nisjon, samt at vi presenterer en notasjon for spesifi kasjon av forvaltning av permanente objekter.

6. 1. Grunnlag for defi nisjon

Vi skal her se nærmere hva slags opplysninger som mangler og hva som må gjøres i forhold til rammeverket, gitt at vi kjenner disse opplysningene

6.1.1 De underliggende informasjonssystem

Har vi kjennskap til de datamodeller/DBMS/programmeringsomgivelser som applikasjonen(e) kjører under, kan vi lage spesialiseringer av klassene *Proxy*, *Meta-Proxy* og *LocalUnit* (jfr. avsnitt 5. 1)[1]. Vi har allerede inkorporert spesialiseringer for ANSAware i beskrivelsen av rammeverket og vi skal bygge videre på disse her.

6.1.2 Skjema informasjon (Typer)

Når vi har tilgang til skjema (spesifi kasjon av grensesnitt) for den aktuelle applikasjonen, betyr det at vi har en defi nisjon av de ulike objekttyper, slik de ser ut for en bruker av applikasjonen. Skjema-informasjon vil være det vi trenger for å generere språkbindinger (jfr. avsnitt 4.1.2). Dette vil kunne gjøres ved hjelp av en kompilator/ preprosessor (tilsvarer stub-kompilator) som tar et skjema som input og genererer kode og defi nisjoner for språkbindinga.

I enkelte tilfeller kan det også være aktuelt å spesialisere *Proxy*, *Meta-Proxy* eller *LocalUnit* for bestemte objekt-typer. Der det lokale systemet ikke tilbyr en felles form for representasjon av objektene (*LocalUnit* klassen vil dermed ikke være fullt defi nert), vil dette være aktuelt. I ANSAware tilsvarende omgivelser vil dette ikke være nødvendig. Alle objekter aksesseres der på en uniform måte (via grensesnitt-referanser og fjerne operasjoner).

1. Og eventuelt *OIDManager*, hvis det for et bestemt lokalt databasesystem skal defineres en subklasse og denne. .

6.1.3 Permanente objekter

En trenger også å ha kjennskap til hvilke objekter som er permanente og hvordan denne egenskapen blir realisert. Dette er informasjon en ikke kan utlede automatisk fra skjema, men som må spesifiseres manuelt (se for øvrig kapittel 3).

Spørsmål en i den forbindelse må stille er: Hvilke objekt-typer kan ha permanente instanser og hvordan skal objekter forvaltes for å oppnå persistens? Det vil si hva gjør en for å (re-) aktivisere objekter, og hvordan skal en gi disse riktig (persistente) tilstand ved aktivisering. Vi spør også her hvordan objekter passiviseres.

Informasjon om permanente objekter er representert med klassen *PermObj* (se avsnitt 5.1.6). Ut fra hva vi vet om forvaltning av applikasjonens permanente objekter, vil vi kunne lage spesialiseringer av *PermObj*. Vi kaller (for anledninga) spesifikasjonen av slike klasser for 'forvaltnings-klasser'.

6.1.4 Permanente instanser og persistens-røtter

En trenger også å vite hvilke permanente objekt-instanser objekt-adapter skal starte opp med. For hver instans spesifiseres forvaltningsklasse og eventuelle parametre. Ut fra denne kunnskapen kan en skrive kode (eller generere datafil) for instansiering av objekter ved oppstart (se for øvrig avsnitt 5.2.1). Endelig vil en trenge å vite hvilke permanente objekt-instanser som klienten skal ha tilgang til som persistens-røtter.

6.1.5 Grensesnitt til klienter

Klienter klassifiseres etter hvilke språk (eller protokoller) som brukes. Det siste en trenger å vite er hvilke typer klienter (ihht. denne klassifisering) som skal "koples" til objekt-adapteren. Ut fra dette vil en kunne generere de riktige språkbindinger og linke koden for disse sammen med koden for objekt-adapter. For hver klasse av klienter som er aktuelle vil en ha en "stub-kompilator" for å generere språkbindings-kode.

6.2. Definisjon av permanente objekter

Vi skal her introdusere en enkel notasjon for å spesifisere forvaltnings-klasser. En kan tenke seg at konkrete *PermObj*-klasser og program-kode blir automatisk generert fra en slik notasjon, men på den andre siden, skulle det ikke være spesielt vanskelig å implementere klassene direkte i det språket objekt-adapter-rammeverket er implementert i. Notasjonen kan altså både være et hjelpemiddel for å å beskrive forvaltningsklasser på en implementasjonsuavhengig måte, og den kan være et verktøy for å generere de nødvendige software-komponenter.

Fordeler med en slik notasjon er:

- En kan spesifisere forvaltningsklasser nokså detaljert, uavhengig av implementasjonsspråk og uavhengig av hvordan objekt-adapter-rammeverket ellers er implementert.
- Implementasjonsarbeidet blir enklere (hvis kode kan genereres automatisk fra notasjon).
- En kan unngå en del programmeringsfeil, ved at den som spesifiserer forvaltningsklasser kan se bort fra en del implementasjonsdetaljer (hvis kode kan genereres automatisk fra notasjon).

Vi skal videre i dette avsnittet presentere de enkelte elementer i notasjonen, samt vise i eksempel hvordan konstruksjoner kan avbildes til implementasjonsspråk (Her bruker

vi C++ [Stroustrup91] og forutsetter at rammeverket er designa slik som beskrevet i tidligere avsnitt).

6.2.1 Definisjon av forvaltningsklasse

Man kan definere flere forvaltningsklasser for en gitt applikasjon. Hver forvaltningsklasse deklareres med nøkkelordet *MGMT-CLASS* og et navn. De spesielle egenskapene forvaltningsklassen skal ha, spesifiseres mellom *BEGIN* og *END*. Grammatikken for klasse-definisjon er slik (BNF):

```
<class-definition> ::= MGMT-CLASS <name> = BEGIN <class-body> END ;
```

Det er nødvendig å navngi klassen slik at vi kan referere til den senere. I avbildning til et implementasjonsspråk er det naturlig at navnet inngår i navnet på den tilsvarende datatypen (klasse i C++). Figuren nedenfor illustrerer hvordan en forvaltningsklasse (*Planet*) kan deklareres og hvordan en slik deklarasjon kan svare til en C++ klasse (*Planet_Perm*).

FIGUR 48. Eksempel på forvaltningsklasse

```
MGMT-CLASS Planet =                    ## Definisjon av forv. klasse 'Planet'
BEGIN
   ...
END;


class Planet_Perm : public PermObj     // Eksempel på avbildning til C++
{
   ...
};
```

6.2.2 Dynamisk informasjon

Det vil finnes dynamisk informasjon som er spesifikk for hver enkelt objekt instans. Dette er informasjon som brukes for å identifisere (den persistente) informasjonen objektet er bærer av, uavhengig av aktiviseringer (se for øvrig avsnitt 3.4.2 og avsnitt 5.1.6). Hva slags dynamisk informasjon som skal brukes, kan spesifiseres som en sekvens av par (navn, type), det vil si variable som kan brukes seinere i spesifikasjonen. En kan ha null, en eller flere variabeldeklarasjoner i en forvaltningsklasse. Grammatikken for en variabeldeklarasjon kan skrives slik :

```
<var-decl> ::= VAR <name> : <typename> ;
```

Der <*typename*> er navn på en predefinert verdi-type. Vi har ikke typekonstruktører i notasjonen og det er ikke behov for et stort utvalg typer. Her foreslår jeg å tillate typene *STRING* og *INTEGER*, noe som skulle dekke de fleste behov i en prototype. Figuren nedenfor viser et eksempel på deklarasjon av to variable. Dette vil for øvrig svare til tilsvarende (private) klasse-variable i C++.

FIGUR 49. Eksempel på dynamisk informasjon

```
MGMT-CLASS Planet =
BEGIN
   VAR name   : STRING;
   VAR pos    : INTEGER;
   ...
END;
```

6.2.3 Tjener objekt

Objekter som forvaltes etter shared-server prinsippet (jfr. avsnitt 4.3.1) er avhengig av å ha en referanse til den tjener som skal anvendes for aktivisering. I notasjonen her vil en kunne angi at et tjener-objekt skal anvendes, ved hjelp av nøkkelordet *SERVER*. I tillegg angis navn på forvaltningsklassen til tjener-objektet. En eventuell tjener-deklarasjon vil stå før eventuelle variabel-deklarasjoner.

```
<server-decl>::= <empty> | SERVER : <name> ;
```

I objekt-adapter rammeverket vil spesifikasjon av tjener føre til at den konkrete klassen som skal genereres, skal være subklasse av klassen *Server_Perm* (se avsnitt 5.1.6) hvor referanse til tjener-objekt er definert. Navn på forvaltningsklassen trengs ikke brukes ved generering av kode, men vil være nyttig for typesjekking.

6.2.4 Aktivisering

I ANSAware omgivelser vil vi (i følge avsnitt 5.1.6) ha et sett med tjenester som kan anvendes for aktivisering[2]. For eksempel *Factory* eller *Trader*. Det som må spesifiseres er navnet på tjenesten som anvendes og parametre til denne. Vi skal se på noen alternativer i mer detalj.

- Factory: Parametre er (1) maskinnavn på node hvor kapsel skal aktiviseres, (2) sti i filsystem hvor program befinner seg og (3) navn på fil for programkode.
- Capsule: Parametre er (1) typenavn, (2) streng med argumenter (fritt innhold). For dette alternativet forutsettes det at det er deklarert en tjener (jfr. avsnitt 6.2.3).
- Trader: Parametre er (1) typenavn, (2) navnekontekst og (3) streng med attributt-uttrykk.

Grammatikk for spesifikasjon av aktivisering kan skrives slik:

```
<activation>  ::= ACTIVATOR = <act-service> ;
<act-service> ::= Trader    ( <string> , <string> , <string> )
                | Factory   ( <string> , <string>, <string> )
                | Capsule   ( <string> , <string> )
```

En kan bruke variablene som ble nevnt i avsnitt 6.2.2 i parametre til slike tjenester. Enten direkte eller som del av eksplisitt formulerte tekststrenger. I notasjonen som blir presentert her, skrives tekststrenger med " " og hvis vi inne i en slik streng skriver tegnet '%' foran et variabelnavn betyr det at dette skal byttes ut med innholdet av den navngitte variabelen. Det kan for eksempel spesifiseres slik:

FIGUR 50. Eksempel på spesifikasjon av aktivisering

```
MGMT-CLASS Planet =
BEGIN
   ...
   ACTIVATOR = Trader ("Planet", "/", "Name == '%name' AND Pos == %pos");
   ...
END.
```

Hvis f.eks. name er "Jupiter" og pos er 5, vil attributt-uttrykk strengen evalueres til følgende:

```
"Name == 'Jupiter' AND Pos == 5"
```

2. En naturlig utvidelse vil her være å kunne spesifisere en brukerdefinert aktiviseringstjeneste. Bruker vil da måtte eksplisitt angi ANSAware operasjon som skullle anvendes eller kildekoden direkte.

En spesifikasjon av aktivisering slik vi har beskrevet det her, vil i konteksten av vårt objekt-adapter rammeverk avbildes til en implementasjon av metoden *impl_activate*, som er en abstrakt metode i *PermObj* klassen. Denne implementasjon skal blant annet sørge for å formattere strenger som skal være argumenter og anrope den riktige ANSAware tjenesten for aktivisering. En slik funksjon kunne f.eks. se ut slik:

FIGUR 51. Utdrag fra aktiviserings funksjon

```
int PlanetPerm::impl_Activate(LocalUnit& lu)
{
    static char properties[BUF_LEN];
    ...
    sprintf(properties, "Name == %s AND Pos == %d", name, pos);
    trade(ref, "Planet", "/", properties);// Biblioteks funksjon for trading
    ...                                   // lu vil måtte tilordnes gr.sn.ref returnert fra trader
}
```

6.2.5 Passivisering

For passivisering har vi (som for aktivisering) et sett med tjenester. Vi skal se nærmere på noen alternativer som er aktuelle for ANSAware (se for øvrig avsnitt 5.1.6).

- NONE: Passivisering tillates ikke av det lokale systemet.
- Capsule: Brukes for kapsel-objekter. En anroper kapsel for å be det terminere seg sjøl. Ingen parametre.
- Object: Objektets eget forvaltnings-grensesnitt. En anroper objektet for å be det terminere seg sjøl. Ingen parametre.

FIGUR 52. Eksempel på spesifikasjon av passivisering

```
MGMT-CLASS Planet =
BEGIN
    ...
    PASSIVATOR = NONE;
    ...
END;
```

På samme måten som for aktivisering, vil dette i konteksten her avbildes til implementasjon av metoden *impl_Passivate*, som er en abstrakt metode i *PermObj* klassen.

Passiviserings policy

Passivisering kan skje fra objekt-adapter, enten ved at klient ber om det eller gjennom en form for passiviserings policy. Denne kan være representert ved egne "policy" objekter som sørger for å kalle passiviserings-metodene i de aktuelle proxy-objektene. En slik "policy" kan være så mangt og vi har ikke plass til å gå mer i dybden i dette her.

En "policy" kan spesifiseres i den notasjonen som beskrives her. Innenfor en forvaltningsklasse kan det f.eks. gjøres slik:

```
PASSIVATION AFTER 120;
```

I dette eksemplet passiviseres ganske enkelt objekter hvis de står ubrukt i mer enn 120 sekunder.

6.2.6 Hvordan oppdage passivisering

Hvis objekter passiviseres av det lokale systemet, er det ønskelig at dette blir kjent for objekt-adapter, slik at denne vet at de skal aktiviseres igjen når de trengs (jfr. avsnitt 3.5.1). For temporære objekter har det ikke noen hensikt, fordi disse ikke kan reaktiviseres, og hvis passivisering skjer før klient er ferdig med å bruke dem, er dette

uttrykk for feil (jfr. avsnitt 3.5.3). Objekt-adapter bør ha måter å håndtere slike feil som oppstår og rapportere dem til klient.

For permanente objekter er vi interessert i å oppdage at passivisering skjer. I avsnitt 3.5.1 identifiserte vi monitorering, notifikasjon og management by exception som alternative måter å oppdage. Her er det relevant å understøtte følgende:

- Notifikasjon: Objekt-adapter varsles gjennom et eget grensesnitt når objekt (kapsel) terminerer.
- "Management by exception": Hvis en forsøker anrop mot et terminert objekt, vil anropet feile. I slike tilfeller vil ANSAware kalle en "exception" funksjon hvis denne finnes. Denne funksjonen kan initiere re-aktivisering, sørge for at grensesnittreferanse blir relokert med den nye aktiviseringa og gi beskjed i returverdien om at ANSAware kan forsøke på nytt med operasjons-anrop.
- Objekt-adapter oppdager at tjener-objekt terminerer. Hvis objektet er avhengig av dette tjener-objektet (vil være tilfelle hvis det er en kapsel), kan en slutte at objektet også har blitt passivt (jfr. avsnitt 3.5.3).

Det vi her trenger applikasjons-spesifikk kunnskap om er kun om notifikasjons skal brukes for tidlig oppdagelse av passivisering. De andre alternativene kan implementeres i den applikasjons-uavhengige delen av rammeverket og gjelde alle ANSAware objekter. Skal notifikasjon brukes, angis dette ganske enkelt med nøkkelordet '*NOTIFICATION*' i en forvaltnings-klasse definisjon. Dette vil svare til at kode for mottak av notifikasjoner for den aktuelle objekt-typen inkluderes i objekt-adapter og at notifikasjons-grensesnitt opprettes og gis til *Factory* ved aktivisering.

6.2.7 Permanente instanser

Instansiering av permanente objekter spesifiseres ved hjelp av nøkkelordet PERMANENT. For en instansiering angis navn på objektet, slik at det kan refereres til senere. For en gitt instansiering må en spesifisere navn på forvaltnings-klassen og en må angi verdiene som skal tilordnes de dynamiske egenskapene (variable) som argumenter. For objekter som forvaltes av et tjenerobjekt angis også navn på tjenerobjektet som argument, før eventuelle verdier på parametre.

Nøkkelordet EXPORT angir at objektet (gjennom språkbindinger) skal gjøres tilgjengelig for klienter som persistens-røtter.

FIGUR 53. Grammatikk for insansiering

```
<instance> ::= PERMANENT <name> = <name> ( arglist ) <export>;
<export>   ::= <empty> | EXPORT
<arglist>  ::= <server-name> , arglist
             |   <argument> , arglist
             |   <empty>
```

6.2.8 Eksempel

For å oppsummere hvordan permanente objekter kan defineres, skal vi se på et eksempel. I figuren nedenfor deklareres to forvaltningsklasser. Først *PlanetServer* som tilsvarer en ANSAware-kapsel. For å identifisere en kapsel, trengs informasjon om maskinnavn, sti i filsystemet, programnavn og eventuelle argumenter som gis ved oppstart av kapsel. Kapsel aktiviseres ved hjelp av *Factory* og passiviseres ved hjelp av sitt eget grensesnitt. Når vi bruker *Factory* kan vi også utnytte notifikasjon hvilket blir gjort i dette eksemplet.

Planet er en klasse av objekter som forvaltes av *PlanetServer*-kapsler.

I eksemplet angis også instanser av disse objektene (gis navnene '*pserv*', '*world1*' og *'world2*'). '*pserv*' fungerer som tjener-objekt for *'world1*' og *'world2*'. *'pserv*' blir heller ikke gjort synlig for klienter.

FIGUR 54. Eksempel

```
MGMT-CLASS PlanetServer =
BEGIN
    VAR nodename  : STRING;
    VAR path      : STRING;
    VAR program   : STRING;
    VAR arg       : STRING;

    ACTIVATOR     = Factory (nodename, path, program, arg);
    PASSIVATOR    = Capsule;
    NOTIFICATION;
END;

MGMT-CLASS Planet =
BEGIN
    SERVER     : PlanetServer;
    VAR name   : STRING;
    VAR pos    : INTEGER;

    ACTIVATOR = Capsule ("Planet", "%name, %pos");
    PASSIVATOR = Object;
END;
```

FIGUR 55. Eksempel

```
# Permanente objekter og eksport

PERMANENT pserv  = PlanetServer ("odslab2", "/usr/local/bin", "pserver", "");
PERMANENT world1 = Planet (pserv, "Jupiter", 5)EXPORT;
PERMANENT world2 = Planet (pserv, "Saturn", 6) EXPORT;
```

6. 3. Oppsummering

Vi har i dette kapitlet sett på hvilken informasjon som er grunnlag for definisjon av en konkret objekt-adapter, Dette er (1) det underliggende informasjonssystem, (2) Skjema, (3) forvaltning av permanente objekter, (4) persistens-rot objekter og (5) grensesnitt til klienter.

Vi har sett spesielt på definisjon av persistente objekter og har introdusert en enkel notasjon for å spesifisere forvaltnings-klasser og permanente objekt-instanser. Vi har antydet at denne kan brukes i et verktøy for automatisk generering av den nødvendige software. I vedlegg 5 finnes et eksempel på en konkret *PermObj* klasse for en eksempel-applikasjon beskrevet i avsnitt 9.2.3

Kapittel 7
Prinsipper for språk-binding

Språkbindinga er en software-entitet som overfor klienter tilbyr aksess til objekt-adapter og de objekter som objekt-adapter formidler kontakt med, på en måte som er naturlig for det spesifikke språket. Når en skal konstruere ei konkret språkbinding, må en finne representasjoner i det spesifikke språket av objekt-referanser, operasjoner og verdi-typer (jfr. avsnitt 4.1.2). Språkbindinga skal også realisere semantikken som i avsnitt 4. 4 blir beskrevet som de konseptuelle operasjonene *bind*, *unbind* og *invoke*.

Hvordan dette blir gjort vil være forskjellig, alt etter hvilket språk eller interaksjonsmodell som språkbindinga skal konstrueres for. Vi skal her identifisere en del prinsipper for konstruksjon av språkbindinger, som er gyldig for mange alternative valg av konkret språk. Ei språkbinding til C++ har blitt realisert og tjener her som eksempel på direkte språkbinding.

7. 1. Språk-alternativer

For å belyse og få fram disse prinsippene, skal vi se på hvordan man i praksis kan representere konseptene og hvordan man kan realisere språkbindinger. Da må vi gjøre visse forutsetninger om egenskaper til det språk det skal definere binding for. For det første skal vi se på tilfeller der det er direkte kopling mellom objekt-adapter programmeringsomgivelsen som klienten bruker. For det andre skal vi se på tilfeller der klient og objektadapter er logisk atskilt, og der bindinga skjer via et kanonisk språk, representert ved en protokoll mellom de to entitetene.

7.1.1 Direkte binding

Direkte binding er basert på at koden for objekt-adapter og språkbinding lenkes sammen med koden for klient-program. Disse entitetene kjører da i samme adresserom og en kan utnytte den nære koplinga ved å la språkbindings-komponentene som klient anvender, inneholde direkte pekere til elementer i objekt-adapter. Vi fokuserer i så tilfelle på imperative kompilerende språk som f.eks. C++ [Stroustrup91].

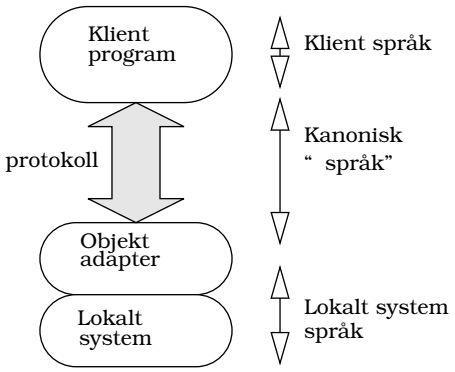
7.1.2 Binding via kanonisk språk (protokoll)

For konfigurasjoner der klient og objekt-adapter er logisk atskilt, kan vi ikke gjøre noen forutsetninger om implementasjonsmåte og hva slags elementer implementasjonen av objektadapter har. All interaksjon mellom klient og objekt-adapter vil skje gjennom et veldefinert grensesnitt, som kan beskrives uavhengig av både klientens språkomgivelse, hvordan objekt-adapter er konstruert og hva slags underliggende applikasjoner klienten skal ha tilgang til.

Vi kan si at språkbindinga skjer til et kanonisk språk. Det vil si et språk som har et minimalt, men tilstrekkelig sett av konsepter for å uttrykke det som er essensielt ved den interaksjon vi er interessert i. Dette skal kunne representere alle sider ved interaksjonen.

Et kanonisk språk er representert ved et grensesnitt som definerer protokollen mellom entitetene. Vi ser av figuren nedenfor at vi har tre nivå når det gjelder språk. Objekt-adapter avbilder mellom klientspråk og det lokale systemets språk via det kanoniske språket. Det vi ser på her er altså bindinga til et kanonisk språk, som igjen kan avbildes til ulike klient-språk.

FIGUR 56. Modell av konfigurasjon med atskilte entiteter



7. 2. Hovedspørsmål om representasjon av konsepter

I følge avsnitt 4.1.2 kan en stille følgende hovedspørsmål om hva en må ta stilling til når en skal lage ei språkbinding:

- Hvordan skal objekt-referanser representeres i det aktuelle språket?
- Hvordan skal operasjoner representeres?
- Hvordan verdi-typer representeres?

7.2.1 Representasjon av objekt-referanser

Objekt-referanser er spesielle verdier som objekt-adapter tilbyr klientprogram gjennom språkbindinga (se for øvrig avsnitt 4.4.1). Ei språkbinding vil typisk ha egne datatyper (verdi-typer) for objekt-referanser. Hver objekt-referanse skal unikt identifisere et objekt. Vi tillater en spesielt identifiserbar verdi som ikke identifiserer noe objekt i det hele tatt. Vi kan kalle denne for NIL.

Det vil være en nær sammenheng mellom manipulering av objektreferanser fra klient-program og de konseptuelle operasjonene *bind, unbind* og *invoke* (jfr. avsnitt 4.4.2). For alle objekt-referanser skal vi kunne utføre prosedyrer som svarer til disse konseptuelle operasjonene. Hvordan dette manifesterer seg i ei bestemt språkbinding, er forskjellig alt etter hvordan objekt-adapter og klient er koplet, og alt etter hvilket språk som blir anvendt. Vi skal her se på de to alternativene: Direkte binding versus binding via kanonisk språk.

Direkte binding

For konfigurasjoner der klient og objekt-adapter er samlokalisert, er det aktuelt å la objekt-referanser være representert med datatyper som er, eller inneholder peker direkte til proxy-objekt. (NIL-referanser vil da f.eks. tilsvare en NULL-peker). Det vil måtte definerers tilhørende prosedyrer for å realisere objekt-referanse semantikk. I språk med statisk typesjekking kan man for hver objekt-type definere en egen objekt-referanse-type for å kunne gjøre statisk typesjekking av objekt-referanser mulig. I C kan det for eksempel gjøres slik (Objekt-referanser er representert som pekere til proxy-objekter):

FIGUR 57. Eksempel på representasjon av objekt-referanser C

```
typedef Proxy *AccountRef;      /* Objekt referanse type */

AccountRef ref_Up(AccountRef); /* Svarer til bind */
void ref_Down(AccountRef);     /* Svarer til unbind */
void credit(AccountRef, Float);/* Svarer til invoke "Credit" */
...
```

I objektorienterte språk som C++ kan objekt-referanser representeres på en naturlig måte ved at operasjoner og identifikator/peker integreres i en klasse. Ved hjelp av klassebegrepet kan vi definere objekt-referanser som oppfører seg som en slags "smarte" pekere. Ved å overlade operatorer for konstruksjon, tilordning og sletting, vil *bind*/*unbind* semantikken (i motsetning til eksemplet over), være transparent for klient-programmerer. Dette kan forenkle programmering og redusere mulighetene for feil. Subklasse-mekanismen kan utnyttes på to måter: For det første ved at klassevariable, og forvaltnings-operasjoner som gjelder alle objekt-referanser kan defineres og implementeres en gang for alle og arves av konkrete objekt-referanse klasser. For det andre kan subtype-relasjoner i lokale skjema, avbildes til subklasse-relasjoner i språkbindinger.

FIGUR 58. Eksempel på representasjon av objekt-referanser i C++

```
class ObjectRef {
protected:
   Proxy* proxy;
public:
};

class AccountRef : public ObjectRef {
public:
   AccountRef ();                          // Konstruktør for NIL-referanser
   AccountRef (AccountRef);                // Kopi konstruktør. Svarer til bind
   ~AccountRef ();                         // Destruktør. Svarer til unbind.
   AccountRef& operator = (AccountRef);    // Tilordnings operator (unbind + bind)

   void credit (Float);                    // Invoke "credit"
   ...
};
```

Kanonisk språk

Ved språkbindinger via et kanonisk språk, kan objektreferanser være representert med en spesiell objekt-identifikatortype. På klient side vil objekt-identifikatorer ikke ha noe eget semantisk innhold, men vil bli oppfattet som en serie med bits, eller et vilkårlig tall. En objektidentifikator kan være en verdi generert av det lokale systemet eller av objekt-adapter. På objekt-adapter side vil OID bli avbildet til riktig proxy-objekt.

Dette svarer til objekt-referanse begrepet i CORBA [OMG.91.12.1]. Her vil objekt-adapter asossiere en portabel[1] objektreferanse med grensesnitt-beskrivelse, identifikasjon av objektets implementasjon (tjener) og en mindre portabel identifikasjon av objektet som bare kan tolkes av objektets implementasjon.

Permanent versus temporær identitet

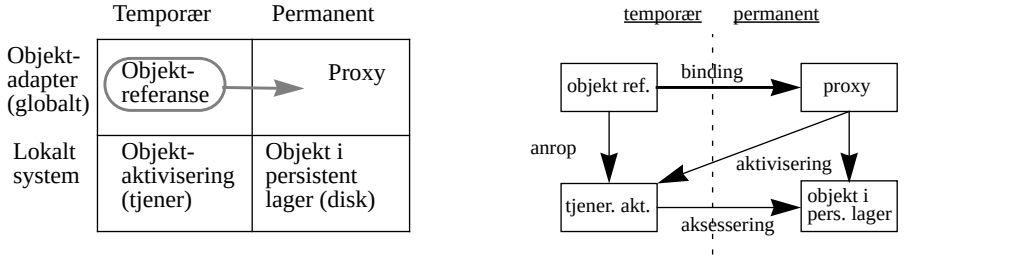
Det er relevant å spørre seg om hvorvidt språkbindingas form for objektreferanser kan være en permanent form for identifikasjon. Det vil si om de kan være gyldig over flere aktiviseringer av klient-programmet. I C++ eksemplet ovenfor (figur 58), vil en instans av klassen *AccountRef* bare gjelde innenfor aktiviseringa av programmet, eller innenfor ei prosedyre eller blokk i programmet. Den kan dermed ikke regnes som permanent. Skal vi f.eks. lagre referanser til objekter over flere aktiviseringer, må vi finne en annen form for identifikasjon.

1. Skal være portabel over ulike ORBer.

I avsnitt 4. 4, defi nerer vi en konseptuell *bind*-operasjon som binder en objekt-referanse til et objekt. Det betyr at vi kan ha temporære referanser til persistente objekter. Det betyr også at vi kan ha flere temporære referanser til samme objekt. Ei eksplisitt binding forutsetter en eller annen form for identifi kasjon av objektet. Dette kan f.eks. være en annen objektreferanse eller et tekstlig navn som ved instansiering av objekt-adapter er gjort kjent (eksportert) for klienter (jfr. avsnitt 5.2.2). Et slikt navn vil være en permanent identifi kasjon.

Bind/unbind semantikken fra avsnitt 4. 4 er basert på en antakelse om at objektreferanser er temporær identifi kasjon og at disse kan " bindes" til persistente objekter Dette har preget beskrivelsen hittil og passer for klient-språk-konstruksjoner med innebygget *bind/unbind* semantikk som f.eks. C++ klassen i fi gur 58 ovenfor. Gjennom binding kan vi altså ha temporære referanser til rot-objekter. Figuren nedenfor (jfr. avsnitt 3.4.3 og avsnitt 4. 5) illustrerer hvordan vi har oppfattet objektreferanser i forhold til andre former for identifi katorer/identifi kasjon vi opererer med.

FIGUR 59. Klassifi sering og sammenheng mellom objekt-representasjoner



I kanoniske språk behøver det ikke nødvendigvis være slik (som fi guren ovenfor antyder). Her opererer vi med objekt-identifi katorer som vi ikke kan anta noe om semantikken til. Slike identifi katorer er defi nert utenfor og eksisterer uavhengig av de konkrete språk klientprogrammer skrives i. Kanoniske objektreferanser kan både være temporær og permanent.

Med permanente objektreferanser, vil en konseptuell *bind*-operasjon (i et kanonisk språk) ikke ha samme mening som med temporære referanser. De synes unødvendig fordi binding av objektreferanse er permanent og gjelder så lenge objektet eksisterer. Bruk av *bind*-operasjon vil da heller måtte oppfattes som deling av ei eksisterende (permanent) binding[2]. Deling kan også gjøres ved å utveksle objektreferanser direkte mellom klienter.

7.2.2 Representasjon av operasjoner

I eksemplene på C og C++ representasjon av objekt-referanser i avsnittet ovenfor, så vi at operasjonene i det lokale systemets grensesnitt blir representert som egne funksjoner og metoder. Hvis klient og språkbinding er samlokalisert og hvis vi har direkte kopling mellom disse, vil disse funksjonene eller metodene kunne være implementert direkte som stub's.

I konfi gurasjoner med atskilte entiteter og et kanonisk språk, vil referanse til stub-funksjonene være indirekte. Dette kan skje ved at hver operasjon tilordnes en identifi -kator. En funksjonsidentifi kator kan bestå av en tekstlig representasjon av type- og funksjonsnavn eller den kan være et tall (surrogat). Denne vil på objekt-adapter siden måtte avbildes til operasjoner i det lokale system, f.eks. gjennom stub-funksjon(er).

2. Deling av referanser i et kanonisk språk, kan i et klient-programmeringsspråk oppfattes som binding av lokale variable/objekt (via kanoniske referanser) til de aktuelle objekter.

7.2.3 Representasjon av verdi-typer

Hver verdi-type i IDL, vil ha en korresponderende type i det språket som klientprogrammet skrives i. Konstruerte typer og type-alias vil ha tilsvarende definisjoner i klient-språket. En typedefinisjon som dette i IDL:

```
Vector : TYPE = ARRAY [3] OF INTEGER;
```

Vil f.eks. ha en slik tilsvarende type i C/C++:

```
typedef int [3] Vector;
```

For språkbinding til C, kan ANSAwares egen språkbinding brukes direkte siden denne er til C. Her blir hver IDL-type representert som en C-type. For atomiske typer tilbyr ANSAware et predefinert sett av typedefinisjoner. For konstruerte typer (brukerdefinerte) vil stub-kompilator generere de nødvendige typedefinisjoner i C. For språkbinding til C++ kan også C-bindinga brukes siden C++ representerer et supersett av C, men for enkelte typer kan bruk av verdier gjøres langt enklere ved å anvende C++ klasser som skjuler representasjon og implementasjon av abstraksjoner som *SEQUENCE* og *CHOICE*. For eksempel kan en ha en parametrisert klasse for sekvenser (lister) med elementer av en vilkårlig type. En typedefinisjon som dette:

```
XSeq : TYPE = SEQUENCE OF X;
```

kan da avbildes til noe slikt:

```
typedef Sequence<X> Xseq;
```

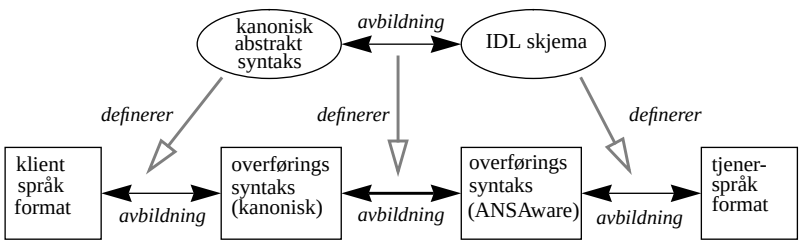
Ei språkbinding vil også måtte definere avbildnings-funksjoner. Disse brukes av stub'ene for å oversette mellom ANSAwares representasjon av verdi-typene og den representasjon vi har valgt i ei språkbinding/klient-grensesnitt. Dette tilsvarer marshalling (se avsnitt 2.7.3).

Verdi-representasjon i kanonisk språk

I det kanoniske språk vil vi kunne beskrive argumenter og resultater ved hjelp av et sett med atomiske verdityper og konstruktører. Sammenlikner vi med presentasjonslaget i OSI-RM [HeSh90], kan vi si at dette svarer til en abstrakt-syntaks som definerer verdi-formatet uavhengig av de konkrete programmeringsspråk som brukes og uavhengig av overføringsformatet. For en gitt abstrakt syntaks vil vi også måtte definere en overførings-syntaks som spesifiserer på hvilken form verdiene skal overføres mellom klient og objekt-adapter.

IDL-skjema i ANSAware kan også oppfattes som abstrakt syntaks. En abstrakt syntaks vil avbildes til typer i et konkret programmeringsspråk. Ei språkbinding vil også måtte ha funksjoner som avbilder mellom verdier i disse typene til sekvenser av oktetter som svarer til overføringssyntaks (marshalling). Vi vil for hver av de typene vi kan beskrive verdier som, definere overføringsformat og avbildingsfunksjoner. Det ei språkbinding definerer er ei avbildning mellom to former for abstrakt syntaks, samt funksjoner for å konvertere mellom aktuelle verdi-formater som svarer til to (muligens ulike) overførings-syntakser. Figuren nedenfor illustrerer dette.

FIGUR 60. Avbildninger



7. 3. Prinsipper for konstruksjon av ANSAware språkbindinger

Vi skal nå se nærmere på hvordan vi kan konstruere konkrete språkbindinger for objektadaptere for ANSAware. Vi skal se på hva et IDL-grensesnitt består av, hva dette kan avbildes til i ei språkbinding og hvilke komponenter en implementasjon av ei språkbinding består av. Vi skal deretter se på hvordan ei språkbinding kan realiseres i praksis, for henholdsvis direkte kopling til C++ og et kanonisk språk.

7.3.1 Grensesnitt

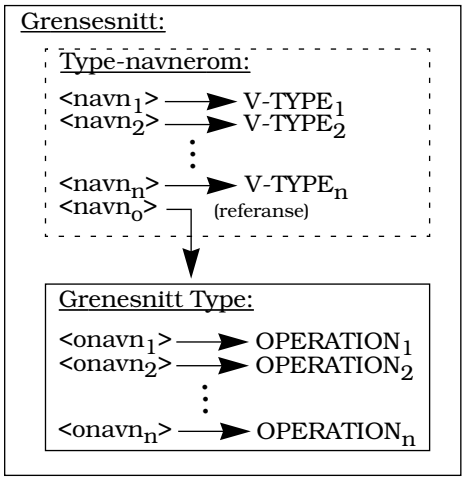
Språkbindinger vil kunne utledes fra skjema for applikasjonen. Skjema består av en eller flere grensesnittspesifikasjoner. Enheten for definisjon av språkbindinger er et grensesnitt. Språkbindinger genereres altså per grensesnitt, og for applikasjoner med flere grensesnitt, kan samme algoritme for generering utføres flere ganger. Når vi nå skal se på hva koden som genereres skal bestå av, skal vi ta utgangspunkt i et grensesnitt og klargjøre hva en IDL-spesifikasjon egentlig definerer, før vi går inn på hvordan dette skal representeres i et gitt språk.

Hva en IDL spesifikasjon definerer

En grensesnittspesifikasjon definerer både et sett med verdi-typer og et sett med operasjons-signaturer for aksess til objekter som grensesnittet definerer atferden for. Et grensesnitt består av:

- En mengde navn på verdi-typer. En av disse navngir en grensesnitt-referanse-type. Vi kan på en måte si at grensesnittet definerer et navnerom for de verdi-typer en har behov for i grensesnittet.
- En mengde operasjons-signaturer. Typen til grensesnittet (og dermed også typen til grensesnittreferanser) er definert ut fra operasjons-signaturene.

FIGUR 61. Grensesnitt spesifikasjon



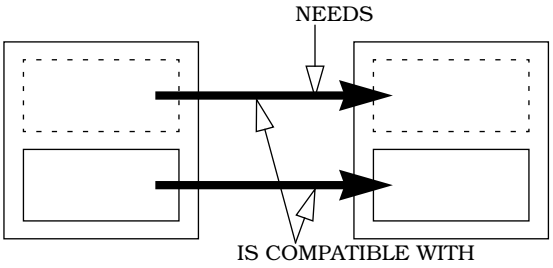
Relasjoner mellom grensesnitt

Mellom grensesnittspesifikasjoner kan en definere to typer relasjoner, nemlig inklusjon av typenavn og arv av operasjoner. Vi har altså følgende:

- Inklusjon av navnerom (for verdi-typer) fra et grensesnitt i et annet. Man spesifiserer dette i IDL ved hjelp av nøkkelordet 'NEEDS'. Dette anvendes når et grensesnitt trenger (verdi) typedefinisjonene fra et annet.

- Inklusjon av navnerom (som ovenfor)[3], pluss inklusjon (arv) av operasjons-signaturer fra et grensesnitt i et annet. Altså den andre grensesnitt-typen blir en subtype av den første. Man spesifiserer dette i IDL ved hjelp av nøkkelordet 'IS COMPATIBLE WITH'.

FIGUR 62. Relasjoner mellom grensesnitt



7.3.2 Representasjon av grensesnitt i den aktuelle programmeringsomgivelse

Representasjon av et grensesnitt i ei språkbinding vil bestå av fire deler:

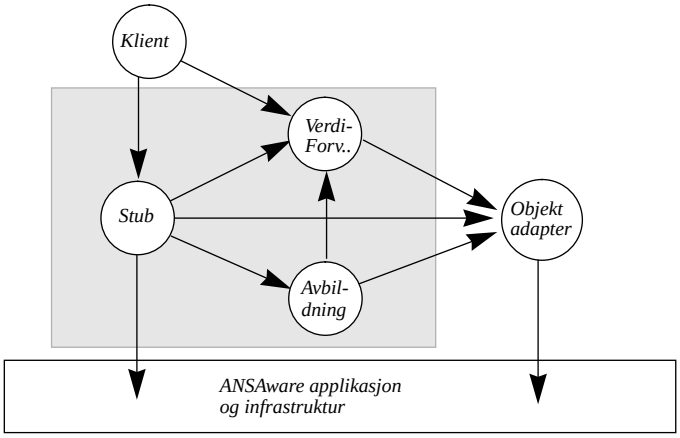
- Definisjoner for objekt-referanser, verdi-typer og eventuelt operasjoner . Dette for å gjøre konseptene tilgjengelig for bruk i klient programmer. Dette er språkbindingas representasjon av skjema for et grensesnitt.
- Stub-prosedyre(r) for avbildning av operasjonsanrop.
- Prosedyre(r) for avbildning av verdier (marshalling).
- Prosedyrer for forvaltning (kopiering, sletting, søppelsamling mm..) av objektreferanser og andre verdi-typer. Spesielt gjelder dette forvaltning av objektreferanser som må inkludere korrekt *bind/unbind* semantikk.

Figuren nedenfor illustrerer hvilke komponenter en implementasjon av ei språkbinding vil bestå av og forholdet mellom disse. Her finner vi stub-prosedyrer, avbildningsprosedyrer og verdi-forvaltnings-prosedyrer og det går fram hvordan disse samarbeider innbyrdes og med klientprogram og objekt-adapter:

- Stub-prosedyre(r) kalles av klient program (direkte eller via en dispatcher funksjon som avbilder fra en operasjons-identifikator til riktig stub).
- Avbildnings-prosedyre(r) anvendes av stub for å konvertere mellom klienten og det lokale systemets format for representasjon av verdier som brukes i argumenter og resultater.
- Verdi-forvaltning anvendes av klient, stub- og avbildnings-funksjoner for å implementere korrekt semantikk ved instansiering, kopiering eller sletting av verdi-type objekter.
- Objekt-adapter anvendes av stub-funksjonene for å konvertere fra objekt-referanser (proxy-objekter) til ANSAwares grensesnittreferanser, av avbildnings-funksjoner for både å konvertere fra objekt-referanser til grensesnittreferanser og omvendt (opprette nye proxyer hvis det er nødvendig) og av Forvaltnings-funksjoner for å håndtere referanse-teller for objekt-referanser (*bind/unbind* semantikken).

3. Dette er nødvendig fordi operasjons-signaturer avhenger av typer i navnerommet. Operasjoner som "arves" må "ta med seg" typedefinisjonene.

FIGUR 63. Komponenter i språkbinding implementasjoner



Definisjoner

For definisjon av grensesnitt, vil en måtte spørre på hvilken måte språkomgivelsen har fasiliteter for å definere og manipulere med navnerom. I språket C vil enheten for definisjon av grensesnitt være ei kildefil (headerfil). Det er eneste mulighet. Grensesnittets navn vil svare til navn på den headerfil hvor typene til det aktuelle grensesnittet er definert. Inklusjon av grensesnittenes navnerom svarer til inklusjon av headerfiler. Dette er også måten det er gjort på i ANSAware. Her vil uttrykk som

```
NEEDS xxxx ;
```

i IDL eller

```
! USE xxxx ;
```

i PREPC, svare til følgende i C (*xxxx.h* er da generert fra grensesnittet *xxxx*):

```
#include "xxxx.h"
```

En mulig problem med å bruke headerfiler slik som dette, er at alle typer som blir inkludert inngår i det globale skopet i programmet som de blir inkludert i. I store program som inkluderer mange headerfiler, er det fare for navnekollisjoner. Det vil f.eks. bli problemer hvis to ulike grensesnitt definerer hver sin type og disse tilfeldigvis har samme navn. Dette kan avhjelpes ved f.eks. å innføre den regel at typenavn prefiksses med navnet på det grensesnitt de er definert i. En type '*Person*' i grensesnittet '*Bank*' kan i språkbindinga f.eks. få navnet '*Bank_Person*' (en tilsvarende prefiksing er brukt i CORBAs språkbinding til C [OMG.91.12.1]).

I C++ er det mulig å definere navnerom med finere granularitet ved hjelp av klassebegrepet[4]. Et navn i C++ som er definert innenfor en klasse vil kun være synlig innenfor klassen sjøl og innenfor subklasser. I det globale skopet er det likevel mulig å referere til navn innenfor klasser ved å bruke klassenavn og operatoren '::' som prefiks til navnet. Det er da mulig i C++ å representere navnerommet til et grensesnitt som en klasse. Det vil være en klasse som ikke har metoder eller variable, men bare en rekke typedefinisjoner og definisjon av en egen objektreferanse klasse (se f.eks. avsnitt 7.2.1). Inklusjon av navnerom vil da kunne realiseres ved hjelp av subklassing. Figuren nedenfor viser et eksempel på dette. Her inkluderer grensesnittet XXX navn fra grensesnittet YYY.

4. Nyere spesifikasjoner av C++ (ANSI X3J16) har et eget nøkkelord 'namespace' som vil være en mer naturlig måte å avgrense navnerom [Penn94]. Dette anvendes i forslag til språkbinding for CORBA [OMG 93.9.2]

FIGUR 64. Eksempel på representasjon av grensesnitt i C++

```
class IFC_XXX : public IFC_YYY // Navnerom
{
public:
    typedef t1 n1;
    typedef t2 n2;
    ...
    typedef tn nn;

    class XXXRef {              // Objekt referanse klasse
    public:
       ...
    };
};
```

Stub funksjoner

Her kan vi tenke oss en stub for hver operasjon i grensesnittet, eller en enkelt generisk stub som kan gis parametre som identifiserer den spesielle operasjonen og angir operasjonens egenskaper (jfr. dynamiske anrop i CORBA [OMG.93.12.1]). Stub-funksjonene vil ha følgende oppgaver:

- Omforme argumenter til det formatet ANSAware forventer (ved hjelp av avbildningsfunksjoner).
- Gjøre ANSAware anrop.
- Omforme resultater fra det formatet ANSAware bruker til språkbindingas format.

Ved implementasjon er det to alternativer. En kan la den stub-funksjon, eller de stub-funksjoner vi genererer for ei språkbinding, erstatte de stub-funksjoner som normalt genereres av ANSAware. Disse vil da gjøre en tilsvarende jobb, men marshalling-funksjoner må tilpasses språkbindinga. Alternativt kan våre stub-funksjoner bruke ANSAwares stub-funksjoner. Vi oversetter da fra språkbindingas dataformat til ANSAwares språkbinding (og omvendt), der formatet er forskjellig.

Avbildnings-funksjoner

Som det ble antydet i avsnittet over, vil avbildningsfunksjonene enten være marshalling-funksjoner, eller det kan være funksjoner som avbilder mellom representasjoner i to ulike språkbindinger. Avbildnings-funksjoner vil bli brukt av stub's, og der man har konstruerte typer, vil de bruke hverandre.

Forvaltnings funksjoner

For oppretting, kopiering og sletting av datastrukturer i språket som representerer verdityper, kan vi trenge rutiner for å:

- Allokere lager (ved kopiering og oppretting)
- Deallokere lager for datastrukturer som det ikke er mer behov for.
- Realisere *bind*/*unbind* semantikk for objekt-referanser. Dette kan gjøres ved å manipulere referansetellere for de tilhørende proxy-objekter. Objekt-adapter skal søppelsamle proxyer som det ikke er mer behov for.

Helst bør språkomgivelsen sjøl sørge for lagerforvaltning og la dette være mest mulig transparent for applikasjonsprogrammerer. Slik er det ikke alltid. Spesielt i C og C++, vil det være nødvendig å skrive kode for forvaltning av lager.

I språk som C++ vil enkelte verdi-typer være representert med klasser. Det kan også tenkes at slike klasser har egne metoder for forvaltning av lager, pekere og liknende.

Lagerforvaltning og bind/unbind semantikk kan gjøres transparent for bruker av klassene ved å overlade operatorene for tilordning, konstruksjon og sletting.

7.3.3 Ei språkbinding til C++

Vi skal her presentere et eksempel på språkbinding til C++. Vi skal ta for oss enkle grensesnitt-definisjoner og se hvordan dette representeres i C++. Dette har vi også realisert som en prototype implementasjon (se avsnitt 9.2.2).

Vi har to grensesnitt: *Account* (konto) og *Bank*. I *Bank*-grensesnittet kan vi tenke oss en enkelt operasjon *Access* som returnerer en grensesnittreferanse for en konto. I Konto-grensesnittet har vi operasjoner for kreditering, debitering og opplysning om saldo. Vi definerer en egen verdi-type *Status* som angir om transaksjonen har vært vellykket eller ikke.

FIGUR 65. Eksempel på grensesnitt-spes.

```
Account : INTERFACE =
BEGIN
    Status : TYPE = {ok, insufficientFunds, systemFailure};

    Credit : OPERATION [amount: REAL] RETURNS [Status];
    Debit  : OPERATION [amount: REAL] RETURNS [Status];
    Saldo  : OPERATION [] RETURNS [REAL];
END.

Bank : INTERFACE =
NEEDS Account;
BEGIN
    Access : OPERATION [n : AccNumber] RETURNS [AccountRef];
    ...
END.
```

Definisjoner - Objekt referanse klasse

Når vi nå skal se hvordan dette avbildes til definisjoner i C++, forutsetter vi at det allerede er definert en superklasse for alle mulige objekt-referanser. Denne definerer hva som er felles, nemlig at hver objektreferanse har en peker til et proxy-objekt. Vi tar med metoder for å sette denne pekeren og for å lese den.

FIGUR 66. Superklasse for alle objekt-referanse klasser

```
class ObjectRef
{
public:
    Proxy* proxy()          {return _proxy;}
    void set_Proxy(Proxy* p){_proxy = p;}

protected:
    Proxy* _proxy;
};
```

Hvis vi tar *Account*-grensesnittet ovenfor, vil dette kunne avbildes til følgende definisjoner:

- En *typedef* for verdi-typen *Status*. I C++ er denne representert som en *enum* type.
- En klasse *AccountRef* som representerer objekt-referanser til konto-objekter. Denne er en subklasse av *ObjectRef*. Vi ser at den har metoder som representerer hver av operasjonene og metoder for oppretting og sletting av objektreferanser.

I en språkbindings-implementasjon vil disse definisjonene være i ei enkelt headerfil. Det vil være ei slik headerfil for hver ANSAware grensesnitt-type. Der et grensesnitt i IDL refererer til et annet ved nøkkelordet '*NEEDS*', vil dette føre til inklusjon av header-

fi lerHvis '*IS COMPATIBLE WITH*' brukes fører dette altså både til at headerfi ler inkluderes og at objekt-referanse klassen blir subklasse av en annen. Hvis vi f.eks. defi nerte et grensesnitt *SpecialAccount* som var kompatibelt med *Account* vil det bli generert en klasse *SpecialAccountRef* som var subklasse av *AccountRef*.

FIGUR 67. Eksempel på objekt-referanse klasse

```
typedef enum {ok, insufficientFunds, systemFailure}Status;

class AccountRef : ObjectRef
{
public:
    AccountRef(Proxy*)      {ref_Up(p); set_Proxy(p);}
    AccountRef(AccountRef& r){set_Proxy(r.proxy(); ref_Up(proxy();}
    ~AccountRef()           {ref_Down(proxy());}
    ...

    Status Credit (Float);
    Status Debit (Float);
    Float Saldo ();
};
```

Verdi-forvaltnings-funksjoner

I dette eksemplet har vi bare egne verdi-forvaltnings funksjoner for objekt-referanser. Hvis vi hadde verdi-typer som var representert som klasser, kunne det være aktuelt med tilhørende metoder for lager allokering og deallokering. De funksjoner vi har for *AccountRef* er følgende:

- Konstruktør som tar peker til proxy som argument. Dette oppfatter vi som binding til objekt representert ved en proxy. Vi ser av fi guren ovenfor at metoden øker referanseteller til proxy-objektet og setter peker til proxy inn i referanse objektet.
- Konstruktør som tar en annen objektreferanse som argument. Denne gjør tilsvarende med peker til proxy som den leser fra den andre objektreferansen.
- Destruktør som teller ned referanseteller til proxy-objektet (oppfattes som *unbind*).

Stubs

For hver operasjon i grensesnittet skal det altså defi neres en stub-funksjon. For operasjoner som er "arvet" fra andre grensesnitt kan også stub for "arvet" operasjon anvendes. I C++ ordner kompilator slik binding til funksjoner, siden dette språket understøtter subklassing. I dette eksemplet er Account-grensesnittets metoder *Credit*, *Debit* og *Saldo* implementert som stub-funksjoner.

Avbildning

Vi trenger avbildningsfunksjoner for hver verdi-type som kan være argumenter og resultater. Disse kalles av stubene. For hver type *T* kan vi defi nere følgende:

- En funksjon *marshall_T* som skriver en verdi av typen *T* til et buffer egnet for overføring.
- En funksjon *unmarshall_T* som representerer den motsatte avbildninga.

For atomiske typer som *Float* vil marshall/umarshall funksjoner være predefi nert. For konstruerte typer, vil en måtte generere funksjoner.

Marshalling av objekt-referanser er litt spesielt. I ANSAware-omgivelsene er det kun grensesnittreferanser som brukes. Det vil si at objekt-referanser må omformes til grensesnittreferanser og omvendt. Figuren nedenfor viser eksempel på marshalling og unmarshalling av objektreferanser. Ved marshalling, trekker vi først proxy-objektet ut av objektreferansen. Deretter ber vi proxy-objektet om å få objektets aktivisering og får

da en lokalenhet (klassene lokalenhet, proxy og meta er beskrevet i avsnitt 5. 1). Lokalenheten inneholder en grensesnittreferanse og det er bare denne som skal marshalles. Ved unmarshalling, får vi en grensesnittreferanse fra ANSAware omgivelsen. Denne oppretter (eller finner) vi en proxy for ved hjelp av meta-klassens metode *activate*. Proxy-objektet settes inn i objektreferansen og referanseteller økes med en (*bind*).

FIGUR 68. Marshalling av objekt-referanser

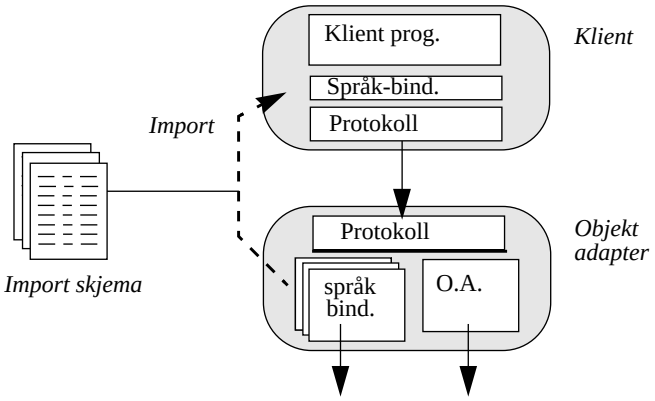
```
int marshall_AccountRef(ansa_BufferLink buf, AccountRef *obj)
{
    int st = 0;
    Proxy *p = obj->proxy();
    ANSA_LU& lunit = (ANSA_LU&) p->get_Activation();
    st = marshall_ansa_InterfaceRef(buf, (ansa_InterfaceRef*) lunit);
    return st;
}

int unmarshall_AccountRef(ansa_BufferLink buf, AccountRef *obj)
{
    int st = 0;
    ansa_InterfaceRef ref;
    if (!(st = unmarshall_ansa_InterfaceRef(buf, &ref))) {
        obj->set_Proxy(META->activate((ANSA_LU) ref));
        ref_Up( obj->proxy() );
    }
    return st;
}
```

7.3.4 Prinsipper for kanonisk språkbinding

Figuren nedenfor illustrerer hvordan en kanonisk språkbinding er representert ved en protokoll mellom to atskilte entiteter, nemlig klient og objekt-adapter. På objekt-adapter side har vi språkbinding til det formatet som kreves i protokollen og på klientside kan en ha språkbinding til det spesielle språket klientprogram skrives i.

FIGUR 69. Klient, språkbinding og objektadapter



Eksport/import skjema

Figuren over illustrerer også at klient-siden importerer skjema-informasjon for å få nødvendig kjennskap til hvilke objekt-grensesnitt (i det lokale systemet) en har med å gjøre. Her gis informasjon om hvilke operasjoner en kan anrope og hvilke argumenter og resultater som forventes til operasjonene, altså en kanonisk abstrakt syntaks for applikasjonen (jfr. avsnitt 7.2.3). Grensesnittet (protokollen) mellom komponentene vil dermed kunne spesifiseres og realiseres uavhengig av applikasjoner. Vi skal se litt på hvordan dette kan gjøres.

Protokoll

En protokoll kan bestå av elementer som mer eller mindre direkte tilsvarer b*ind, unbind* og *invoke* operasjonene fra avsnitt 4. 4. Man kan f.eks. i tillegg ha elementene *Session_Start* og *Session_End* (som antydet i avsnitt 5.2.4). Dette for på en mer effektiv måte å kunne kommunisere *unbind*-semantikken. Det vil si at bruk av *Session_End* er ekvivalent med *unbind* på alle objekter som er bundet til siden siste bruk av *Session_Start*.

7. 4. Oppsummering

I dette kapitlet har vi diskutert prinsipper for språkbindinger. Vi har sett på to alternative konfi gurasjoner, nemlig direkte binding, hvor kode for objekt-adapter/språkbinding og klient-program kjører i samme prosess (f.eks. C++) og binding via kanoniske språk hvor klient og objekt-adapter er logisk atskilte entiteter. Et kanonisk språk er representert ved et grensesnitt som defi nerer protokollen mellom entitetene.

I lys av disse to alternativene har vi diskutert hovedspørsmålene ved språkbindinger, nemlig hvordan objekt-referanser representeres, hvordan operasjoner representeres og hvordan verdi-typer representeres. Objekt-referanser er egne verdi-typer som unikt identifi serer objekter. Disse kan også ha verdien NIL. For direkte binding, vil disse gjerne være, eller inneholde peker til proxy-objekt. Man defi nerer gjerne egne datatyper med en peker-liknende semantikk. I C++ er klassebegrepet egnet til å defi nere objekt-referanse typer.

For binding til kanonisk språk opererer man gjerne med objekt-identifi katorer hvor semantikk ikke er kjent for klient-programmer. Her kan man også tenke seg at objekt-referansre er permanente. Tidligere har vi forutsatt at de er temporære (varer bare så lenge de er defi nert klient-programmet).

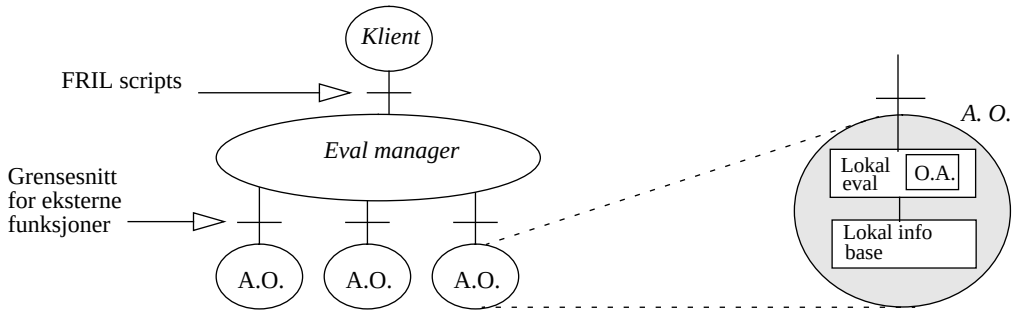
Vi har diskutert hvordan en kan konstruere konkrete språkbindinger for ANSAware applikasjoner. Vi vet at disse kan genereres fra IDL-skjema (per grensesnitt). En IDL spesifi kasjon defi nerer et navnerom med defi nisjon av verdi-typer og et sett med operasjonssignaturer som defi nerer grensesnitt-typen.

En konkret språkbinding vil inneholde (1) defi nisjoner (representasjon av skjema), (2) Stub-prosedyrer, (3) marshalling/unmarshalling prosedyrer og (4) verdi-forvaltnings prosedyrer. Vi har også presentert ei språkbinding til C++ (og realisert denne som en prototype implementasjon). Til slutt har vi kort nevnt noe om kanoniske språkbindinger, noe vi skal komme tilbake til i neste kapittel. Der ser vi spesielt på språkbinding til FRIL (som er et eksempel på kanonisk binding).

Kapittel 8
Binding til FRIL

Den komponenten i et FRIL-programmeringssystem som evaluerer programmer er eval-manager. (evaluerings-forvalter). Denne er tenkt realisert som en distribuert tjeneste, bestående av en eller flere sentrale evalueringstjenere, pluss en eller flere lokale evalueringstjenere (jfr. avsnitt 2. 6). Vi skal ikke fokusere på problemstillingene rundt distribuert evaluering av FRIL-programmer, men ganske enkelt ta for oss grensesnittet mellom eval-forvalter og applikasjonsobjekt. Vi kan skissere en forenklet modell: Vi har en klient, en evaluerings-tjeneste og et sett med applikasjonsobjekter, det vil si lokale informasjonssystemer som er " bundet" til evalueringstjenesten via lokal eval./objekt adapter. Klienten genererer FRIL-programmer som eval-manager evaluerer blant annet ved å anrope eksterne funksjoner via grensesnittene til applikasjonsobjektene.

FIGUR 70. FRIL-Evaluering



8. 1. Representasjon

Før vi går i detalj hvordan grensesnittet kan realiseres, er det nødvendig å diskutere hvordan konseptene objekt-referanse, operasjon og verdi skal representeres i FRIL og i grensesnittet mellom komponentene.

8.1.1 Objektreferanser

FRIL har en egen objekt-identifikator type (*OID*), som er dette språkets form for objekt-referanse. For FRIL-programmer vil denne oppfattes som en atomisk verdi-type, det vil si den er ikke satt sammen av flere andre typer (dette vil være annerledes i den interne representasjon av OIDer). OID må også være gyldig identifikasjon for objekt så lenge objektet eksisterer, eller den må være gyldig så lenge sesjonen varer, hvis det er snakk om temporær identitet. (jfr. avsnitt 2. 5 og avsnitt 7.2.1).

OID struktur på det globale nivå

Følgende må kunne avledes fra en global OID:

- Applikasjonsobjektets (lokal eval/objektadapter) lokasjon.
- De lokale data elementer.

For å oppnå det første kravet kan man tenke seg at en identifikasjon av applikasjonsobjekt inngår som en del av globale OIDer (jfr. f.eks. [Elia93b]). Dette kan for eksempel være et navn som av evalueringssystemet avbildes til den til enhver tid gjeldende lokasjon. Man kan her f.eks. tenke seg en global katalog som representerer ei slik avbildning. Når det gjelder det andre kravet er det opp til applikasjonsobjektet, å generere OID og sørge for avbildning til det egentlige objektet.

I [Elia93b] skisseres også muligheten for såkalte heterogene OIDer. Her tenker man seg at OIDer består av en global del og en lokal del. Den lokale delen har forskjellig struktur alt etter hvilket underliggende system en har med å gjøre. Vi kan (i følge [Elia93b]) argumentere for heterogene OIDer ut fra hensynet til skalering og ytelse. Hvis vi opererer med homogene OIDer (uavhengig av lokale systemer), vil objektadapter i hvert applikasjonsobjekt måtte ha ei avbildning fra den globale identitet til den lokale. En slik avbildningfunksjon kan for en del typer underliggende systemer utgjøre en signifikant overhead. I mange tilfeller kan avbildninger bli svært komplekse og dermed ressurskrevende. Dette er spesielt viktig hvis hver enkelt avbildning må lagres (en datastruktur). Lagermengde som da forbrukes, vil bli i størrelsesorden *O(n)* (der n er antall objekter). Dette er altså en grunn til at heterogene OIDer skalerer bedre enn homogene.

8.1.2 Operasjoner

Det er naturlig å avbilde en beskrivelse av ANSA-grensesnitt til eksterne abstrakte datatyper i FRIL. Der vil operasjonene i ANSA-grensesnittet svare til funksjoner i en tilsvarende ekstern ADT. Ved spesifikasjon av abstrakte datatyper kan FRIL funksjoner skrives på følgende form:

$$f : S_1 \times S_2 \times \ldots \times S_n \rightarrow T$$

Navn på funksjoner *(f)* vil bestemmes ut fra navn på operasjoner. FRIL funksjoner kan ha et vilkårlig antall argument-typer (kan oppfattes som kartesiske produkt) og en enkelt resultat-type. Der funksjonen inngår som en del av en ADT (i eksport skjema fra lokalt system), vil første argumentet alltid være et objekt av den typen operajonen inngår i beskrivelsen av (object of interest)[1]. Sjøl om ANSA-operasjoner kan ha flere, er ikke det noe stort problem, for argument- og resultatlister kan oppfattes som tupler av flere verdier. I slike tilfeller vil argument- og resultattypene i FRIL være tuppel-typer. Figuren viser et eksempel på avbildning av operasjonssignatur (her er *MyIF* navn på grensesnittet, som opptrer som type of interest i FRIL):

FIGUR 71. Eksempel på avbildning av operasjonssignatur

```
MyOp : OPERATION [arg: INTEGER] RETURNS [STRING, INTEGER];
       ⇓
MyOp : MyIF × Int →[String, Int]
```

8.1.3 Verdi-typer

I FRIL har vi et svært lite sett av typekonstruktører og innebygde verdi-typer. Det fører til at ikke alle typekonstruktører og innebygde typer i IDL har noe tilsvarende i FRIL. I FRIL har vi de atomiske typene, *Bool*, *Int*, *Real*, *Char*, *String* og *OID*. Vi har også konstruktører for tupler, mengder og lister (se for øvrig avsnitt 2.2.1).

1. Vi skal ikke undersøke nærmere hvordan man spesifiserer en korrekt ADT ut fra IDL skjema. Her antar vi at IDL-operasjoner kan avbildes direkte til tilsvarende funksjoner i FRIL med tilsvarende argument- og resultat-typer. Dette vil ofte være avledete funksjoner. Det vil si de inngår ikke som en del av den formelle ADT spesifikasjonen, men kan avledes fra funksjoner i denne.

Før vi går inn på verdiers representasjon i protokollen mellom evaluerings-tjeneste og applikasjonsobjekt, skal vi se nærmere på den språklige representasjon. Figuren nedenfor oppsummerer forholdet mellom IDL-typer og FRIL-typer (vi ser bort fra OID og grensesnittreferanse-typer her). Vi ser at oppramstyper og disjunkte unioner ikke umiddelbart har noe tilsvarende i FRIL og at det ikke finnes noe som tilsvarer mengder i IDL.

FIGUR 72. Avbildning mellom IDL og FRIL typer

Klasse	*IDL Type*	*FRIL type*	*Kommentar*
Atomiske typer	**INTEGER** **SHORT INTEGER** **LONG INTEGER** **CARDINAL** **SHORT CARDINAL** **LONG CARDINAL**	**Int**	CARDINAL er semantisk sett ulik INTEGER i den forstand at negative tall er utelukket.
	BOOLEAN	**Bool**	
	CHAR **OCTET**	**Char**	
	STRING	**String**	
	{a_1, .. a_n}	?	Heltall?, Algebraisk type?
Kartesiske produkt	**RECORD [a_1:T_1 .. a_n:T_n]**	**[T_1 .. T_n]** *(tupler)*	Navn på elementer går tapt i FRIL.
Mengder	?	**{*T*}** *(mengder)*	Sekvens?
Rekursive typer (lister)	**SEQUENCE OF *T***	***T*** *(lister)*	
Avbildninger	**ARRAY *n* OF *T***	***MAP(Int, T)*** *(avbildning) eller* ***T** (lister)*	
Disjunkte unioner	**CHOICE *S* OF {...}**	?	Algebraisk type? avbildning?

Atomiske verdier

IDLs *INTEGER*-type svarer til FRILs *Int* (heltall). I IDL kan man spesifisere tre slags representasjoner av heltall: *SHORT INTEGER, INTEGER* og *LONG INTEGER*. Semantisk sett er det forskjell mellom disse fordi de setter forskjellige grenser for hvor store tall de kan representere. LONG INTEGER kan representere de største tallene. Dermed kan man si at LONG INTEGER svarer til et større domene[2] av tall. For ordens skyld[3]:

dom(SHORT INTEGER) ⊆ *dom(INTEGER)* ⊆ *dom(LONG INTEGER)*

IDLs *CARDINAL*-type (positive heltall) svarer til et subdomene av *Int*. Eller man kan si:

dom(CARDINAL) ⊆ *dom(Int)*

Vi har ikke en egen *Cardinal*-type i FRIL. Da kan man avbilde fra *CARDINAL* til *int*, siden alle verdier av typen *CARDINAL* faller inn under domenet til *INTEGER*. Skal man avbilde fra *Int* til *CARDINAL*, forutsetter dette at det i FRIL-skjema er satt en begrens-

2. Med domenet til en type mener vi den mengde mulige verdier som tilfredsstiller typen.

3. Det er stort sett et implementasjonsspørsmål å velge et format som er i stand til å representere store nok tall. IDL tilbyr altså tre INTEGER typer som svarer til tre ulike representasjoner.

ing på hvilke verdier et bestemt argument av typen int kan ha for at avbildninga skal være korrekt. F.eks. en regel som sier at argument mindre enn null resulterer i feil.

På samme måte som med Int og *INTEGER* kan man si at FRILs *Real* og ANSAwares *REAL* svarer til hverandre. Man kan spesifisere to representasjonsformater av *Real*: *REAL* og *LONG REAL*. Det siste svarer til et større verdidomene enn det første.

Oppramstyper

I IDL kan programmerer definere nye typer ved å ramse opp en rekke navn. F.eks. slik:

```
WeekDay : TYPE = {mon, tue, wed, thu, fri, sat, sun};
```

Man ramser ganske enkelt opp alle elementer i verdi-domenet og det er dermed et avgrenset domene. Det er også ei ordning mellom disse elementene, noe som ikke defineres eksplisitt mellom de enkelte elementer, men som kan avledes fra rekkefølgen elementene står i type-deklarasjonen.

FRIL har ikke konstruktører for oppramstyper, men vi kan tenke oss at slike typer ble representert som heltall med et avgrenset verdiområde og at hver tillatte verdi ble bundet til et navn.

Tupler og lister

IDLs RECORD og FRILs tuppel-konstruktør svarer til samme konsept, nemlig kartesisk produkt. Records er en form for tupler hvor en navngir posisjonene i tuplene. FRILs tupler har i utgangspunktet ikke brukerdefinert navngiving av posisjoner

IDLs SEQUENCE betegner lister med et variabelt antall elementer. Dette svarer til listekonstruktør i FRIL.

Avbildninger

I enkelte programmeringsspråk. som f.eks. Pascal, kan array oppfattes som avbildninger (funksjoner) fra en hvilken som helst avgrensa diskret type (heltall eller oppramstyper), eller tupler av slike verdier, til en hvilken som helst type. I IDL er array et mer snevert begrep. Man avbilder kun fra positive heltall som angir posisjon i array.

Arrays kan altså i FRIL oppfattes som funksjoner som avbilder fra verdier i et domene til verdier i et annet. En array type i IDL som dette:

```
MyArr : TYPE = ARRAY n OF T
```

Vil kunne avbildes til ei avbildning[4] i FRIL. Det må også spesifiseres en begrensing på hvilke verdier argumentet til funksjonen kan ha. Det vil si hvis argument ikke ligger mellom 0 og n vil det føre til feil.

```
MyArr ::= MAP(Int, T)
```

IDL arrays kan også representeres som lister med fast størrelse.

Disjunkte unioner

Dette er verdi-domener hvor verdier er valgt fra en av flere typer. Vi snakker altså om verdi-domener som er union av flere verdi-domener. For hver verdi av et disjunkt-

4. FRIL har ikke en egen innebygget avbildnings type. En kan eventuelt vurdere utvidelser av språk eller definere avbildingstyper ved hjelp av andre typer (f.eks. lister og tupler).

union-domene er det nødvendig å ha assosiert et navn eller en "tag" for at uttrykk som bruker verdiene skal vite hvilke typer de er av. Formelt kan en definerer disjunkt union mellom to typer S og T slik:

$$S + T = \{ \text{navn}_1 x \mid (x \in S) \} \cup \{ \text{navn}_2 y \mid (y \in T) \}$$

IDLs CHOICE type er en form for disjunkte unioner. De navn (eller "tag") som indikerer hvilken type verdien er av, er av en opprams-type som definerers for seg sjøl. Figuren nedenfor viser et eksempel på en union av opprams-typen *Reason* og *STRING*.

FIGUR 73. Eksempel på enumeration og choice

```
Op_Status : TYPE = {Success, Failure};
Reason : TYPE = {E_range, E_server, E_unknown};
Op_Result : TYPE = CHOICE Op_Status OF {
    Success => STRING;
    Failure => Reason;
};
```

FRIL har ikke konstruktører for noe tilsvarende. En mulig løsning kan være er å utvide definisjonen av hva som er verdier i FRIL til å kunne uttrykke disjunkte unioner. Funksjonelle språk som ML og Miranda kan uttrykke tilsvarende med sitt konstruktør-begrep. I Miranda vil eksemplet over kunne uttrykkes slik:

FIGUR 74. Miranda-konstruktør typer

```
reason ::= E_range | E_server | E_unknown
op_Result ::= Success(string) | Failure(reason)
```

Her er opprams-typer og disjunkte unioner samla i et enkelt konsept, det vil si konstruktør-funksjoner som returnerer en verdi av den angitte typen. Ei rekke med konstruktører uten argumenter fungerer som oppramstyper, mens konstruktører med argumenter svarer til disjunkte unioner.

8. 2. Protokoll for aksess til eksterne objekter

Vi skal her skissere en protokoll/grensesnitt mellom eval-tjeneste og applikasjonsobjekt. Dette tilsvarer det vi i kapittel 7 kalte en kanonisk protokoll. Protokollen kan spesifiserers som et IDL grensesnitt slik:

FIGUR 75. Grensesnitt

```
ObjectAdapter : INTERFACE =
BEGIN
    LocalOID : TYPE = SEQUENCE OF OCTET;    -- Lokal objekt-identifikator
    ARG      : TYPE = SEQUENCE OF OCTET;    -- Argument/resultat liste
    FID      : TYPE = STRING;               -- <typenavn>.<operasjonsnavn>

    Status : TYPE =  {
        invocationOK,       -- Ok!
        oidError,           -- Ukjent OID-format eller ukjent OID
        typeError,          -- Objekt er av feil type
        argumentError,      -- Argumentliste ikke som forventet
        invocationError     -- Annen feil ved anrop
    };

    Bind   : OPERATION [x: STRING]                          RETURNS [LocalOID];
    Unbind : OPERATION [oid: LocalOID]                      RETURNS [];
    Invoke : OPERATION [oid: LocalOID; fid: FID; arg: ARG] RETURNS [Status; ARG]

    Session_Start : ANNOUNCEMENT OPERATION [] RETURNS [];
    Session_End   : ANNOUNCEMENT OPERATION [] RETURNS [];
END.
```

8.2.1 OID format

I design av et rammeverk for objektadaptere (kapittel 5) er alle objekter representert som proxy-objekter, og vi har i den forstand en form for homogen objekt-identifikasjon. Mer generelt er det ønskelig at vi åpner for heterogen objekt-identifikasjon og at dette gjenspeiles i et grensesnitt mellom evalueringstjeneste og applikasjonsobjekter. Vi kan både tenke oss framtidig støtte for heterogene OID i rammeverket, samt at systemet kan tenkes å ha objektadaptere som ikke er bygd på akkurat dette rammeverket.

I dette grensesnittet skal vi derfor ikke forutsette noe som helst om hva en objekt-identifikator egentlig erDette er transparent for evalueringstjenesten og protokollen. En objekt-identifikator oppfattes her ikke som noe annet en en streng med oktetter (med variabel lengde). Det er opp til det lokale systemet eller objektadapter å tolke denne strengen.

8.2.2 Format for argument og resultat

Hvis vi ser på hvordan verdi-typer skal representeres i protokollen (jfr. typen '*ARG*' i grensesnittet over), bør vi først avklare om vi skal basere oss på IDL-typene eller de tilsvarende FRIL-typer. Generelt er det ønskelig at grensesnittet er mest mulig språkuavhengig, det vil si det burde kunne beskrive alle tenkelige typer. Her skal vi først og fremst basere oss på det enkle sett av typer og type-konstruktører som tilbys av FRIL.

Det er irrelevant for protokollen her hvilke typer argument og resultat egentlig består av. Den kan her forstås som en streng med oktetter. Det er opp til stub'er og eval-tjeneste å kode og dekode disse. Når det gjelder spørsmålet om hvordan disse konkret skal kodes, har vi i hovedsak to alternativer[5]:

- Løsninger der mottaker kan bestemme hvilke typer de innkommende data er av og allokere de riktige datastrukturer, kun ut fra informasjon som er inneholdt i disse data. Det vil si at typeinformasjon er kodet inn i data-elementer.
- Løsninger der mottaker kan avgjøre typer ut fra andre kriterier. Det betyr i vårt tilfelle at eksportskjema vil måtte inneholde informasjon om typer til argumenter og resultater.

5. Se f.eks. [Rose91] for en diskusjon av alternativer for overføringssyntaks.

FIGUR 76. Forslag til (eksperimentelt) format for overføring av verdier

FRIL type	*Representasjon*
Int	32 bit enhet (*long int*)
Bool	8 bit enhet (*char*)
Real	64 bit enhet (*double*)
Char	En oktett (*char*)
String	Et 16 bits heltall (*short int*) som angir antall tegn + tegnene
Kartesiske produkt (tupler)	Hvert element kommer i den rekkefølge det står spesifisert i skjema
Lister	Et 16 bits heltall (*short int*) som angir antall elementer + elementene.
Avbildninger (array)	Hvert element kommer fortløpende. Først det som svarer til posisjon 1, så posisjon 2 osv..

Så lenge vi har tilgang til informasjon om argument og resultattyper i skjema, er det ikke nødvendig å kode typeinformasjon inn i datastrømmen mellom komponentene. Type-informasjon trenger vi bare hvis vi ikke på forhånd kan vite hva som forventes. Lengde-informasjon trenger vi bare når vi ikke på forhånd vet hvor mange dataelementer som forventes. Her vil det gjelde strenger, lister (og mengder). Figuren over viser et forslag til (eksperimentell) overføringssyntaks for verdi-typer. For tall velges en representasjon med tilstrekkelig størrelse til å romme de tall en kan forvente (C-typer som tilsvarer repr. er angitt i parantes). Det er også mulig å ha alternative representasjoner av tall (variabel størrelse), sjøl om dette er transparent i FRIL-omgivelsen.

8.2.3 Funksjons-identifikasjon

Operasjoner vil i protokollen være representert med funksjons-identifikatorer. Når informasjon om grensesnittene til applikasjonen eksporteres til de føderative omgivelsene, vil dette bety eval-forvalter får informasjon om typer, operasjonsnavn og deres respektive identifikatorer, slik at når klient refererer til en funksjon med et navn, vil evalueringstjenestens omgivelser avbilde til riktig identifikator når tilhørende ekstern funksjon skal kalles. Dette kan i praksis være en tekstlig representasjon av operasjonsnavn.

8.2.4 Operasjonene i grensesnittet

Operasjonene i grensesnittet svarer i stor grad til de konseptuelle operasjonene som ble beskrevet i avsnitt 4.4.2). Jeg skal her kort forklare virkemåten til hver av dem.:

- Bind. Returnerer objekt-identifikator til et persistensrot-objekt identifisert med et tekstlig navn. Det forutsettes at evalueringstjenesten på forhånd har fått informasjon om type og navn på mulige persistensrot-objekter.
- Unbind. Eksplisitt unbind for et angitt objekt.
- Invoke. Anroper en ekstern funksjon identifisert med en funksjons-identifikator i et objekt identifisert med OID.
- Session_Start og Session_End kan brukes av evalueringstjensten for å markere starten og slutten på en sesjon. Når en sesjon avsluttes (*Session_End*) vil objektadapter implisitt oppfatte dette som at det blir gjort Unbind på alle objekter som har blitt bundet innenfor sesjonen. Dette gjelder spesielt objekter med temporær identitet som har blitt bundet til som resultat av beregninger (se for øvrig avsnitt 5.2.4).

8. 3. Litt om arkitektur

I følge avsnitt 2. 6 har FRIL-arkitekturen en typeforvalter og en objektforvalter. Når en FRIL-tjener eksporterer et skjema vil det si at skjema (typedefinisjoner) registreres i typeforvalter og referanser til persistens-rot-objekter registreres i objektforvalter. FRIL-klienter vil interagere med disse forvaltere for å få skjema informasjon og binde til rot-objekter. Dette kan tenkes å ha form som en trading tjeneste (jfr. avsnitt 5.2.2).

Her er modellen forenklet. Vi trenger kun forholde oss til en evalueringstjeneste som representerer føderasjonen (se figur 70). Vi kan oppfatte den som en sentralisert interpreter for FRIL programmer[6]. Vi er interessert i å eksportere skjema-informasjon til denne. Det kan gjøres ved at det genereres filer med tekstlige representasjoner av skjema-informasjon. FRIL interpreter kan lese inn disse.

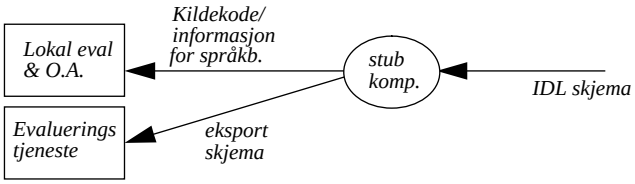
8. 4. Eksport skjema

Vi skal her beskrive et format for eksport-skjema. Et eksport-skjema brukes for å utveksle skjema-informasjon fra applikasjons-objektet til evaluerings-tjenesten, slik at denne får kjennskap til typer, operasjonssignaturer og persistensrot-objekter som applikasjonsobjektet tilbyr (jfr. avsnitt 7.3.4).

8.4.1 Et eksperimentelt format for eksport

Vi tenker oss at skjema informasjon eksporteres på en kompakt form. Denne vil kunne anvendes nesten direkte og inneholder kun tekst som er nødvendig. Det er meninga at et slikt skjema genereres av en kompilator, enten fra et mer lesbart format eller fra det skjemaspesifikasjons-språk som brukes i det aktuelle lokale systemet. For ANSAware er det aktuelt å generere eksport skjema fra IDL-skjema ved hjelp av en stub-kompilator. Figuren nedenfor illustrerer hvordan en stub-kompilator genererer både eksport-skjema og kode (eller tilsvarende informasjon) som inngår i objekt-adapterens språk-binding. I forbindelse med realisering av en eksperimentell prototype, er en slik kompakt form det enkleste, da den er enkel å implementere, spesielt med hensyn import i evalueringstjenesten.

FIGUR 77. Stub-kompilator og eksport skjema



Skjema struktur

Når vi skal bestemme oss for et format, legger vi vekt på at tolking av denne skal kunne implementeres enkelt og oversiktlig. Formatet for skjema vil være nært det interne formatet for denne informasjonen i implementasjon av evalueringstjenesten, men det vil være slik at det også ved hjelp av en tekst editor er mulig å lese eller redigere eksport-skjema. Det som må definieres av et slikt skjema er følgende:

- Et navn på applikasjons-objektet, slik at objekt-adapter kan identifiseres.
- Et navn på hver ADT i skjema.
- Et navn på hver operasjon i skjema.

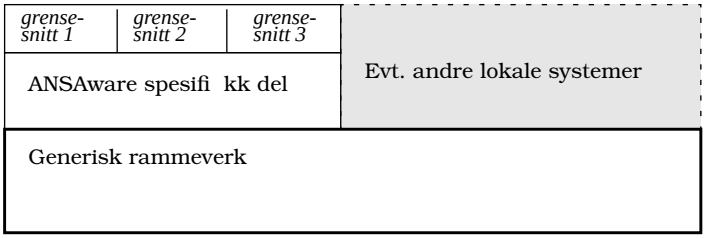
6. En tidlig prototype av eval-tjenesten er også implementert slik (jfr. [Sundsfjord94]).

- For hver operasjon, en angivelse av forventet argument- og resultattype.
- For hver type, en mengde navn på persistens-rot objekter. Denne mengden kan være tom.

8. 5. Et eksperimentelt rammeverk for binding til operasjonelle grensesnitt

Her skal vi beskrive et enkelt objekt-orientert rammeverk for konstruksjon av FRIL språkbindinger. På samme måte som i avsnitt 5. 1, kan vi designe og implementere en del av softwaren, uavhengig av ANSAware eller andre lokale systemer. En annen del av softwaren vil være spesifikk for lokale systemer, og en mulig tredje del av softwaren vil være spesifikk for applikasjonen igjen. Vi har tidligere antydet at det for hvert grensesnitt vil måtte genereres en del software. Figuren nedenfor illustrerer dette. Her er det tegnet inn tre applikasjonsavhengige moduler, generert ut fra grensesnittspesifikasjon:

FIGUR 78. Byggeklosser for språkbinding til FRIL



Vi bør strebe etter et rammeverk, hvor skjema-spesifikk atferd i størst mulig grad er en dynamisk egenskap. Dette har den fordel at lokal eval./objekt-adapter ikke trenger å rekompileres ved endring av skjema.

Det er mulig å realisere språkbindinger slik for ANSAware. En kan for det første tenke seg en generisk stub som er implementert uavhengig av de spesielle operasjonssignaturene. Denne vil ta operasjons-navn (identifikator), argument-type og resultat-type som parametre under kjøretid. En kan for det andre tenke seg at stub'en anvender generiske marshalling-rutiner som tar type-definisjoner som parameter (argument- og resultattyper).

I den prototype-implementasjonen som vi skal skissere her, har vi en generisk stub, men vi vil i første omgang ha separate marshalling-prosedyrer for de ulike argument- og resultat-typene. Skjema-spesifikk kode genereres av stub-kompilator. Dette gjøres av pragmatiske grunner: Det vil si vi kan raskere realisere en implementasjon ved at vi blant annet anvender kode-genererings funksjoner som allerede eksisterer i ANSAware plattformen.

8.5.1 Generisk rammeverk

Det generiske rammeverket består av en implementasjon av grensesnittet til eval-forvalter, en abstrakt klasse for operasjoner og en avbildning fra operasjons-identifikator til den riktige operasjon.

Abstrakt operasjons klasse

Vi representerer operasjonene i grensesnittene som objekter. For disse definerer vi en abstrakt klasse som definerer atferd felles for alle mulige operasjoner. For ANSAware og andre lokale systemer vil vi definere subklasser som definerer atferd som er spesifikk for det lokale systemet.

For hver operasjonssignatur i skjema vil vi opprette en instans av operasjons-klassen. Egenskaper ved utførelsen av operasjoner, som er felles for alle operasjoner innenfor et lokalt system, er definert i denne klassen. Signatur-spesifikke egenskaper er definert som attributter i hver enkelt instans. Dette er i så måte dynamisk informasjon. En abstrakt operasjonsklasse definerer i hovedsak følgende:

- Navn på grensesnitt-type og operasjon.
- En generisk metode for å gjøre anrop. Den er avhengig av en abstrakt metode som er definert i konkrete subklasser.

Operasjons-tabell

Ved anrop på operasjoner vil funksjons-identifikator (her vil det si tekstlig navn på type og på operasjon) måtte avbildes til et operasjonsobjekt. For dette formålet har vi en operasjonstabell. Hvert operasjons-objekt settes inn i denne tabellen. Når en operasjon anropes, vil vi slå opp i tabellen, finne operasjonsobjektet og kalle metoden for anrop.

Anrop

Operasjonsklassen har en metode *Invoke* som sørger for å få utført anropet. Operasjonen fungerer i essens slik:

Først konverteres objekt-identifikator til et proxy-objekt ved oppslag i objekt-adapterens forvalter-objekt (*MANAGER*). Deretter skaffes en aktivisering av objektet ved hjelp av proxy-objektets metode *get_Activation*. Denne returnerer lokalenhet hvis proxy-objektet representerer ei gyldig aktivisering og vil forsøke å aktivisere objektet hvis det er permanent og ikke allerede aktivt. Hvis typen[7] til denne lokalenheten stemmer overens med den typen som det konkrete operasjonsobjektet forventer (sjekkes ved hjelp av den abstrakte metoden *typeOf*), kalles den abstrakte metoden *Invoke*, med lokalenheten som argument. Den som implementerer den abstrakte *Invoke* metoden kan ta for gitt at lokalenheten som blir brukt som argument er gyldig og av riktig type. Figuren nedenfor illustrerer dette (*OA_oid* er objekt-adapters egen representasjon av objekt-identifikator).

FIGUR 79. Algoritme for anrop (C++)

```
...

if ((p = MANAGER->lookup(OA_oid))== NULL)     // Slå opp proxy i forvalter
   return oidError;                           // --> Ukjent objekt
else
{
   LocalUnit& lu = p->get_Activation();       // Aktivisering
   if (typeOf() != lu.typeOf())               // Sjekk lokalenhet-typen
      return typeError;                       // --> Typefeil
   else
      return Invoke(lu, argument, result);    // Abstrakt Invoke
}
```

7. Vær obs på at type her som regel angir hvilken konkret type lokalenhet man har med å gjøre. Alle objekter for ANSAware applikasjoner vil f.eks. ha samme *typeOf* verdi her. . .

Grensesnitt til eval-forvalter

Har vi en abstrakt klasse for operasjoner, kan software for interaksjon med eval-forvalter realiseres som en del av det generiske rammeverket. Denne delen vil altså kunne realiseres uavhengig av hvilke lokale system som anvendes. Denne kan realiseres ved hjelp av ANSAware[8]. Rammeverket implementerer i så måte hver av operasjonene i grensesnittet. De aktuelle operasjons-implementasjonen gjør i essens følgende:

- Bind: Avbilder fra navn til objekt-identifikator ved hjelp av en egen tabell som er opprettet for formålet. Hvert objekt som er ment å være persistensrot, vil bli satt inn i denne tabellen (se for øvrig avsnitt 6.2.7 hvordan man kan spesifisere rot-objekter).
- Unbind: Referanse teller vil bli telt ned og objekter som ikke er referert til mer, vil bli søppelsamlet (dette har bare mening for temporære objekter).
- Invoke: Avbilder fra operasjonsidentifikator til operasjonsobjekt og kaller operasjonen *Invoke* hos dette.
- Session_Start: Markere start på sesjon. Fra denne operasjonen blir anrop og inntil *session_End* blir anropt, vil alle objekter som returneres fra kall på *Invoke*, måtte settes inn i en egen mengde (vi kan kalle den sesjons-mengde).
- Session_End: Det blir gjort *unbind* på alle objekter som befinner seg i sesjonsmengden.

8.5.2 Realisering av ANSAware anrop

For realisering av ANSAware anrop, lager vi en konkret operasjonsklasse for ANSAware grensesnitt. Vi kaller den *AnsaOperation*. I denne klassen implementeres de abstrakte metodene *Invoke* og *typeOf*. I tillegg til de attributter som er definert i den abstrakte operasjonsklassen, har *AnsaOperation* følgende[9]:

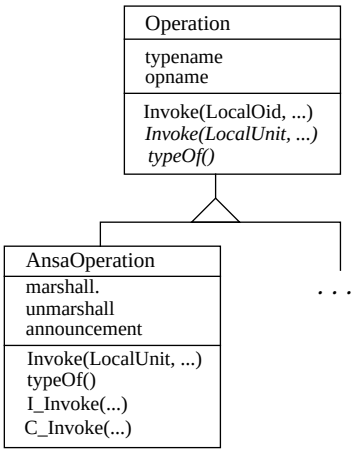
- Peker til funksjon som avbilder argumenter til ANSAwares buffer-format (*marshall*).
- Peker til funksjon som avbilder resultater fra ANSAwares buffer-format (*unmarshall*).
- Et attributt som indikerer om kallsemantikk er annonserende (*ANNOUNCEMENT)* (se avsnitt 2.7.4).

Figuren nedenfor oppsummerer den abstrakte og den konkrete operasjonsklassen og hvilke attributter og metoder de har (jfr. notasjon fra avsnitt 5. 1).

8. Dette har for øvrig ingen sammenheng med at binding til ANSAware applikasjoner studeres spesielt i denne avhandlinga.

9. I en mer ideell løsning tenker vi oss at de to første attributtene inneholder abstrakt syntaks beskrivelser av argumenter og resultater som kan tolkes av generiske marshalling funksjoner.

FIGUR 80. Operasjons klassene



Stub-metode

Den abstrakte *Invoke*-metoden fungerer som en stub. En instans av *AnsaOperation* klassen har tilstrekkelig informasjon i seg til at anrop kan utføres, gitt at lokalenhet med grensesnittreferanse og argument-liste oppgis som argumenter. Framgangsmåten for anrop kan skisseres som følger:

- Konvertere lokal-enhet til grensesnittreferanse.
- Initiere anrop: Dette betyr at argumenter avbildes ved hjelp av *marshall* funksjon.
- Hvis kallsemantikk ikke skal være annonserende, hentes resultat fra anrop og *unmarshall* funksjonen kalles for avbildning av resultater.

8.5.3 Instansiering av objekt-adapter

Når en for en gitt ANSAware applikasjon skal realisere en objektadapter, må følgende gjøres.

- Definere permanente objekter i henhold til framgangsmåten som ble skissert i avsnitt 6. 2. Ved oppstart av objekt-adapter, aktiviseres det en proxy for hver av disse, og for hvert av de permanente objekter som skal eksporteres, etableres det ei avbildning mellom et tekstlig navn og objekt-identifikator
- For hver type som blir definert i grensesnitt-spesifikasjonene generères (av stub-kompilator) avbildnings-funksjonene *MAP_T* og *UNMAP_T* (der *T* er navn på den aktuelle typen). Disse avbilder henholdsvis argumenter og resultater. For operasjonssignaturer med flere argumenter eller resultater genereres det egne funksjoner som om operasjonen bare hadde en enkelt argument- eller resultat-type (det vil si en tuppel type). Disse kan f.eks. kalles henholdsvis *MAP_f_ARG* og *UNMAP_f_RES* (der *f* er operasjons-navn)[10].
- Ved oppstart av objekt-adapter instansieres det et *AnsaOperation* objekt for hver av operasjonene i grensesnittene. De instansieres med de attributter de skal ha, nemlig navn, pekere til avbildnings-funksjoner og indikasjon om kallsemantikk skal være annonserende. Det etableres ei avbildning fra operasjonsidentifikator til hvert av disse objektene.

  Stub-kompilator vil måtte generere tabeller eller fil med den informasjon som brukes til instansiering av operasjonsobjekter. Disse leses ved oppstart av adapter-program og objekter instansieres.

10. Ved bruk av generiske marshalling funksjoner genereres ikke funksjoner her.

8. 6. Oppsummering

Vi har diskutert representasjon av henholdsvis objektreferanser, operasjoner og verdier i bindinga til FRIL. Objektreferanser vil være representert ved objekt-identifikatorer som kan være permanente og hvor semantikk er transparent for FRIL-klienter. Fra en OID må vi kunne avlede både applikasjonsobjekters lokasjon og de enkelte lokale data-elementer. Vi har ut fra dette en todelt forståelse av globale objektidentifikatorer. I grensesnittet mellom evalueringstjeneste og applikasjonsobjektet er vi bare interessert i den delen av OID som identifiserer lokale data. Denne kan være homogene (samme format for alle underliggende systemer) eller heterogene (ulike format). Heterogene lokale OIDer gir mulighet for å bruke den lokale form for identifikasjon direkte og vil kunne gi bedre effektivitet mht. skalering.

FRIL har et lite sett av verdi-typer i forhold til IDL. Vi har f.eks. sett at IDL har mange varianter av heltall som alle vil måtte avbildes til FRILs *Int* type. Ulike måter å representere tall på er transparent i FRIL. Begrensinger på hvor store tall som kan representeres kan gjenspeiles i FRIL ved hjelp av integritetsregler. Vi har også sett at oppramstyper og union-typer ikke har tilsvarende i FRIL og vi må enten utvide FRIL med flere typekonstruktører eller lage ad. hoc. løsninger (f.eks. at oppramstyper representeres som heltall).

Vi har skissert en protokoll mellom eval-tjeneste og applikasjonsobjekt. Denne er spesifisert som et IDL grensesnitt. Her er OID en variabel sekvens av oktetter, for å tillate heterogene OIDer. Argumenter og resultater er også bare en sekvens med oktetter her siden vi ønsker at grensenittet skal kunne realiseres uavhengig av spesifikke applikasjoner. Vi har skissert et forslag til overføringssyntaks for verdier (argumenter/resultater). Funksjoner identifiseres med tekstlige representasjoner av navn. Grensesnittet har operasjonene *bind*, *unbind*, *invoke*, *session_start* og *session_end* som vi har presentert i tidligere kapittel.

I en tidlig prototype fase kan vi ha en forenklet tilnærming til arkitekturen og til eksport skjema. Et eksport skjema er ganske enkelt ei fil med en (kompakt) tekstlig representasjon av den nødvendige informasjon. Et slikt skjema inneholder navn på applikasjonsobjekt, navn på ADTer, navn på hver enkelt operasjon, spesifikasjon av argument og resultattyper og referanse (navn) til persistensrot objekter.

Til slutt har vi skissert et rammeverk for realisering (implementasjon) av FRIL språkbindinger. Dette består av et generisk rammeverk (implementasjon av de deler som er uavhengig av lokale systemer), påbygninger for ANSAware og eventuelle tillegg for applikasjoner. Sentralt her er en generisk stub som kan parametriseres med informasjon om de spesifikke operasjonssignaturer.

Kapittel 9
Implementasjon og evaluering

I dette avsnittet skal vi vise hvordan rammeverket for objekt-adapter og språkbindinger lar seg implementere. Prototype-implementasjoner har blitt realisert for deler av dette rammeverket og vi skal presentere disse og se hvordan de kan anvendes på eksisterende ANSAware applikasjoner. Vi vil videre diskutere dette med hensyn til kvalitetsparametre som ytelse, skalering, anvendelighet og generalitet. Vi vil se på enkelte sterke og svake sider ved vårt rammeverk og undersøke muligheter for forbedringer der dette synes hensiktsmessig. Et spørsmål er i hvilken grad objekt-adapter rammeverket kan bidra til "verdi-økning" i forhold til ANSAware omgivelsene og i hvilken grad dette gjøres til en akseptabel kostnad. Skillet mellom definisjon av forvaltningsoppgaver fra programmering av klient, samt notasjon/verktøy for beskrivelse og mekanismer for transparens mht. persistens, er for eksempel et viktig bidrag til "verdi-økning". ANSAware ikke tilbyr dette i utgangspunktet.

Rammeverket undersøkes stort sett ved kvalitativ vurdering av ulike sider og anvendelsen av dette på konkrete applikasjoner. Analysen er langt fra fullstendig, men belyser en del aspekter vi har sett på som viktig. Andre viktige aspekter som f.eks. feiltoleranse er utelatt her, men fortjener oppmerksomhet i eventuelle videre arbeid med rammeverket. Før vi går nærmere inn på detaljene, skal vi utdype hva vi ønsker å legge i kvalitets begrepet.

9. 1. Aspekter ved kvalitet

Vi har valgt å se på følgende aspekter ved kvalitet:

- Ytelseskostnad ved integrasjon. Hvordan oppnå best mulig ytelse?
- Tap eller vinning med hensyn til skalerbarhet.
- Tap eller vinning med hensyn til anvendelighet. Kan vi si bruk av applikasjoner kompliseres eller avgrenses, eller kan vi si at det motsatte skjer?

I tillegg er vi interessert i hvilken grad objekt-adapter-rammeverket lar seg anvende på andre underliggende informasjonssystemer enn ANSAware, altså vi ser litt på generalitet.

9. 2. Beskrivelse av implementasjoner

Vi skal først vise hvordan rammeverket lar seg implementere. Sentrale deler av dette har blitt implementert og testet. Dette gjelder spesielt objekt-adapter-rammeverket og språkbinding for C++. Vi presenterer her en første prototype versjon. Det er rom for forbedring og videreutvikling av denne i eventuelt videre arbeid med rammeverket.

9.2.1 Prototype implementasjon av objekt-adapter-rammeverket

En prototype-implementasjon av objekt-adapter rammeverket (beskrevet i avsnitt 5) er realisert i C++ og PREPC. Grensesnittene til brukere av rammeverket er C++ klasser som tilsvarer klassene beskrevet i avsnitt 5.

Rammeverket er realisert som et bibliotek. Programmer som realiserer konkrete objekt-adaptere anvender dette og adderer følgende software moduler i den grad det er nødvendig:

- Applikasjons-spesifikke klasser (spesialiseringer av klasser i rammeverket) der det er nødvendig. Spesielt gjelder dette spesialiseringer av klassen *Perm*.
- Kode for initialisering av objekt-adapter instans (jfr. avsnitt 5.2.1). Dette inkluderer også kode for instansiering av permanente objekter (jfr. kapittel 6).
- Kode for språkbinding for det aktuelle skjema (jfr. kapittel 7).
- Kode for klientprogram hvis det skal være i samme prosess.

Kildekoden til biblioteket (*libOA*) er i sin helhet gjengitt i vedlegg 1. Her skal vi kort trekke fram det mest essensielle ved hver modul:

Generisk rammeverk

Proxy.h og Proxy.C inneholder implementasjon av de generiske klassene (uavhengig av ANSAware) som er beskrevet i avsnitt 5. 1. Følgende er definert her:

- En objekt-identifikator type *OID* (Dette er i første omgang et heltall).
- Abstrakt klasse *LocalUnit* (lokal representasjon av aktivt objekt).
- Abstrakt klasse *Proxy*
- Abstrakt klasse *Meta_Proxy*.
- Klasse *OIDManager*. Denne inneholder en katalog over aktive proxy-objekter. For å realisere katalogen effektivt, brukes typen *dictionary* fra klassebiblioteket LEDA [LEDA.3.0] som der er implementert som et randomisert søketre. Tidskompleksitet for innsetting, oppslag og sletting her er *O(log n)*, og det forutsettes at det er definert ei ordning på objekt-identifikatorer. Hvis man søker med hensyn på *LocalUnit* har dette (i denne implementasjonen) tidskompleksitet *O(n)* siden dette gjøres ved å gå gjennom alle elementer i *OIDManager*.
- Abstrakt klasse *Perm.*

ANSAware spesifikt rammeverk

AnsaProxy.h og *AnsaProxy.C* inneholder implementasjon av de ANSAware spesifikke klassene som er beskrevet i avsnitt 5. 1. Følgende er definert her:

- *ANSA_LU* - Lokalenhet spesialisert for ANSAware objekter. Disse består ganske enkelt av en grensesnittreferanse.
- *ANSA_Proxy* - Proxy-klasse spesialisert for ANSAware objekter.
- *Meta_Ansa* - Meta-klasse spesialisert for ANSAware objekter.
- *ansa_Server_Perm* - *Perm*-klassen spesialsert for permanente objekter som avhenger av et tjener objekt. Det er fortsatt en abstrakt klasse som må spesialiseres videre.

Objekt forvaltnings funksjoner

GA_mgmt.h og *GA_mgmt.dpl* inneholder generelle funksjoner for objekt-forvaltning, det vil si aktivisering og passivisering av ANSAware objekter. Funksjonene brukes av konkrete *Perm*-objekter og kan kalles fra C++. Funksjonene som tilbys er for følgende:

- Trading, det vil si å finne eksisterende aktiviseringer.
- Aktivisering av kapsel ved hjelp av *Factory*
- Passivisering av kapsel ved hjelp av kapselens eget grensesnitt.

- Aktivisering av objekt ved hjelp av kapsel-grensesnitt.
- Aktivisering av objekt ved hjelp av objektets eget grensesnitt.

Kapsel forvaltning

CapsulePerm.h og *CapsulePerm.C* inneholder implementasjon av klassen *ansa_FactCapsule_Perm* som er en konkret spesialisering av *Perm*-klassen. Den skal representere kapsel-objekter. Vi trenger å kunne forvalte kapsler, sjøl om disse ikke vil være synlig for klienter. Kapsler aktiviseres ved hjelp av *Factory* og passiviseres ved hjelp av sitt eget grensesnitt. Implementasjonen realiserer foreløpig ikke notifikasjon. Kapsel klassen har følgende attributter som brukes ved aktivisering.

- Node navn
- Sti i filsystem hvor template for kapsel befinner seg.
- Filnavn som representerer template.
- Argument-streng som oppgis til kapsel ved oppstart.
- Omgivelses streng.

Likhet mellom passive kapsler definerers (som en ad. hoc. løsning) slik: Hvis både node-navn, sti, filnavn og argumentstreng er lik, kan vi slutte at det dreier seg om samme (passive) kapsel.

Inklusjon av ANSAware omgivelser i C++ programmer

ansa++.h er ei headerfil for å inkludere nødvendige definisjoner fra ANSAware i C++ programmer. Dette er ikke problemfritt, siden ANSAware i utgangspunktet er implementert i C og beregnet på språkbinding til C. Vi viser til vedlegg 2 for kildekode og en beskrivelse av hvordan ANSAware applikasjoner kan implementeres i C++.

9.2.2 Prototype implementasjon av språkbinding

Ei språkbinding for C++ klienter er realisert. Dette er i essens en stub-kompilator som genererer definisjoner og kode fra grensesnitt-skjema. Dermed er det mulig å integrere vilkårlige ANSAware applikasjoner i C++ programmer. ANSAware grensesnitt opptrer her som C++ klasser (jfr. eksemplet i avsnitt 7.3.3). Se vedlegg 3for kildekode. Stub-kompilatoren *stubcpp* er en modifisert utgave av ANSAwares *stubc*. Endringene som er gjort er i essens slik:

- Større omskriving av kompilatorens back-end som skriver ut den genererte koden (outsubs.c)
- En liten endring i front-end for å få med tilleggsinformasjon om subtyperelasjoner. Se vedlegg 3 for en kort oppsummering av endringer som er gjort i tillegg til back-end modulet.

Hva som genereres av *stubcpp*

Vi skal kort se på hvilke elementer av kode som genereres av *stubcpp*. Det er som følger (gitt at vi har et grensesnitt med navn "*Example*"):

- *cExample.C* - stub-funksjoner for innledning av anrop og henting av resultat. De er stort sett lik de stub-funksjonene som genereres av den opprinnelige *stubc*.
- *mExample.h mExample.C* - marshalling-funksjoner. Disse er stort sett lik de marshalling funksjonene som genereres av den opprinnelige *stubc*, men det er gjort endringer for noen typer. Objekt-referanser/grensesnitt-referanser krever helt spesiell behandling.

- *IFC_Example.h* - Deklarasjoner av objekt-referanse klasse og verdi-typer. Denne inkluderes i klient-programmer.
- *IFC_Example.C* - Nødvendig implementasjon av metoder til typene som er definert i *IFC_Example.h*. Det gjelder spesielt metoder som tilsvarer operasjoner. Disse er modellert etter kode-sekvenser som genereres av *PREPC* for anrop. Det som gjøres er i essens å kalle stub-funksjoner som er definert i *Example.C*. *IFC_Example.C* vil i tillegg inneholde implementasjon av metoder for eventuelle verdi-typer som representeres som klasser.

Andre komponenter i C++ språkbinding

Språkbindinga til C++ som er beskrevet her forutsetter følgende i tillegg til det som blir generert per gresnesnitt.

- En parametrisert type *Seq_List<T>* for å representere SEQUENCE verdier. Denne skal ha metodene *get, put, rewrite* og *reset*. I vedlegg 4 presenteres et eksempel på implementasjon av *Seq_List* ved hjelp av lenkede lister (*seqlist.h*).
- En superklasse for alle objekt-referanse klasser, der en definerer det som er felles. Se vedlegg 3 (*IFC_ansa.h*).

Mot en prototype-implementasjon av FRIL binding

Binding til FRIL-evalueringstjeneste som skissert i kapittel 8 er ikke fullt ut realisert innenfor rammen av denne avhandlinga. For å illustrere ideene i rammeverket som ble beskrevet i kapittel 8. 5, viser vi til kildekode i vedlegg 4. Der finner vi følgende:

- En IDL-spesifikasjon av grensesnittet til objekt-adapter og FRIL-binding (*Object-Adapter.idl*)
- Definisjon og implementasjon av den abstrakte klassen *Operation* (jfr. avsnitt 8.5.1) som representerer generiske operasjoner (*Operation.h* og *Operation.C*).
- Definisjon og implementasjon av klassen *AnsaOperation* (jfr. avsnitt 8.5.2) som er en spesialisering av *Operation* for ANSAware. Denne tilbyr en generisk stub (*AnsaOperation.h* og *AnsaOperation.C*).
- Klasser som kan brukes som grensesnitt til sekvensen med oktetter som representerer argumenter og resultater til operasjoner. Her tilbys lesing og skriving av de vanligste verdi-typer i C/C++ (*ArgList.h* og *Arglist.C*).
- Testprogram som demonstrerer bruk av disse klassene sammen med objekt-adapter. Programmet bruker et ANSAware grensesnitt *Ident* som tilbyr en enkelt operasjon *Echo* som ganske enkelt returnerer den tekststreng som blir gitt som argument (sammen med navnet på det spesifikke objektet) *Ident*-objekter forvaltes av kapsel. Testprogrammet anvender funksjoner for å konvertere OID og for marshalling/unmarshalling av tekststreng (*test.C* og *map.C*).

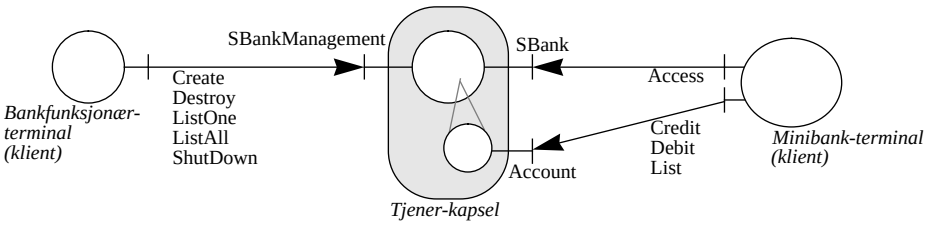
9.2.3 Test applikasjon 1: SBank

ANSAware installasjonen inneholder en demonstrasjons-applikasjon som kalles *SimpleBank*. Dette er en enkel bank-tjeneste. En bank har en sentral node og denne tenkes aksessert av minibank-terminaler og bankfunksjonær-terminaler fra ulike steder i landet.

Bank-tjenesten er representert som et bank-objekt som har to grensesnitt. For det første *SBankManagement* som er beregnet på bankfunksjonær-terminaler og som kan opprette, slette og lese opplysninger om kontoer, samt å ta ned hele bank-tjenesten. For det andre har vi *SBank* som er beregnet på minibank-terminaler og her kan en aksessere en konto, hvis korrekt kode oppgis.

Ved vellykket aksess til en konto (*Account*), returneres grensesnittreferansen for denne. Dette grensesnittet har operasjoner for innskudd, uttak og status for konto.

FIGUR 81. Sbank applikasjonen



Figuren nedenfor viser en IDL spesifikasjon for *SBank* grensesnittet. Den eneste operasjonen her er *Access* som tar et kontonummer og en personlig kode som argumenter og returnerer en grensesnittreferanse til en konto, hvis ikke feil har oppstått.

FIGUR 82. Bank grensesnittet

```
SBank : INTERFACE =
NEEDS SBankTypes FROM SBTypes;
NEEDS Account;
BEGIN
    AccessResult : TYPE = CHOICE OpStatus OF
    {
        OpSuccess => AccountRef,
        OpFailure => OpReason
    };

    Access : OPERATION
    [
        acct : AccountNumber;
        pin : PersonalIdentificationNumber
    ] RETURNS [ AccessResult ];
END.
```

Konto-grensesnittet har (som figuren nedenfor viser) fire operasjoner. Det er *Credit* for innskudd, *Debit* for uttak og *List* for å få opplysninger om konto (saldo og tid for siste aksess). Operasjonen *Destroy* sørger for at grensesnitt for konto blir passivisert etter bruk.

FIGUR 83. Konto grensesnittet

```
Account : INTERFACE =
NEEDS SBankTypes FROM SBTypes;
BEGIN
    ListResult : TYPE = CHOICE OpStatus OF {
        OpSuccess => AccountRecord,
        OpFailure => OpReason
    };

    Credit:OPERATION [ Amount : REAL ] RETURNS [ OpResult ];
    Debit:OPERATION [ Amount : REAL ] RETURNS [ OpResult ];
    List:OPERATION [ ] RETURNS [ ListResult ];
    Destroy: OPERATION [ ] RETURNS [ OpStatus ];
END.
```

For deklarasjon av typer som både *SBank*, *Account* og *SBankManagement* bruker, er det spesifisert et grensesnitt *SBankTypes*. Her er felles typer definert. *SBankTypes* har ingen operasjoner. IDL-spesifikasjon er vist nedenfor

FIGUR 84. Felles verdi-type definisjoner

```
SBankTypes : INTERFACE =
BEGIN
   OpStatus : TYPE = {OpSuccess, OpFailure};

   OpReason : TYPE = {NoSuchAccount, InvalidPin, Credited, Debited,
             InsufficientFunds, Created, Destroyed, Initiated,
             ResourcesExhausted, StaleAccountReference};

   OpResult : TYPE = RECORD [
      status : OpStatus,
      reason : OpReason
   ];

   AccountNumber : TYPE = CARDINAL;
   PersonalIdentificationNumber : TYPE = CARDINAL;
   AccountRecord : TYPE = RECORD [
      owner : STRING,
      balance : REAL,
      lastaccess : STRING
   ];
END.
```

Permanente objekter

Neste skritt er å identifisere permanente objekter og hvordan de forvaltes. Vi vet at konto-grensesnitt opprettes midlertidig for å gi aksess til bestemte kontoer, slik at disse kan gis temporær identitet. For *SBank*-grensesnittet derimot, kan vi ha permanente instanser. For klienter er det bank-grensesnitt (av typen *SBank*) som brukes som persistensrot (aktivisering framskaffes gjennom trading).

En tjener-kapsel startes opp manuelt og har ikke støtte for å bli startet fra *Factory*. *SBank*-grenesnitt blir eksportert til trader og får der (i tillegg til bruker-id og kapsel-id), navnet på noden som den kjøres på som attributt (*Node*). Vi tenker oss at det er mulig å identifisere en bestemt bank ved hjelp av den node som kjører *SBank* tjenesten. Implementasjon av tjener forutsetter at kun en tjener kjører pr. node. Vi kan altså bruke attributtet *Node* for a skille mellom ulike banker.

En forvaltningsklasse og instansiering kan se slik ut i notasjonen som ble beskrevet i avsnitt 6. 2. En C++ implementasjon som svarer til dette er realisert i kildefila *Bank-Perm.C* (se vedlegg 5 for kildekode).

FIGUR 85. Forvaltningsklasse for SBank

```
MGMT-CLASS SBank =
BEGIN
   VAR nodename : STRING;
   ACTIVATION = Trader("SBank","/ansa/testservices","Node=='%nodename'");
END;

PERMANENT odsBank = SBank("odslab2.cs.uit.no") EXPORT;
```

Språkbinding

I dette forsøket er språkbinding generert av den tidligere nevnte *stubcpp*. Komplett listing av språkbindingskode for bank-eksempelet er presentert i vedlegg 5.

Det som er mest interessant for klient-programmerer her er *IFC_Account.h* og *IFC_SBank.h*. Her finner man klassedefinisjon for objektreferanser, samt typedefinisjoner for verdi-typer. Figuren nedenfor viser noe av det som blir generert for konto-grensesnitt. (Makroen STUB_CONSTRUCTORS genererer konstruktør-metoder for klassen).

FIGUR 86.

```
#include “IFC_SBankTypes.h”

....

class AccountRef : public Ansa_Stub
{
public:
    STUB_CONSTRUCTORS(AccountRef)

    OpResult  Credit ( ansa_Real Amount );
    OpResult  Debit ( ansa_Real Amount );
    ListResult  List ( );
    OpStatus  Destroy ( );
};
```

Klient program

Klient-programmet er realisert i *client.C* som er gjengitt i vedlegg 5. Gangen i programmet kan forklares slik: Først spørres bruker etter kontonummer og kode. Deretter kalles *Access*-metoden i bank-objektet. Resultatet fra denne testes om aksess virkelig har blitt innvilget. Hvis ikke skrives det ut informasjon om årsak. Ellers kalles funksjonen *use_Account*. Denne presenterer brukeren for en meny med de ulike operasjoner som kan utføres på en konto. Hvert valg i denne menyen svarer til en operasjon i *Account*-grensesnittet.

9.2.4 Test applikasjon 2: Clone

Clone er et enkelt grensesnitt som vi bruker for demonstrere/teste operasjoner som aktiviserer og returnerer nye (temporære) grensesnitt, samt til å demonstrere/teste helt enkle operasjoner. *Clone* er en del av ANSAwares sett med demonstrasjonsprogram. Her finner vi grensesnittspesifikasjon og implementasjon av klient og tjenerprogrammer. Vi skal her anvende tjeneren slik den er i ANSAware installasjonen og skrive ulike klient-programmer egnet for vårt formål.

Clone-grensesnittet (vist i figuren nedenfor) har to operasjoner: En for kloning (*Clone*) som oppretter en ny grensesnitt-instans av samme type og med et nytt navn som angis som argument i operasjonsanropet, og en som ganske enkelt returnerer navnet. I tillegg til dette har vi *Destroy* som sørger for at grensesnitt blir passivisert og *DumpStats* som får tjenerprogram til å skrive ut statistikk.

FIGUR 87. Clone grensesnittet

```
Clone : INTERFACE =
BEGIN
    Clone: OPERATION [ name: STRING ] RETURNS [ CloneRef ];
    Name: OPERATION [] RETURNS [ STRING ];
    Destroy: ANNOUNCEMENT OPERATION [] RETURNS [];
    DumpStats: OPERATION [ message: STRING ] RETURNS [];
END.
```

Permanente objekter

Når *Clone* tjeneren starter opp opprettes et objekt med navnet “*First Clone*”. Dette eksporteres til trader. Dette objektet kan brukes som utgangspunkt for kloninger til nye objekter. Vi kan oppfatte denne som ei persistens-rot. Objekter som opprettes (klones) av en klient vil oppfattes som temporære. Figuren nedenfor illustrerer definisjonen av dette objektet. Vi viser ellers til vedlegg 6 (*ClonePerm.C*) for kildekode.

FIGUR 88. Klone rot-objekt

```
MGMT-CLASS Clone =
BEGIN
    VAR name : STRING;
    ACTIVATION = Trader("Clone", "/ansa/testservices","CloneName == '%name'");
END;

PERMANENT firstClone = Clone("First Clone");
```

9. 3. Litt om ytelsesmålinger

Ytelsesmålinger er gjort i et begrenset omfang, og det er gjort for å se på tidsforbruk for operasjonsanrop som aktiviserer og returnerer objekter (vi anvender slike målinger i avsnitt 9.4.1). Vi ønsker å sammenlikne ytelse med og uten bruk av objekt-adapter. Vi har laget et klient-program som anvender objekt-adapter beskrevet i avsnitt 9.2.4 og språkbinding til C++ og et klientprogram som anvender ANSAware direkte (PREPC). Testprogrammene som ble brukt er gjengitt i vedlegg 7 (*test.C* og *test.dpl*). De gjør i essens følgende:

- Gjøre et anrop på operasjonen *Clone* 100 ganger og registrere tidsforbruket for det siste av de 100 anropene. De returnerte objektreferansene lagres i en tabell, ellers vil de bli søppelsamlet av objekt-adapter.
- Gjenta dette 20 ganger (til vi har aktivisert 2000 objekter).

Dermed har vi målt tidsforbruk pr. operasjonsanrop, for et stigende antall samtidige aktive objekter, med et intervall på 100. Vi har utført disse programmene 8 ganger og anvendt middelverdien for hvert intervall. Dermed kunne vi få et mer sikkert anslag over tidsforbruket og vi kunne kontrollere varians.

Mange ukjente faktorer spiller inn på tidsforbruket for et anrop. Nettbelastning, samtidige brukere og prosesser på samme maskin, disk-aksess er kjente faktorer som kan gi store utslag. For å få standardavviket ned på et akseptabelt nivå ble eksperimentet gjort på to *HP 755* datamaskiner på et laboratorum med avskjermet nettsegment og hvor ingen andre brukere var logget inn.

9. 4. Diskusjon

Vi skal i dette avsnittet diskutere rammeverket i forhold kvalitetsparametre som ytelse, skalering, anvendelighet og generalitet. Vi vil se på enkelte sterke og svake sider ved vårt rammeverk og undersøke muligheter for forbedringer der dette synes hensiktsmessig.

9.4.1 Ytelse

Enkle anrop

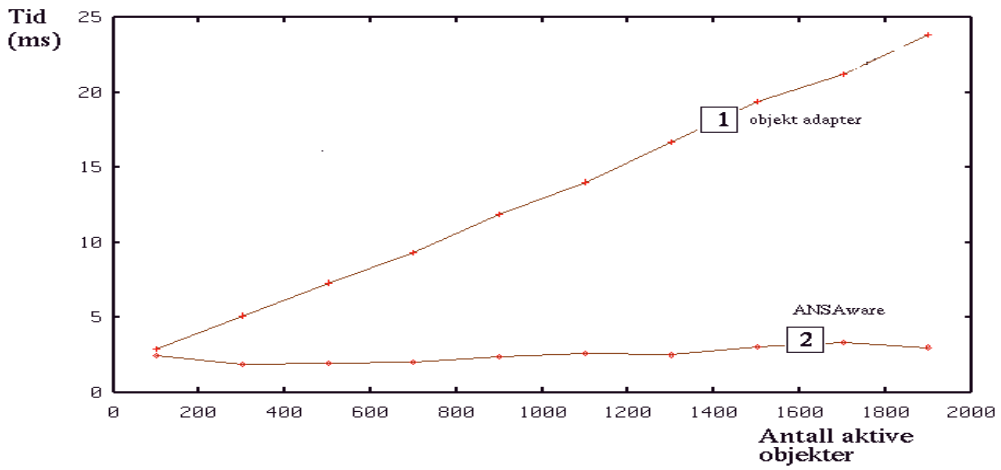
Vi skal føst se på ytelse for enkle anrop (som ikke behandler andre grensesnitt). Rammeverket for objekt-adapter som er beskrevet tidligere i denne avhandlinga genererer surrogat-identifi katœr for hvert proxy-objekt som kan inngå i globale objekt-identifi - katorer. Vi har ei avbildning fra disse til proxy-objekter. Dette fører til en mulig ekstra kostnad med hensyn til ytelse. Avbildning skjer ved et oppslag i en datastruktur hvor pekere til proxy-objekter er lagret. Slike oppslag kan optimaliseres ved hjelp av effektive algoritmer og datastrukturer. Vi kan for eksempel anvende søke-trær eller hashing (jfr. [Aho83, LEDA3.0]). Da blir tidskompleksiteten for oppslag henholdsvis *O(log n)* for trær eller *O(1)* for hashing. Ved hjelp av slik effektivisering kan denne ekstrakostnaden alltid holdes på et nivå som er ubetydelig i forhold til kostnaden ved RPC[1].

For direkte språkbindinger (jfr. avsnitt 7.1.1) kan slike oppslag unngås, da objektreferanser her kan inneholde pekere direkte til proxy-objekter. Et spørsmål er om dette kan unngås helt, også ved bruk av " kanoniske" språkbindinger ved å bruke peker til proxy som OID. En slik OID er ikke så sikker som en surrogat, fordi den ikke vil være gyldig lenger hvis objekt-adapter passiviseres[2].

Anrop som returnerer objekter

Vi har altså ikke noen ytelseskostnad av betydning for enkle anrop, men der anrop returnerer objekter, skal det aktiviseres en proxy for hvert av disse objektene. Vi har foreløpig gjort det valg at kun en proxy og kun en surrogat-OID på et gitt tidspunkt skal identifi sær et gitt objekt. Dette har den ulempe at ved aktivisering av nytt proxy-objekt må en sammenlikne med eksisterende proxy-objekter for å eliminere duplikater. I prototype-implementasjonen gjøres dette ved å sammenlikne med hver enkelt av de eksisterende proxyer. Dette har tidskompleksitet *O(n)*, noe som gjør at ytelsen svekkes betydelig ved store antall aktive objekter (lineær økning av tidsforbruk). Eksperimenter bekrefter dette (se også avsnitt 9. 3). Figuren nedenfor illustrerer hvordan tidsforbruket øker ved økende antall aktiviserte objekter i objekt-adapter (kurve 1). Dette kan sammenliknes for en tilsvarende applikasjon som er implementert uten objekt-adapter (kurve 2). Her er tidsforbruket tilnærmet konstant ved økende antall objekter.

FIGUR 89. Ytelsesmålinger



Denne kostnaden kan reduseres ved hjelp av bedre implementasjoner. For det første kan vi tenke oss at vi partisjonerte mengden objekter over fl ere instanser av *OIDManager* som antydet i avsnitt 5.1.7. Vi kunne på denne måten ha en delmengde av den totale objektmengden assosiert ved hvert meta-objekt. Dette alene gir kun marginale forbedringer, da antall meta-objekter vil være lite i forhold til antall objekter. Tidskompleksiteten vil være *O(n/k)* der *k* er antall meta-objekter. For det andre kan vi anvende effektive algoritmer og datastrukturer som f.eks. trær eller hashing for å miske kostnaden. Vi kan da oppnå tidskompleksitet *O(log n)* eller *O(1)*, men ulempen er at vi legger en ekstra byrde på programmerer. For hver type lokalenhet/proxy må vi da implementere effektivt oppslag eller defi nær ei ordning eller en hash-funksjon, som et eventuelt generisk søke-tre eller hash-tabell ville være avhengig av.

1. I følge [Dale93] og egne forsøk, kan vi anslå tidsforbruket for et enkelt anrop til å være ca. 1 millisekund på maskinvaren som vi bruker her.
2. Det forutsettes her at objekt-adapter kan lagre sin tilstand i persistent lager (inkludert surrogater).

Forvaltning og ytelse

Vi ønsker å minimalisere forsinkelse av operasjonsanrop, som skyldes forvaltning. Det oppnår vi ved i størst mulig grad søke å aktivisere objekter før de trengs, og da på en måte som ikke forsinker andre aktiviteter. Vi ønsker på den andre siden å unngå unødvendig aktivisering, fordi aktiviseringer forbruker ressurser. Vi ønsker å innvirke minst mulig på ANSAware-applikasjonens forbruk av ressurser.

I følge avsnitt 3. 5 kan vi karakterisere forvaltnings-policies langs følgende dimensjoner:

- Sein eller tidlig oppdagelse av passivisering.
- Lat eller ivrig aktivisering.

Det første ønsket møtes ved tidlig oppdagelse og ivrig aktivisering. Hvis dette kan gjøres i parallell med andre aktiviteter, vil det ikke forsinke anrop. Det andre ønsket møtes ved å utsette aktivisering til objektet trengs (lat aktivisering). Vi ser her at ivrig aktivisering er i konflikt med det andre ønsket.

Tidlig oppdagelse oppnår en ved bruk av monitorering, notifikasjon eller en kombinasjon av disse. Dette behøver ikke føre til forsinkelse og kan utnytte paralellitet. Det medfører en ekstra kostnad med hensyn til trafikkmengde og antall aktive grensesnitt. En tjener må ha grensesnittreferanse til alle som skal ha notifikasjon. Dette er ikke skalerbart med hensyn til antall klienter. Vi kan også observere at disse mekanismer ikke alltid vil oppdage passivisering før klient forsøker anrop. Det tar alltid litt tid fra objekt passiviseres til klient vet om det og klient kan tenkes å gjøre anrop i dette tidsrommet. Derfor må tidlig oppdagelse kombineres med sein oppdagelse.

Sein oppdagelse (“management by exception”) skjer når operasjoner feiler fordi grensesnittreferansen ikke er gyldig. Sein oppdagelse impliserer lat aktivisering[3]. Når operasjoner feiler fører dette til ekstra ventetid. Først operasjonen som feiler, så de nødvendige forvaltningsoperasjoner og så kan anropet forsøkes på nytt.

For å analysere med hensyn til tidsforbruk, er det nyttig å kunne anslå kostnad for ulike typer forvaltnings-operasjoner. Vi har gjort følgende observasjoner med hensyn til tid:

- Anrop (RPC): Noen få millisekunder
- Aktivisering (RPC + disk aksess): Noen hundre millisekunder for kapsler.
- Sein oppdagelse (feil, timeout): Noen sekunder hvis kapsel har terminert. Noen millisekunder hvis kapsel eksisterer på den angitte nettverksadressen.

Det er mest å hente på å unngå anrop på objekter hvor deres kapsel har terminert. Tidlig oppdagelse har altså mest for seg for objekter der passivisering skyldes av kapsel har terminert. Hvis vi kombinerer notifikasjon for kapsler og utnyttelse av kunnskap om avhengighet (jfr. avsnitt 3.5.3), vil dette gi en betydelig gevinst. Dersom en kapsel terminerer kan vi automatisk slutte at objekter avhengige av kapsel har blitt passivisert.

En vanlig strategi for optimalisering i databasesystemer kalles “prefetching” (jfr. [Liskov93]). Det vil si at ved lesing av et objekt fra databasen, leses samtidig en del andre relaterte objekter som det er sannsynlig at klientprogrammet vil ha bruk for i nærmeste framtid. Gjennom å lagre relaterte objekter sammen på disk, vil en og

3. Aktivisering betyr her, sett fra klientens side, enten at et objekt aktiviseres eller at klienten finner et objekt som har blitt aktivisert av andre (trading, relokering o.l.).

samme disk-aksess kunne lese både objektet som er av interesse og relaterte objekter. Dermed kan en redusere antall aksess til databasen betydelig og forbedre ytelsen.

Vi kan generalisere dette begrepet og anvende det på ANSAware omgivelsene. "Pre-aktivisering" vil si at når et objekt aktiviseres, kan man samtidig aktivisere relaterte objekter som en tror klient vil ha bruk for i nærmeste framtid. Her vil en først og fremst oppnå ytelsesgevinst ved at aktivisering kan gjøres i parallell med andre aktiviteter i klient, før klient skal anvende objektet. For å utnytte denne muligheten kreves det kunnskap om applikasjonen, ut over det som går fram av skjema. Det er relevant å stille følgende spørsmål:

- Gitt at klient får aktivisert et objekt. Finnes det da andre objekter som det er sannsynlig at klient vil ha bruk for i nær framtid? En kan da initiere aktivisering av disse hvis de er passive.
- Fører aktivisering av et objekt automatisk til aktivisering av et annet? Det vil si: Aktiviseres de sammen? RM-ODP åpner for at flere objekter kan eksistere innenfor et og samme cluster. Cluster er her enhet for aktivisering, og dermed vil alle objekter innenfor et cluster aktiviseres samtidig. Det er ønskelig å kunne utnytte aktiviseringer som en " får med på kjøpet". I så fall må en ha en måte å få tak i disse aktiviseringene.

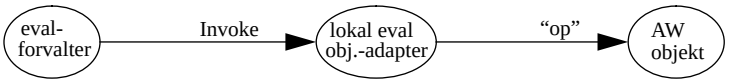
Ved oppdagelse av passivisering kan det være verdt å se om det er relaterte objekter som sannsynligvis har blitt passivisert samtidig. For eksempel vil objekter innenfor et cluster alltid passiviseres samtidig.

Både avhengighet og pre-aktivisering krever at kunnskap om applikasjonen " programmeres" inn i objekt-adapter (jfr. kapittel 6). Hvis vi ser på notasjonen for forvaltningsklasser og permanente objekter i avsnitt 6. 2, kan vi tenke oss at denne ble utvidet med konstruksjoner for å spesifi sere avhengighet og "pre-aktiviserings-relasjoner".

Ytelse og kanonisk språkbinding

Kanoniske språkbindinger (jfr. avsnitt 7.1.2), der klient/eval. forvalter og lokal eval/objektadapter er atskilt, har den ulempen at for hver operasjon så må det gjøres to anrop. Først *invoke*-operasjonen over grensesnittet til lokal eval/objektadapter og så en operasjon over grensesnittet til det aktuelle ANSAware objektet. Figuren illustrerer dette:

FIGUR 90.



Vi har følgende alternativer for effektivisering av dette:

- Lokal eval og objektadapter samlokaliseres med eval-forvalter.
- Samle flere operasjoner til et lokalt system i et anrop på invoke. En slik mekanisme foreslås av [Bogle94]. Dette er basert på den observasjonen at operasjonen som klienten ikke forventer noe resultat fra, kan utsettes til klienten anroper operasjon som returnerer et resultat.
- [StaGif90] viser at ytelse kan forbedres (kommunikasjon reduseres) ved å gå over fra " fjerne prosedyrekall" til " fjern evaluering". Det vil si at klienten sender hele program-biter til tjener for evaluering. FRIL-arkitekturen er tenkt slik. I følge f.eks. [ElKa91] har FRIL-tjenere såkalte programmerbare grensesnitt, det vil si i den forstand at de kan akseptere komplette FIOL-uttrykk for evaluering.

9.4.2 Skalerbarhet

En definisjon på skalerbarhet finner vi i [Coul88]. Her omtales skalerbarhet som en distribusjonstransparens på denne måten:

> *Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.*

Det dreier seg altså om hvordan systemet/applikasjonen skal kunne vokse i omfang, uten at vi trenger å endre design for at det fortsatt skal fungere tilfredsstillende. I vår kontekst er vi interessert i hva det betyr at applikasjonen (som objekt-adapter skal bidra til å integrere) får et stort omfang. Vi er interessert i å vite i hvilken grad vår modell og design setter grenser for applikasjonens størrelse og vi er interessert i å finne måter å maksimalisere slike grenser og eventuelt eliminere dem. Grenser nås når antallet objekter (avbildninger) som objekt-adapter forvalter blir så stort at enten ytelse blir uakseptabelt dårlig eller at minnet er oppbrukt.

Temporære objekter

Et design-valg som angår skalerbarhet er at hvert enkelt proxy-objekt er eksplisitt representert i objektadapter, noe som både forbruker minne og som påvirker ytelse (jfr. avsnitt 4.2, avsnitt 5.1.3 og avsnitt 8.1.1). Hvis avbildninger for alle objekter må lagres i objekt-adapter, betyr dette at det er ei grense for hvor mange objekter en objekt-adapter kan definere avbildning til på et gitt tidspunkt.

Vi kan likevel påstå at objekt-adapter er skalerbar ut fra bestemte forutsetninger. Det vil si at applikasjonen kan bli svært stor og omfatte et svært stort antall objekter, uten at antall avbildninger (proxy-objekter) på et gitt tidspunkt blir stor. Dette henger sammen med at klienter på et gitt tidspunkt bare trenger å "se" en avgrenset del av applikasjonens objekter. Ut fra persistensmodellen (jfr. avsnitt 2.4), vil en stor majoritet av objekter være temporære. En klient vil ofte bare trenge å aksessere en liten del av applikasjonens objekter, og den tiden en klient refererer til et temporært objekt vil ofte være kort. Dersom objekt-adapter effektivt kan søppelsamle proxy-objekter så snart de ikke trengs mer, er det mulig å referere til langt flere objekter innenfor en klient-sesjon enn det objekt-adapter klarer å håndtere. Gitt at applikasjoner har de nevnte egenskapene, kan vi si at objekt-adapter designet vårt er skalerbart i praksis.

Heterogen OID versus proxy-objekter

Det vil fortsatt være ei grense for antall objekter som er referert til samtidig. Man kan f.eks. tenke seg spørringer mot databaser som gir store mengder temporære objekter. Det er ikke sikkert klient vil bruke alle disse, men det vil bli generert objekt-referanser for disse. Vi har tidligere antydet muligheten for heterogene globale OID og for å kode grensesnittreferanse[4] direkte inn i globale OID (jfr. avsnitt 4.2, avsnitt 8.1.1 og avsnitt 9.4.1). Dette ser ut til å være mulig ut fra den antakelse at temporære objekter ikke trenger å forvaltes (jfr. avsnitt 3.5.3). Dermed kan en unngå eksplisitt proxy-objekt for disse. Objekt-adapters rolle for disse vil i så fall være å kode og dekode objekt-identifikatorer. Permanente objekter derimot må forvaltes av objekt-adapter, da ANSAwares objekt-identitet ikke er sterk nok (jfr. avsnitt 3.4). Dermed vil vi for ANSAware applikasjoner måtte operere med to ulike former for lokal OID. Det vil si surrogater/pekere til proxy-objekter for permanente objekter og grensesnittreferanser/*LocalUnit* for temporære objekter.

Til dette alternativet kan det innvendes at for konfigurasjoner der det er direkte binding mellom klient og objekt-adapter, er det ikke noe å vinne mht. minneforbruk. En objekt-referanse vil likevel inneholde grensesnittreferanse/lokalenhet. For temporære

4. Eller LocalUnit som er en abstraksjon over lokal representasjon av objekt (se avsnitt 5.1.4).

objekter vil en proxy kun bestå av en lokalenhet. For konfi gurasjoner med atskilte entiteter, vil en unngå å lagre avbildninger i objekt-adapter, men hver objekt-identifi kator som klient-prosess får, vil for ANSAware ta betydelig større plass enn om en opererte med proxy-obejekter, fordi de inneholder hele grensesnittreferanser, altså all informasjon en trenger for å lokalisere og interagere med et tjener-objekt. I vår modell har ikke klient-program bruk for denne informasjon i og med at interaksjon med tjener-objekter skjer via objekt-adapter gjennom et operasjonelt grensesnitt.

Mer generelt kan vi si at det kommer an på den lokale objekt-identifi kators størrelse om det med hensyn til ytelse og skalerbarhet, lønner seg å representere objektet som proxy-objekt i objekt-adapter. For underliggende systemer som generer surrogater for objektene, vil proxy-objekter være unødvendig, men for systemer som ANSAware der identifi katoren inneholder lokaliserings-informasjon (en kompleks struktur), kan bruk av proxy-objekter gi bedre ytelse fordi mindre enheter behøver å overføres mellom objekt-adapter og klient. Dette må vurderes opp mot ytelseskostnad ved å sjekke om det eksisterer proxy for et gitt objekt ved aktivisering (jfr. avsnitt 9.4.1). Det igjen avhenger at antall samtidige aktiviserte objekter.

9.4.3 Anvendelighet

Et aspekt ved kvalitet er rammeverkets anvendelighet. Vi er interessert i tilfeller der bruk av rammeverket svekker anvendelighet, det vil si der bruk av applikasjoner kompliseres eller avgrenses, og vi er interessert i tilfeller der bruk av rammeverket forbedrer anvendelighet.

Vi har tidligere behandlet ytelse, og skalerbarhet. Dette har også sammenheng med anvendelighet. Vi kan f.eks. si at begrensing av skalerbarhet også innebærer svekket anvendelighet, det vil si bruk av applikasjoner avgrenses.

Transparens

Transparens har betydning for anvendelighet. Aksess transparens har vi både ved bruk av vårt rammeverk og ved bruk av ANSAware direkte. Uten aksesstransparens ville systemet vært mer komplisert å bruke, fordi klient programmerer måtte forholde seg til ulike interaksjonsmekanismer, og dermed mindre anvendelig.

Objekt-adapter kan beskrives og implementeres uavhengig av hvilken språkbinding en velger å bruke og ulike språkbindinger kan eksistere sammen med objektadapter på samme tid. Dette kan sies å være en form for transparens som gjør objekt-adapter rammeverket mer anvendelig. Bruksområdet kan spenne over flere språkomgivelser.

Transparent forvaltning er et betydelig bidrag til anvendelighet. Det vi gjør er å skille forvaltning og bruk av objekter. Et klient program kan på denne måten beskrives uavhengig av forvaltningsoperasjonene det er avhengig av og forvaltningsoperasjonene kan beskrives uavhengig av klient programmet. Dette kan gjøre det enklere å bruke applikasjoner, det kan gjøre forholdet mellom klient-program og forvaltningsoperasjoner mer fleksibelt, og det kan legge bedre til rette for gjenbruk av program-deler. Med et verktøy som beskrevet i avsnitt 6. 2, kan også implementasjon av forvaltningsoperasjoner og policy automatiseres, noe som vil kunne lette integrasjonsarbeidet.

9.4.4 Generalitet

Med ordet generalitet mener vi i hvilken grad objekt-adapter rammeverket lar seg anvende på andre lokale informasjonssystemer enn ANSAware. Vi skal ikke gi en fullstendig analyse av dette her, men fokusere på rammeverkets evne til å representere og generere ulike former for objekt-identitet. Dette er et viktig aspekt ved generalitet.

Rammeverket baserer seg på at proxy-objekter er eksplisitt representert, at objekt-adapter har en tabell over aktive proxy-objekter og at det genereres surrogat-OID for hvert av dem (jfr. avsnitt 5.1.7). Peker til proxy-objekt kan også brukes som global objekt-referanse (jfr. avsnitt 7.2.1). Vi har også diskutert muligheten for lokale identifikatorer inngår direkte i heterogene globale OIDer og at objekt-adapter ikke lagrer avbildninger som eksplisitte proxy-objekter (jfr. avsnitt 9.4.2).

Med hensyn til generalitet, kunne det være ønskelig at rammeverket tillot begge disse alternativer. I noen tilfeller er eksplisitte proxy-objekter helt unødvendig, f.eks. ved integrasjon av objekt-orienterte databasesystemer hvor objekter er identifisert med surrogater Ved integrasjon av OODMBS som tilbyr sterk OID, vil det strengt tatt ikke være behov for objekt-adapter slik den er beskrevet i denne avhandlinga. Det man trenger er funksjoner som genererer global OID ut fra lokal OID og som trekker ut lokal OID fra globale OID. I andre tilfeller kan det tenkes at objekt-adapter understøttet proxy-objekter for permanente objekter, men ikke temporære (jfr. diskusjonen i avsnitt 9.4.2).

Et bidrag til et mer generelt rammeverk vil være å understøtte muligheten for å generere og tolke globale OID som er basert på lokale OID. Ut fra dette er det grunn til å revurdere designet av *OIDManager* klassen (jfr. avsnitt 5.1.7). Her kan f.eks. tenke oss at metoden *getNewOID* (som genererer globale OID), som en opsjon kunne ta en *LocalUnit* som argument. Typedefinisjonen for globale OID må da også endres, fra å være et heltall til å være en variabel streng med oktetter. Denne må kunne representere både surrogat baserte OID og OID basert på *LocalUnit*.

Språkbindinger

Ser vi på de språkbindingene vi har presentert i denne avhandlinga, kan vi lære mer om objekt-identitet og generalitet.

I bindinga til C++ (jfr. avsnitt 7.3.3) inneholder objektreferanser direkte pekere til proxy-objektet. De kunne også inneholdt surrogat-OIDer, men dette synes unødvendig når bindinga er direkte og medfører en ekstra kostnad med hensyn til ytelse fordi man for hvert operasjonsanrop måtte gjøre ei avbildning fra OID til proxy-objekt. For C++ språkbindinga kan vi gjøre følgende observasjoner:

- Generering av surrogat basert OID i objekt-adapter er unødvendig.
- Språkbindingas objekt-referanser forutsetter at proxy-objekter eksisterer og inneholder ikke lokale OIDer direkte. Hvis bare direkte språkbindinger blir anvendt, har ikke dette særlig betydning, siden lokal OID likevel må lagres i samme prosess. Men dette påtvinger design av objekt-adapter, bruk av eksplisitte proxy-objekter, noe som kan være for restriktivt, dersom en ved bytte av språkbinding for samme applikasjon og objekt-adapter, ønsker å gå over til en kanonisk språkbinding der klient er atskilt fra objekt adapter.

Problemet i det siste punktet kan løses ved å gjøre objekt-referanse klassen mer fleksibel, det vil si at det er mer valgfritt hva den kan inneholde: Enten peker til proxy-objekt eller lokal-enhet. Den første observasjonen leder oss til spørsmålet om et objekt-adapter skal generere surrogat-OIDer, eller om dette kan overlates til eventuelle språkbindinger som ønsker slik OID. Argument for at dette hører heime i språkbindinger er det er språkbindingers oppgave å definere objekt-referanser for klienter og at ikke alle språkbindinger bruker surrogat-OID som objekt-referanser. Argumenter mot er mer pragmatiske: At flere ulike språkbindinger kan ha bruk for surrogater og at det av den grunn eller av implementasjonstekniske grunner vil være minst kostnad forbundet med å integrere dette i objekt-adapter rammeverket.

I bindinga til FRIL som ble diskutert i kapittel 8, kom vi fram til at en heterogen form for global OID var ønskelig. Språkbindinga vil ha støtte for å omforme objekt-adapter-

ens OID-format, til det formatet som brukes i kommunikasjon med evalueringstjenesten (jfr kapittel 8. 5). Den transparente representasjonen av OID i grensesnittet til klienter/eval-tjeneste setter ingen restriksjoner på hva slags OID objekt-adapter kan tilby til føderasjonen.

Aktiviserings-policies

I avsnitt 4.3.1 ble de fire aktiviserings-policies fra CORBA BOA beskrevet. Vårt objekt-adapter rammeverk har støtte for to av disse: "shared-server" og "persistent-server". Et aspekt ved generalitet er i hvilken grad rammeverket understøtter også de andre policies. Når det gjelder "unshared server" betyr dette ganske enkelt at aktivisering av implementasjon gjøres sammen med aktivisering av objekt. Vårt rammeverk støtter egentlig dette (det vil si objekter kan aktiviseres uten referanse til noe eget tjener objekt, eller at tjener objektet er implisitt).

"Server per method" er mer problematisk. Her er rammeverket for aktivisering (*PermObj*) meningsløst, fordi det modellerer aktivisering av objekter (eller tjenere som behandles som objekter), og da kun aktivisering av et objekt per proxy. Server per method betyr aktivisering for hvert metode/operasjons-anrop.

9.4.5 Andre problemstillinger

Vi har til nå gjort den antakelsen at forvaltning (oppdagelse av passivisering og reaktivisering) er unødvendig for temporære objekter. Et tema som vi bare så vidt har berørt er passivisering av objekter fra objekt-adapter (gjennom en policy eller ved at klient ber om det). Dette har gjeldt permanente objekter.

Vi bør revurdere denne antakelsen, fordi i noen tilfeller kan det være ønskelig å kunne passivisere også temporære objekter, for eksempel når klienter ikke lenger refererer til dem, og proxy-er for disse skal søppelsamles. Det vil være ulikt hvordan applikasjoner behandler slike objekter. Noen kan tenkes å understøtte passiviseringsmekanismer og deling av slike grensesnitt mellom ulike klienter, mens andre kan tenkes å overlate ansvaret til klienten å gi beskjed om at objektet ikke lenger trengs, f.eks. gjennom en egen grensesnitt-operasjon. Dette gjelder for eksempel *SimpleBank* applikasjonen (jfr. avsnitt 9.2.3). Her har grensesnittet *Account* en operasjon *Destroy* som fjerner grensesnittet når operasjonen anropes.

Vi kan velge å overlate til klient-programmerer å anrope slike operasjoner når det er nødvendig, men dette er ikke noen ideell løsning. Det ville være mer ønskelig at dette blir gjort automatisk og transparent for klient når proxy-objekter blir søppelsamlet. Det vil si denne oppgaven ble lagt til objekt-adapter. Dette ville f.eks. forenkle oppgaven til klient-programmerer og redusere mulighetene for feil. Et poeng som vi har vært inne på tidligere er at forvaltning bør være ortogonalt til klient-program.

9. 5. Oppsummering

I dette kapitlet har vi for det første vist hvordan rammeverket kan implementeres. Vi har realisert en prototype-implementasjon av objekt-adapter rammeverket, vi har realisert ei språkbinding til C++ og vi har skissert hvordan ei FRIL-binding kan realiseres. Vi har også sett hvordan vi kan anvende rammeverket på konkrete applikasjoner.

For det andre ble rammeverket også diskutert i forhold til ytelse, skalerbarhet, anvendelighet og generalitet. I implementasjonen blir ytelsen betydelig svekket ved store mengder samtidig aktiviserte objekter fordi ved anrop og spesielt ved aktivisering, må det gjøres oppslag for å finne proxy-objekt eller avgjøre om objektet allerede eksisterer. Dette kan avhjelpes med mer avanserte datastrukturer for søk blant proxy-objekter, som f.eks. hashing. Vi har også diskutert hvordan forvaltning virker inn på ytelse. Sein

versus tidlig oppdagelse og lat eller ivrig aktivisering er parametre som er viktig for ytelse. Kunnskap om avhengighet mellom objekter og "pre-aktivisering" kan f.eks. anvendes for å redusere ytelseskostnad ved forvaltning.

I kanoniske språkbindinger må en gjøre to operasjonsanrop per anrop, noe som kan være kostbart. Alternativer for effektivisering er samlokalisering eller utsettelse av operasjonsanrop. Vi vet også at "Fjern evaluering" kan være et bedre paradigme for interaksjon enn "fjerne prosedyrekall".

Vi har diskutert skalerbarhet. Her kan rammeverket sette grenser fordi hvert objekt blir representert eksplisitt som proxy-objekt. Objektadapter kan likevel skalere godt i den grad bare små fraksjoner av applikasjonens totale objekt-mengde er aktiv på samme tid (dette forutsetter at vi også har effektiv søppelsamling). Vi har også diskutert å sløyfe proxy-objekt for temporære objekter og bruke lokal OID (grensesnittreferanser) direkte som global OID. Dette vil ikke alltid hjelpe, men kan i noen tilfeller redusere skalerbarhet og ytelse.

Når det gjelder anvendelighet er skillet mellom forvaltning og klient-program, samt mekanismer for transparent et viktig bidrag. Ytelse og skalerbarhet har også sammenheng med anvendelighet.

I forbindelse med generalitet har vi diskutert mulig støtte for heterogen OID og vi har i den forbindelse sett på språkbindinger og hvordan disse representerer objekt-referanser. Ser vi på aktiviserings-policies slik de er definert i CORBA BOA, skiller "server per method" seg ut. Designet med *PermObj* klasser kan ikke brukes her.

Til slutt har vi påpekt at rammeverket mangler støtte for passivisering av temporære objekter, noe som er ønskelig i noen tilfeller.

Kapittel 10
Konklusjoner og videre arbeid

Dette avsnittet oppsummerer hva vi har gjort i denne avhandlinga, sier noe om hva vi har lært og skisserer hva vi tror arbeidet kan danne grunnlag for av videre arbeid.

10. 1. Oppsummering

Målet med dette arbeidet var å utvikle et rammeverk for modellering, design og implementasjon av avbildningsmekanismer (objekt-adapter og tilhørende programmeringsgrensesnitt) mellom ANSAware applikasjoner og føderative omgivelser som FRIL. Dette for a få kunnskap om problemområdet avbildning og hvordan dette effektivt kan utføres.

Vi har gjort et skille mellom to oppgaver som avbildningsmekanismene utfører, som er representert med to komponenter: Objekt-adapter og språkbinding (jfr. kapittel 4). Objekt-adapter har ansvaret for å realisere transparens med hensyn til persistens og tilby sterk global identitet for objekter i lokale systemer. Språkbindinger realiserer aksess-transparens og tilbyr en representasjon av objekt-referanser, operasjoner og verdier i det språket klienten anvender, det vil si språkbindinger representerer programmeringsgrensesnitt.

Vi har videre innført begrepet 'proxy-objekt'. Proxy-objekter vedlikeholdes av objekt-adapter og er representanter for lokale objekter i en føderativ omgivelse. De har ansvaret for identifi kasjon og eventuell forvaltning (aktivisering, passivisering) av lokale objekter. Objekt-adapter (gjennom språkbindinger) tilbyr klient-programmer aksess til objekter gjennom objekt-referanser. En objektreferanse vil identifi sær et proxy-objekt. Vi har presentert en interaksjonsmodell med tre basale (konseptuelle) operasjoner i forhold til objekt-referanser. Disse er invoke (operasjonsanrop), bind (binding til referanse) og unbind. Dermed kan en bl.a. modellere temporære referanser til permanente objekter.

Vi har realisert et gjenbrukbart objekt-orientert rammeverk som representerer design og delvis implementasjon av objekt-adapter (kapittel 5). Denne påbygges med applikasjonsspesifi kk kode for å bli komplett. Denne koden er først og fremst for forvaltning av objekter som skal fremstå som permanente og som skal kunne opptre i klient-program som persistens-røtter. Slik forvaltning kan vi modellere som såkalte forvaltnings-klasser. Vi har presentert en egen notasjon for å beskrive forvaltningsklasser og permanente objekter (kapittel 6). Vi tror at det er mulig å utvikle verktøy for å generere forvaltnings-kode ut fra denne notasjonen.

Vi har undersøkt oppbygning og virkemåte til språkbindinger (kapittel 7). Vi kan skille mellom to slags språkbindinger: Direkte bindinger hvor klient og objekt-adapter kjører i samme prosess og "kanoniske" bindinger hvor klient og objekt-adapter i det minste er logisk atskilt og interagerer gjennom et veldefi nert grensesnitt som vi kan si representerer et kanonisk språk. Vi har realisert språkbinding til C++ som er et eksempel på direkte binding. Her er objektreferanser representert som klasser og *bind/unbind* semantikk ligger implisitt i metoder som oppretter, sletter, tilordner, kopierer instanser av objekt-referanse klassene, eller som returnerer objekter.

I binding til kanonisk språk er objekt-referanser, sett fra klientens side, transparente strenger med oktetter som ikke har noen semantisk mening. Objekt-referanser kan også være permanente og eksisterer da uavhengig av prosesser eller sesjoner. Vi har sett spesielt på binding til FRIL (kapittel 8) og her har vi åpnet for heterogene lokale objekt-identifikatorer i grensesnittet. Vi har spesifisert protokollen mellom evalueringstjeneste og applikasjonsobjekt som et ANSAware IDL grensesnitt. En prototype implementasjon vil også kunne realiseres ved hjelp av ANSAware. Protokollen er uavhengig av spesifikke applikasjoner. De konseptuelle oeprasjonene *bind*, *unbind* og *invoke* er eksplisitt representert som operasjoner i denne protokollen. Vi har også skissert et rammeverk for implementasjon av FRIL-binding. Sentralt her er en generisk stub.

10. 2. Konklusjoner

Vi har i denne avhandlinga utviklet et rammeverk for forståelse og realisering av objekt-adapter med programmeringsgrensesnitt. Arbeidet med dette har blant annet vist oss at:

10.2.1 Forvaltning og identifikasjon

Prinsippet om nåbarhetspersistens er viktig for å gi et bilde av ANSAware objekter som persistente. På grunn av dette trenger vi ikke sterk OID for alle objekter. For objekter som skal opptre med permanent OID, trenger vi mekanismer for transparent oppdagelse av passivisering og reaktivisering. Vi har da også behov for ei avbildning mellom global og lokal OID som kan relokeres (lokal OID kan skiftes ut). Dette ivaretas av proxy-objekt. Følgende gjelder for øvrig forvaltning:

- Det er et betydelig bidrag til effektiv implementasjon og anvendelighet, at forvaltning kan spesifiseres uavhengig av klient-program. Man spesifiserer bare måten forvaltning blir gjort på, og rammeverket vil sørge for transparent bruk av dette, når klientprogrammet anvender objekter. Hvis notasjonen for spesifikasjon av forvaltning fungerer etter hensikten, vil denne være til hjelp.
- Dimensjonene tidlig/sein oppdagelse og ivrig/lat aktivisering ser ut til å ha betydning for ytelse. Tidlig oppdagelse og ivrig aktivisering kan gi bedre ytelse hvis operasjonene kan utføres i parallell med andre aktiviteter, men ivrig aktivisering har den ulempe at den kan belaste systemet med unødvendig aktivisering. Det ser ut til at det er mest å hente på tidlig oppdagelse for kapsler og at vi kan utnytte kunnskap om avhengigheter og pre-aktiviseringer for å effektivisere (se. for øvrig avsnitt 9.4.1).

Det har videre vært nyttig å se på objekt-identitet fra fire sider: (1) Proxy-objekt som er objektadapters representasjon av det lokale objektet i føderative omgivelser. (2) Objektreferanse/global OID som er språkbindingas representasjon av objektet. Denne identifiserer (eller består av) proxy-objektet. (3) Grensenittreferanse (lokal aktivisering) og (4) identifikasjon av passiv tilstand som kan brukes for å aktivisere lokalt objekt.

10.2.2 Realisering og evaluering

Implementasjon av objekt-adapter rammeverk og språkbinding, samt forsøk på anvendelse av dette viser at objekt-adapter rammeverket er et anvendelig hjelpemiddel for effektiv implementasjon av objekt-adaptere. Dette representerer et gjenbrukbart design og delvis implementasjon. Det forutsettes at språkbindinger kan genereres av en stubkompilator for at rammeverket skal være virkelig anvendelig. Denne sørger for transparens (stubs, marshalling, bruk av forvaltning).

Ut fra evaluering av rammeverk og implemtasjon (kapittel 9), kan vi trekke ut følgende:

- Siden en ved aktivisering av objekter som returneres fra operasjonsanrop, sjekker om objektet er aktivisert fra før, er det av stor betydning at det bygges inn en effektiv søkealgoritme som for eksempel hashing.
- Av hensyn til skalerbarhet og generalitet kan det være ønskelig at rammeverket understøtter heterogen OID, det vil si at lokal OID kodes inn i global OID. For ANSAware kan dette gjøres for temporære objekter, men kostnaden kan være større enn vinninga, på grunn av grensesnittreferansers kompleksitet og størrelse. Rammeverket kan for øvrig skalere godt gitt at visse forutsetninger holder.
- Støtte for passivisering av temporære objekter mangler.

10.2.3 Grunnlag for videre arbeid

Vi kan konkludere med at det ut fra vårt rammeverk vil være grunnlag for eksperimenter og videre studier, med både ANSAware og med samvirke med applikasjoner mer generelt. Vi har utviklet et rammevert, men har bare i liten grad anvendt dette eksperimentelt for å se hvordan det oppfører seg i praksis. Vi vil trekke fram følgende punkter som aktuelle for videre arbeid:

- Objekt-orientert notasjon for spesifikasjon av forvaltning og permanente objekter. PermObj klassen (avsnitt 5.1.6) og notasjonen i avsnitt 6. 2 kan være et grunnlag. Her kan det være interessant å implementere en kompilator og eksperimentere med bruk av denne. Det kan være interessant å se pa generalisering av notasjon, det vil si om man kan gjøre (deler av) den uavhengig av ANSAware. Det er sjølsagt også rom for forbedringer av notasjonen.
- Effektiv forvaltning. Vi har vært inne på utnyttelse av avhengighetsforhold og preaktivisering. For å finne svar på om dette har noe for seg, er det behov for nærmere undesøkelser og eksperimentell etterprøving.
- Rammeverk som er mer generelle med hensyn på representasjon av lokal objektidentetet og generering av globale identifikatorer. Vi har diskutert muligheten for heterogene OID og bruk av lokale identifikatorer i globale OID. Dette bør undersøkes nærmere.
- Nærmere undersøkelser av feiltoleranse. Kan rammeverket understøtte dette?
- Effektiv og feiltolerant søppelsamling. Vi har realisert en enkel strategi for søppelsamling basert på referanse telling. Den har vist seg svært effektiv ved direkte binding til klient-programmer, men det mer åpent om effektiv og feiltolerant denne metoden er med mer distribuerte løsninger.
- Realisering av binding til FRIL. Vi har skissert hvordan dette kan gjøres og beskrevet og delvis implementert et rammeverk for binding av objekt-adaptere/ANSAware stubs til FRIL. Det vil være interessant å realisere dette og gjøre eksperimentelle undersøkelser hvordan dette fungerer sammen med en FRIL-tolk (se også [Sundsfjord94]).
- Vi har realisert en prototyp for C++ språkbinding til ANSAware. Objekt-orienterte språk har vist seg svært anvendbart til å representere konseptene i ANSAware og vårt rammeverkt. Vår enkle prototyp kan kompletteres og utvikles videre og vil kunne anvendes som ei mer generell ANSAware-språkbinding. Den kan erstatte helt bruk av innbakt PREPC kode.

Referanser

[Aho83] A. V. Aho, J. E. Hopcroft, *Data Structures and algorithms*, Addison Wesley 1983

[AtBu87] M.P. Atkinson, O.P. Buneman: *Types and Persistence in Database Programming Languages*, ACM Computing Surveys, Vol. 19. No. 2, June 1987.

[AP-APM] ANSAware 4.0 *Application Programmers Manual*

[AR.000.00] R. J. van der Linden, *An Overview of ANSA*, ANSA architecture report 000.00, APM 1993

[AR.001.01] R. T. O. Rees, *The ANSA Computational Model*, ANSA architecture report 001.01, APM 1993.

[Banc87] F. Bancilhon, T. Briggs, S. Khoshafi an, PValduriez, *FAD, a Powerful and Simple Database Language,* Proc. 13th VLDB conference, Brighton 1987.

[Beeri90] C. Beeri, *A formal approach to object-oriented databases*, Data & Knowledge Engineering, 5, 1990, pp. 353-382

[Bertino89] E. Bertino, M. Negri, G. Pelagatti, L. Sbatella, *Integration of Heterogenous Database applications through an object-oriented interface*, Information Systems, Vol. 14, No. 5, 1989.

[Blair91] G. Blair, J. Gallagher, D. Hutchison, D. Shepherd (eds), *Object-Oriented Languages, Systems and Applications*, Pitman 1991.

[Blair92] G. Blair, P. Dark, N. Davies, J. Mariani, C. Snape, *Integrated Support for Complex Objects in a Distributed Multimedia Design Environment*, Lancaster University, 1992.

[Black87] A. Black, N. Hutchison, E. Jul, H. Levy, L. Carter, *Distribution and Abstract Types in Emerald*, IEEE trans. on Software Engineering, Vol. SE-13, No. 1, January 1987.

[Bogle94] P. L. Bogle, A *Safe, Effi cient Object Database Interface Using Batched Futures*, TR-624 MIT LCS, Cambridge Ma., July 1994.

[Coul88] G. F. Couloris, J. Dollimore, *Distributed Systems, Concepts and Design*, Addison Wesley 1988

[Dahl92] O. J. Dahl, *Verifi able Programs*, Prentice Hall 1992

[Dale93] K. Dale, *En kvalitativ og kvantitativ vurdering av ANSAwares interaksjonsmekanismer*, Cand. Scient avhandling, UiTø, feb. 1993

[Dixon89] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, S. M. Wheater, *The Treatment of Persistent Objects in Arjuna, The Computer Journal,* Vol. 32, No. 4, 1989

[Elia89] F. Eliassen, FRIL - *Linguistic support for Interoperable Information System*, CS RR 89-02.

[Elia93a] F. Eliassen, *Aspects of Abstraction and Heterogenous Object Management in a Language Based Integration System*, Working Document 27. april 1993

[Elia93b] F. Eliassen, *On the Quality of Object Names in Heterogenous Autonomous Data Repositories*, Draft Report, Sep. 16, 1993

[ElNa89] R. Elmasri, S. B. Navathe, *Fundamentals Of Database Systems*, Benjamin/Cummings Publishing Company Inc. 1989

[ElKa91a] F. Eliassen, R. Karlsen, *Interoperability and Object Identity*, CS RR 91.12, October 1991

[ElKa91b] F. Eliassen, R. Karlsen, Providing Application Interoperability using functional programming concepts, Proc. EurOpen '91, F. Brazier (ed.)

[ElKa91c] F. Eliassen, R. Karlsen, *Interoperability using Functional and Object Oriented Programming Concepts: Problems and Solutions*, CS RR 91-13, October 1991

[ElVe88] F. Eliassen, J. Veijalainen, *A functional approach to information system interoperability, In Research into Network and Distributed Applications* (Proc. EUTECO '88), Speth, R. Ed. pp. 1121-1135.

[HaeDitt93] M. Härtig, K. R. Dittrich, *An Object-Oriented Integration Framework for Building Heterogenous Database Systems, Interoperable Database Systems* (DS-5) (A-25) 1993 IFIP

[HeLo85] D. Heimbigner, D. McLeod, *A Federated Architecture for information Management*, ACM Trans. on Office Information Systems, Vol. 3. july 1985, pp. 253-278

[HeSh90] J. Henshall, S. Shaw, *OSI EXPLAINED, end-to-end computer communication standards*, Second Edition, Ellis Horwood, 1990

[KhosCop86] S. N. Khoshafian, G. PCopeland, *Object Identity*, Proc. OOPSLA'86, ACM Sept. 1986, pp. 406-416

[KhVa90] S. Khoshafian, PValduriez, *Sharing, persistence and object-orientation: A database perspective*, In: Advances in Database Programming Languages, F. Bancilhon & P. Buneman (eds), ACM Press

[Lecluse88] C. Lecluse, P. Richard, F. Velez, *O2, an object-oriented data model*, Proc. ACM SIGMOD Conference, Chicago 1988

[LEDA3.0] *Leda User Manual, version 3.0*, S. Naher, Max-Planck-Institut fur Informatik, Stadtwald

[Liskov93] B. Liskov, M. Day, L. Shrira, Distributed Object Management in Thor, In: Distributed Object Management, T. Ozsu et. al. (eds), Morgan Kaufmann 1993

[LitAb86] W. Litwin, A. Abdellatif, *Multidatabase interoperability*, Computer 19, Dec. 1986, pp. 10-18

[MooVer92] H. Morris, P. Verbaeten, *Persistency Support for Mobile Objects in the COMET Heterogenous Environment*, In: Proc. 2nd. International Workshop on Object-Orientation in Operating systems, Sept. 1992

[Netm91] Network Monitor, *ANSA A Model for Distributed Computing*, Vol. 6, No. 11, Nov. 1991

[Nicol93] J. R. Nicol, C. T. Wilkes, F. A. Manola, *Object Orientation in Heterogenous Distributed Computing Systems*, Computer, june 1993, pp. 57-67

[ODS.Stud92] ODS gruppen, IIS prosjektet, *Generisk studentprosjekt beskrivelse*, F. Eliassen 17.01.92

[Olsen92] M. H. Olsen, *A Persistent Object Infrastructure for Heterogenous Distributed Systems*, In: Proc. 2nd. International Workshop on Object-Orientation in Operating systems, Sept. 1992

[OMG.90.9.1] *Object Management Architecture Guide*, OMG TC Document 90.9.1, v. 1.0, R. M. Soley (ed), November 1991

[OMG.92.12.1] *The Common Object Request Broker: Architecture and Specifi cation*, OMG Document number 91.12.1, Revision 1.1, Draft, 10. Dec. 1991

[OMG.93.9.2] HyperDesk corp., *Revised Submission in Response to the OMG RFP for a C++ Language Mapping*. OMG TC Document 93-9-2, Sept. 1993

[OStore2.0] *User Guide, ObjectStore, Release, 2.0 For UNIX Systems*, Object Design inc. DU 1002-100, October 1992

[Renesse89] R. V. Renesse, *The Functional Processing Model*, Dr. grads avhandling, Vrije Universiteit, Amsterdam 1989

[RM-ODP.1] ISO/IEC JTC/SC21/WG7, *ITU-T X.901 ISO/IEC 10746.1, ODP Reference Modeo Part 1*, N. 885 Working document, November 1993

[RM-ODP.3] ISO/IEC JTC/SC21/WG7, *Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model* , ISO/IEC DIS 10746-3.1

[Rose91] M. T. Rose, The Open Book, Prentice Hall 1991

[Rumbaugh91] J. Rumbaugh, M. Blaha,W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall 1991

[ShLa90] Sheth. A, Larson, J, *Federated Database Systems for managing Heterogenous and Autonomous Databases*, ACM Computing Surveys, Vol. 22, No. 3, September 1990

[StaGif90] J. W. Stamos, D. K. Gifford, *Remote Evaluation*, ACM Trans. on Programming Languages and Systems, Vol. 12, No. 4, October 1990, pp. 537-565

[Stroustrup91] B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison-Wesley, 1991

[TR 38.00] *Architecture and Design Frameworks*, ANSA Technical report TR,038,00 February 1993

[Watt90] D. A. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall 1990

[Wirf90a] R. J. Wirfs-Brock, R. E. Johnson, Surveying Current Research in Object-Oriented Design, CACM, Vol. 33, No. 9, September 1990

[Wirf90b] R. J. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990

Vedlegg 1:
Kildekode for objekt-adapter bibliotek

- Proxy.h
- Proxy.C
- AnsaProxy.h
- AnsaProxy.C
- CapsulePerm.h
- CapsulePerm.C
- GA_mgmt.h
- GA_mgmt.dpl

Vedlegg 2:
ANSAware og C++

- HVORDAN IMPLMENTERE EN ANSAWARE TJENER I C++, Notat til D-341 studenter, høst 1994. (gjelder også implementasjon av klienter)
- ansa++.h
- CC_body.C

Vedlegg 3:
Språkbinding for C++

- Endringer til stubc (kort notat)
- outsubs.c (back end for kompilator)
- IFC_Ansa.h (superklasse for objektreferanser)
- seqlist.h (generisk liste klasse for SEQUENCE)

Vedlegg 4:
Språkbinding for FRIL

- ObjectAdapter.idl
- Operation.h
- Operation.C
- AnsaOperation.h
- AnsaOperation.C
- ArgList.h
- ArgList.C
- test.C
- map.C

Vedlegg 5:
SBank applikasjonen

- SBank.idl
- Account.idl
- SBTypes.idl
- client.C
- BankPerm.h
- BankPerm.C
- IFC_SBank.h
- IFC_Account.h
- IFC_SBTypes.h
- FC_SBank.C
- IFC_Account.C
- IFC_SBTypes.C

Vedlegg 6:
Clone applikasjonen

- Clone.idl
- ClonePerm.C
- IFC_CLone.h
- IFC_Clone.C

Vedlegg 7:
Program for ytelses tester

- test.C (med objektadapter og C++ språkbinding)
- test.dpl (PREPC program uten objektadapter)