# Dynamisk konfigurering av komponentbaserte tjenester

Hovedoppgave
ved
sivilingeniørutdanningen i
Informasjons- og Kommunikasjonsteknologi

av
Morten Thorsen og Arnfinn Andersen

Grimstad, Juni 2001

# Sammendrag

Utbredelsen av komponentbaserte tjenester har i den senere tid økt. Selv om komponent-teknologien i seg selv bidrar til en viss dynamikk, er allikevel komponentene/tjenestene ofte nokså statiske.

Rapporten belyser ulike komponent-teknologier og muligheter for å gjøre komponenter og komponentbaserte tjenester konfigurerbare. Ideen er å prøve å skape mer dynamikk og gjenbruk i tjenestene, ved at man kan velge ut, tilpasse og bruke hver enkelt komponent/tjeneste dynamisk ut fra klientens type, behov, bruksmønster og kjøretidsomgivelser.

Av komponent-teknologier dekkes CORBA Component Model, Sun JavaBeans/Enterprise JavaBeans og Microsoft COM+, og når det gjelder mekanismer for dynamisk konfigurering har rapporten hovedtyngde på bruk av script. På grunn av stor utbredelse av komponent-teknologi i nettbaserte tjenester, og økt behov for dynamikk i dette miljøet, har vi valgt å fokusere mye av oppgavens problemstillinger mot web-teknologi.

Fremgangsmåten i prosjektet har bestått i å først kartlegge og beskrive teori for de valgte komponentstandardene og metodene for dynamisk konfigurering, for så å sette sammen og teste ulike kombinasjoner av elementer fra det teoretiske arbeidet i praksis.

I vårt arbeid har vi kommet frem til både generelle og praktiske (konkrete) løsninger for realisering av dynamisk konfigurering med hensyn på komponentbaserte tjenester. Vi mener at det å implementere dynamisk konfigurering (scripting) direkte i komponentene, hvis komponentstandardene tilbød dette, ville være det mest hensiktsmessige med hensyn på effektivitet. Siden dette ikke er tilfelle i dagens standarder, mener vi at det å la en webtjener (med tilknyttet tolk) fronte og utføre den dynamiske konfigureringen (scriptene), vil være den mest anvendelige løsningen. Vi mener at dynamisk konfigurering av komponentbaserte tjenester ved hjelp av script, spesielt i web-sammenheng, kan anbefales.

# Forord

Denne rapporten er et resultat av en 10-vekttalls hovedoppgave gitt ved sivilingeniørstudiet i Informasjons- og Kommunikasjonsteknologi (IKT) ved Høgskolen i Agder (HiA), avdeling Grimstad, våren 2001. Oppgaven er utformet av studentene Morten Thorsen og Arnfinn Andersen, i samarbeid med faglærerne Jan P. Nytun, Mikael Snaprud og Øyvind Hanssen.

Utførelsen av arbeidet har funnet sted ved Høgskolen i Agder, avdeling Grimstad, hvor vi som studenter både har jobbet individuelt og i tett samarbeid.

Vi vil takke våre veiledere Jan P. Nytun, Mikael Snaprud og Øyvind Hanssen for god støtte og oppfølging gjennom prosjektet.

Morten Thorsen                Arnfinn Andersen

Grimstad, juni 2001

# Innhold

**SAMMENDRAG** ..... **2**
**FORORD** ..... **3**
**INNHOLD** ..... **4**
**1 INNLEDNING** ..... **6**
1.1 BAKGRUNN ..... 6
1.2 PROBLEMSTILLING OG OPPGAVEBESKRIVELSE ..... 6
1.3 MÅL ..... 7
1.4 AVGRENSNINGER ..... 8
1.5 TILNÆRMING TIL PROBLEMET ..... 9
1.6 PROSJEKTGRUPPEN ..... 10
1.7 ARBEIDSGANGEN I PROSJEKTET ..... 10
1.8 RAPPORTENS STRUKTUR ..... 11
**2 METODIKK** ..... **12**
2.1 LITTERATURSTUDIE ..... 12
2.2 WEBSIDER OG NYHETSGRUPPER ..... 12
2.3 PRAKTISK ARBEID ..... 12
2.4 VEILEDNING ..... 12
**3 KOMPONENTBASERTE TJENESTER** ..... **13**
3.1 INNLEDNING ..... 13
3.2 ARKITEKTUR ..... 14
3.3 NETTJENESTER ..... 15
3.4 KOMPONENTTEKNOLOGI ..... 16
*3.4.1 Innledning* ..... *16*
*3.4.2 Objekt* ..... *17*
*3.4.3 Komponent* ..... *18*
*3.4.4 Grensesnitt* ..... *19*
*3.4.5 Rammeverk* ..... *20*
3.5 KOMPONENTSTANDARDER ..... 22
*3.5.1 Innledning* ..... *22*
*3.5.2 OMG CORBA/CCM* ..... *24*
*3.5.3 Sun JavaBeans/Enterprise JavaBeans* ..... *32*
*3.5.4 Microsoft COM/DCOM/COM+* ..... *39*
**4 DYNAMISK KONFIGURERING** ..... **45**
4.1 INNLEDNING ..... 45
4.2 DYNAMISK KONFIGURERING AV KOMPONENTER/TJENESTER (VÅR IDÉ) ..... 47
4.3 INFORMASJONSINNSAMLING (KLIENT) ..... 50
*4.3.1 Innledning* ..... *50*
*4.3.2 Brukerprofil* ..... *50*
*4.3.3 HTML Forms* ..... *50*
*4.3.4 JavaScript* ..... *52*
*4.3.5 Cookies* ..... *53*

*4.3.6 Miljø-variabler (environment variables)* ... *54*
4.4 KONFIGURERING (TJENER) ... 55
*4.4.1 Innledning* ... *55*
*4.4.2 Common Gateway Interface (CGI)* ... *55*
*4.4.3 Server Pages* ... *56*
*4.4.4 Scriptspråk* ... *59*
**5 PRAKTISKE LØSNINGER (EKSPERIMENTELL DEL) ... 63**
5.1 INNLEDNING ... 63
5.2 AVGRENSNINGER ... 63
5.3 SYSTEMBESKRIVELSE OG KONFIGURERING (TJENER) ... 64
*5.3.1 Microsoft Internet Information Services 5.0* ... *64*
*5.3.2 Perl 5.7.1 med JPL-modul* ... *65*
*5.3.3 Jython 2.0* ... *65*
*5.3.4 Java 2 SDK, Standard Edition 1.3.0_02* ... *66*
*5.3.5 JRun Server 3.0, Developer Edition* ... *66*
*5.3.6 Python Server Pages 1.0.3* ... *67*
*5.3.7 Perl Dev Kit 2.1 (PerlCOM)* ... *67*
5.4 SCENARIOS ... 68
*5.4.1 IIS, CGI, Perl, JPL, JavaBeans* ... *68*
*5.4.2 IIS, CGI, Jython, JavaBeans* ... *69*
*5.4.3 IIS, JRun, PSP, JavaBeans* ... *70*
*5.4.4 IIS, JRun, JSP, JavaBeans* ... *70*
*5.4.5 IIS, JRun, Servlets, JavaBeans* ... *71*
*5.4.6 IIS, ASP, COM* ... *72*
*5.4.7 IIS, ASP, PerlCOM, COM* ... *74*
**6 DRØFTING ... 76**
6.1 INNLEDNING ... 76
6.2 GENERELLE LØSNINGSMETODER ... 77
6.3 PRAKTISKE LØSNINGER (SCENARIOS) ... 80
6.4 ANVENDELSE ... 81
**7 KONKLUSJON ... 84**
**REFERANSER ... 85**
**APPENDIX ... 88**
APPENDIX A: OPPRINNELIG OPPGAVEBESKRIVELSE ... 88
APPENDIX B: TEKNOLOGIER I SUN J2EE ... 89
APPENDIX C: TEKNOLOGIER I MICROSOFT WINDOWS DNA ... 89

# 1 Innledning

## 1.1 Bakgrunn

Oppgaven "Dynamisk konfigurering av komponentbaserte tjenester" ble våren 2001 formulert som en hovedoppgave ved sivilingeniørstudiet i Informasjons- og Kommunikasjonsteknologi (IKT) ved Høgskolen i Agder (HiA), avdeling Grimstad. Oppgaven er utformet av studentene Morten Thorsen og Arnfinn Andersen, i samarbeid med faglærerne Jan P. Nytun, Mikael Snaprud og Øyvind Hanssen.

## 1.2 Problemstilling og Oppgavebeskrivelse

Mange av dagens klient-tjener systemer er basert på en eller annen form for komponent-teknologi. Selv om dette i seg selv bidrar til økt gjenbruk, er tjenestene allikevel ofte basert på forholdsvis statiske modeller, dvs at tjenestene/komponentene er nøye tilpasset en bestemt klients type, behov og bruksmønster (løser en oppgave). Oppgaven skal belyse ulike komponent-teknologier og muligheter for å gjøre komponenter og komponentbaserte tjenester konfigurerbare. Ideen med dette vil være å prøve å skape en mer dynamisk modell, ved at man kan velge ut, tilpasse og bruke hver enkelt komponent/tjeneste dynamisk ut fra klientens type, behov, bruksmønster og kjøretidsomgivelser. Ved å la komponentene/tjenestene være dynamisk konfigurerbare, vil det sannsynligvis være enklere å tilby tjenester som kan skreddersys for hver enkelt klient.

Oppgavens eksperimentelle del skal ha hovedfokus på web-teknologi, med browser som klient og webserver som tjener. Det skal undersøkes om man kan sette sammen og konfigurere en komponentbasert tjeneste dynamisk ved hjelp av script (f.eks. Perl, Python, PHP). Valg av komponenter (f.eks. JavaBeans) og tilpasning av disse skal gjøres ved hjelp av script på tjeneren. Det skal også belyses ulike metoder på klienten for å beskrive behov, bruksmønster og brukerprofil, gjerne ved bruk av cookies, parametre og ulike klientscript.

Det er også ønskelig at det gis eksempler på type funksjonalitet som kan være hensiktsmessig å realisere som konfigurerbare tjenester/komponenter.

**Kommentar:**

Oppgavebeskrivelsen for dette prosjektet ble som nevnt ovenfor utformet av oss selv. Vi har ikke hatt noen direkte oppdragsgivere å forholde oss til, så oppgaven ble i starten utformet mest basert på egne ideer. I samarbeid med våre veiledere, kom vi imidlertid frem til at den opprinnelige oppgavebeskrivelsen (Appendix A) i utgangspunktet var noe for vid og uklar, og derfor trengtes å konkretiseres og avgrenses mer. I den gjeldende oppgavebeskrivelsen (ovenfor) har vi dessuten valgt å ha et høyere fokus på teori og flere mulige løsningsscenarios, fremfor prototypeimplementasjon av en spesifikk tjeneste, som var utgangspunktet til og begynne med.

## 1.3 Mål

Målet med oppgaven vil overordnet være å kartlegge hvilke metoder/teknologier som kan brukes for å implementere dynamisk konfigurering av komponentbaserte tjenester, vurdere disse opp imot hverandre, og vise praktiske eksempler på hvordan systemer basert på dynamisk konfigurering kan modelleres og utformes.

**Delmål:**

- Belyse ulike komponent-teknologier med hensyn på komponentbaserte tjenester.
- Beskrive generelle metoder for å gjøre komponenter og komponentbaserte tjenester konfigurerbare, og evaluere disse opp mot hverandre.
- Belyse ulike metoder på klienten for å beskrive behov, bruksmønster og brukerprofil.
- Beskrive teknologier (mekanismer) for dynamisk konfigurering.
- Beskrive praktiske løsninger (scenarios) for oppsett og realisering av dynamisk konfigurering med hensyn på komponentbaserte tjenester, og evaluere disse opp mot hverandre.
- Identifisere hvilken type funksjonalitet som er hensiktsmessig å plassere i separate konfigurerbare komponenter (anvendelse).

## 1.4 Avgrensninger

Av komponentstandarder, har vi valgt å ta for oss CORBA Component Model, Sun JavaBeans/Enterprise JavaBeans og Microsoft COM, siden vi mener at disse er de mest utbredte og at de er tilstrekkelige for å belyse våre problemstillinger. I de praktiske løsningene har vi derimot kun sett på Sun JavaBeans og Microsoft COM, hvilket har med kompleksitet og tilgjengelighet å gjøre.

Når det gjelder mekanismer for dynamisk konfigurering har vi valgt å ta utgangspunkt i bruk av scriptspråk. Eventuelle andre former for konfigurering er ikke dekket. Videre har vi ikke direkte tatt hensyn til om konfigureringen skal skje på sesjonsbasis eller pr. invokering. Vi har også utelatt vanlig kode-logikk (if-then-else) som trengs for å realisere konfigurering, da dette vil variere kraftig fra tjeneste til tjeneste.

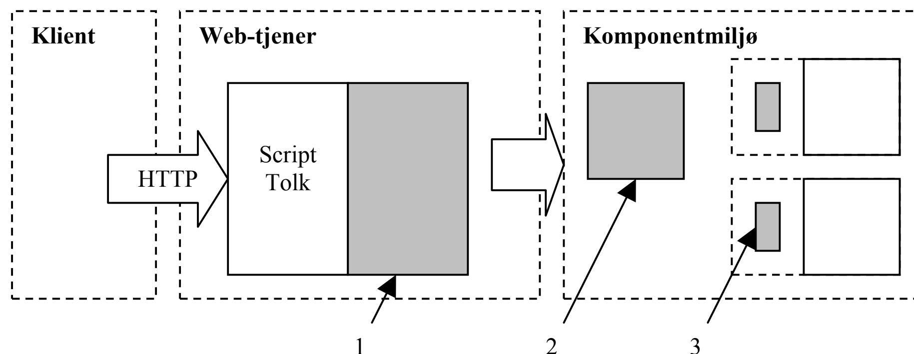
I de praktiske løsningene har vi valgt å ha fokus på flere mulige løsningsscenarios, fremfor prototypeimplementasjon av en spesifikk tjeneste og teknologi. Dette innebærer derfor at løsningene på ingen måte beskriver fullverdige tjenester, men viser prinsipper for oppsett og virkemåte.

En generell avgrensning vi har gjort, er å sette mye av oppgaven i web-sammenheng. Dette kan eventuelt medføre at enkelte deler ikke er like relevante for annen tjenesteutvikling. Grunnen til denne avgrensningen er at vi mener at dynamisk konfigurering er mest aktuelt og nødvendig i dette miljøet, der det eksisterer mange ulike typer klienter og man har ulike behov til ulike tider.

## 1.5 Tilnærming til problemet

Som oppgavebeskrivelsen forespeiler, skal det undersøkes om den dynamiske konfigureringen kan realiseres ved hjelp av scripts. Med dette som utgangspunkt, ser vi for oss at det mulig å implementere slik komponent-/tjenestekonfigurering på tre prinsipielle måter:

1. Plassere scriptene hos en webtjener, og la webtjeneren styre konfigureringen.
2. Lage egen komponent for scripting (som kan integreres i komponentmiljøet).
3. Utvide komponentstandarden(e) slik at komponentene kan inneholde egne script.




For å teste ut dynamisk konfigurering i praksis, mener vi at det er enklest, og tilstrekkelig for oppgaven, å realisere dette i form av måte 1. Vi har derfor valgt å basere våre eksperimentelle forsøk på denne løsningen.

## 1.6 Prosjektgruppen

Prosjektgruppen har bestått av studentene Morten Thorsen og Arnfinn Andersen ved sivilingeniørstudiet innen Informasjons- og Kommunikasjonsteknologi (IKT) ved Høgskolen i Agder, avdeling Grimstad. Prosjektarbeidet har bestått av både fellesarbeid og individuelt arbeid, og dette har vært fordelt mest mulig jevnt på begge deltakerne.

## 1.7 Arbeidsgangen i prosjektet

De to første månedene av prosjekttiden ble først og fremst brukt til litteraturstudie. Hensikten med dette var å tilegne oss tilstrekkelig informasjon om komponentbaserte tjenester, scriptspråk og web-teknologi til å kunne starte på vårt konkrete arbeid. Deretter startet vi med å definere hvilke elementer vi ønsket å ha med i rapporten, og laget en struktur for denne. Etter dette startet vi med å forfatte teori angående de valgte komponentstandardene, samt beskrive ulike elementer vi mente måtte inngå i dynamisk konfigurering (metoder, språk osv). Parallelt kom vi også frem til noen generelle løsninger for hvordan vi mente dynamisk konfigurering kunne implementeres. Etterhvert som de teoretiske delene om komponentstandardene og dynamisk konfigurering kom på plass, begynte vi å sette sammen og teste elementer fra de teoretiske kapitlene, for å se om kombinasjonene lot seg gjennomføre i praksis. Ut i fra dette arbeidet beskrev vi så de ulike testoppsettene (scenarioene) med hensyn på teknologi og virkemåte. Den siste tiden av prosjektperioden ble brukt til å evaluere/drøfte de ulike teoretiske og praktiske løsningene, samt skrive sammendrag og finpusse de allerede eksisterende delene av rapporten.

## 1.8 Rapportens struktur

**Kapittel 2 (Metodikk)** består av en kort beskrivelse av fremgangsmåte (metodikk) anvendt under gjennomføringen av oppgaven.

**Kapittel 3 (Komponentbaserte tjenester)** inneholder teknologibeskrivelser av dagens mest brukte komponentstandarder, og definerer sentrale begreper innenfor utvikling av komponentbaserte tjenester. Komponentstandardene beskrives, og begreper som arkitektur, objekt, komponent, grensesnitt og tjeneste defineres.

**Kapittel 4 (Dynamisk konfigurering)** beskriver våre ideer bak dynamisk konfigurering av komponenter og tjenester, samt hvilke løsningsmetoder og mekanismer vi mener er anvendelige for dette formålet. Vi presenterer teknologier som CGI, Server Pages og scriptspråk

**Kapittel 5 (Praktiske løsninger)** beskriver ulike praktiske løsninger (scenarios) for dynamisk konfigurering av komponentbaserte tjenester. Kapittelet gir et overblikk over anvendt programvare og oppsett av dette, og forklarer metoder for datainnsamling og objektaksessering i de ulike scenarioene.

**Kapittel 6 (Drøfting)** definerer drøftingskriterier, og gjør en drøfting av generelle løsninger, praktiske løsninger (scenarios) og anvendelse av dynamisk konfigurering med hensyn på komponentbaserte tjenester.

**Kapittel 7 (Konklusjon)** oppsummerer erfaringene vi har gjort gjennom prosjektet, og trekker konklusjoner.

# 2 Metodikk

## 2.1 Litteraturstudie

For å tilegne oss tilstrekkelig informasjon om komponentbaserte tjenester til å kunne starte på vårt konkrete arbeid, ble de to første månedene brukt til litteraturstudie. I litteraturstudie-perioden ble det også foretatt en utvelgelse av ytterligere litteratur som vi så for oss å kunne gjøre bruk av senere i prosjekt-perioden. Litteraturstudie har i så måte vært en kontinuerlig del av prosjektperioden.

## 2.2 Websider og Nyhetsgrupper

Internett har nok vært vårt største hjelpemiddel i dette prosjektet, og både websider og nyhetsgrupper har vært flittig brukt. Problemet med disse mediene er at det kan være vanskelig å sortere ut den informasjonen man virkelig er interessert i (man finner for mye), og det kan også til tider være vanskelig å verifisere informasjonen. Allikevel mener vi at anvendte kilder er tilstrekkelig kvalitetssikret.

## 2.3 Praktisk arbeid

Det er alltid ønskelig å kunne bevise påstander og teoretiske antagelser, vi så det derfor som en fordel å sette opp og prøve ut forskjellige løsninger vi mente kunne være egnet for dynamisk konfigurering av komponentbaserte tjenester. Dette arbeidet foregikk først og fremst i april og mai måned (etter litteraturstudie-perioden), selv om dette er noe vi har tenkt på helt fra starten av.

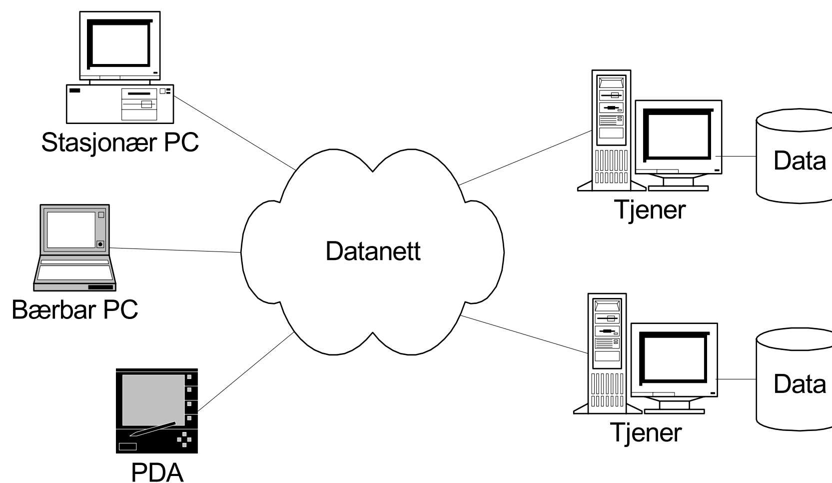
## 2.4 Veiledning

Gjennom prosjekt-perioden har vi hatt en fin dialog med våre tre veiledere: Jan P. Nytun, Mikael Snaprud og Øyvind Hanssen. De har hjulpet og støttet oss ved behov, både med organisatoriske og faglige elementer.

# 3 Komponentbaserte tjenester

## 3.1 Innledning

Dagens datasystemer består ofte av flere datamaskiner koblet sammen i en eller annen form for nettverk. Nettverkene kan være åpne og fritt tilgjengelige (Internett), lukkede bedriftsnett (intranett), eller en blanding av begge typer (ekstranett). Brukerene har tilgang til datasystemet via en lokal klient applikasjon (brukergrensesnitt). Tjenesten som brukeren ønsker at datasystemet skal utføre, er ofte distribuert over flere tjenermaskiner, og brukeren ønsker ofte å aksessere denne tjenesten via forskjellige typer klienter. Se figur under:

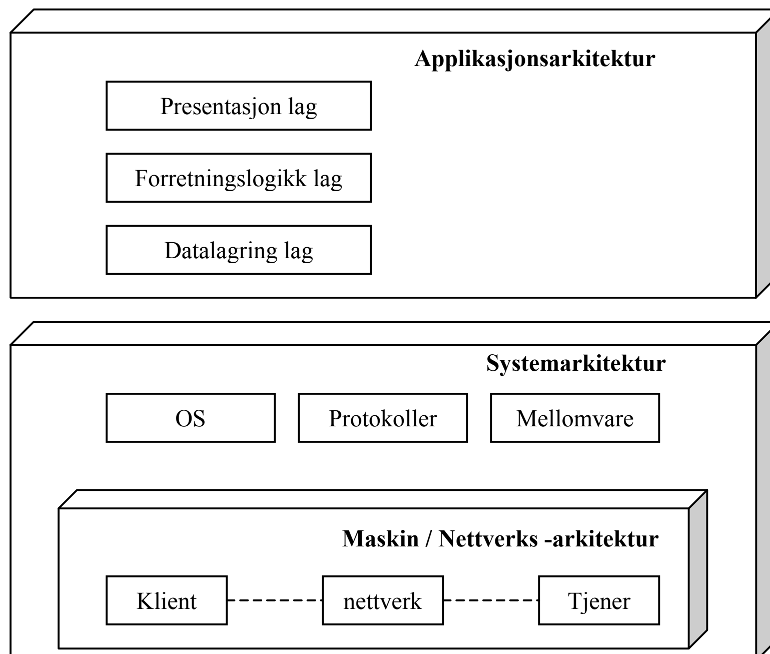



**Figur 1, Datasystem bestående av sammenkoblede datamaskiner**

For å forklare hvordan datasystemene er sammenkoblet, samt hvordan tjenestene som tilbys er organisert/distribuert, snakker man ofte om arkitekturen til datasystemene/tjenestene. Det er derfor naturlig å begynne å se på begrepsapparatet som brukes for å beskrive datasystemenes egenskaper, slik som nettverksarkitektur, systemarkitektur og applikasjonsarkitektur.

## 3.2 Arkitektur

Når vi beskriver et datasystems egenskaper bruker vi ofte begrepet arkitektur. I dagens datasystemer snakker en ofte om tre typer arkitektur: maskin/nettverksarkitektur, systemarkitektur og applikasjonsarkitektur. (se figur under)




**Figur 2, Datasystem og Arkitektur**

Med maskin/nettverksarkitektur menes hvilke datamaskiner (klienter eller tjenere) som datasystemet består av, samt hvordan disse er sammenkoblet i et nettverk. Med systemarkitektur mener vi som oftest underliggende maskin/nettverksarkitektur, samt operativsystem, protokoller og mellomvare. Applikasjonsarkitekturen beskriver ”plattform” og ”rammeverk” som muliggjør applikasjonsutvikling. Dette kan f. eks være web-tjener software (og utviklingsverktøy som disse tilbyr), komponentteknologi, eller spesifikke applikasjoner og utviklingsplattformer.

Utviklingen til dagens nettbaserte system- og applikasjonsarkitektur har foregått over flere faser gjennom det siste tiåret. Drivkraften til utviklingen har vært populariteten og utbredelsen til netteknologi, og utviklingen har vært en gradvis prosess over mange år:

**Første fase:** Denne kom på starten av nittitallet da klient/tjener arkitekturen i stor grad erstattet en-maskin arkitekturen, som til da var den mest vanlige formen for dataprosessering. Utbredelsen av lokalnett gjorde at klient og tjener kunne adskilles på forskjellige maskiner. Typisk anvendelse av klient/tjener arkitekturen var å la klienten selv gjøre ”enkle” prosesseringsjobber, mens dersom ”tunge” prosesseringsjobber skulle utføres, måtte klienten gjøre en forespørsel mot en tjener som var mere tilpasset denne type oppgave. Tjenerne var ofte utstyrt med mere prosesseringsresurser, men utførte allikevel ofte svært spesifikke tjenester.

**Andre fase:** Denne kom på midten av nittitallet da flerlagsarkitekturen (*N-tier*) kom for fullt. Dette kom av populariteten og utbredelsen av Internett og intranett. Istedenfor at klientene hadde direkte tilgang til tjeneren, ble det mere vanlig å la klientene få aksess via Internett eller intranettet, til en webtjener eller applikasjonstjener. Disse tjenerne har ofte videre aksess til andre tjenere som f.eks. databasetjenere, filtjenere ol. via lokalnett. På denne måten ble webtjenerne og applikasjonstjenerne mere generelle og kunne tilby et bredere utvalg av tjenester. De fleste av dagens datasystemer kan sies å være i denne fasen.

**Tredje fase:** Vi er nå på vei inn i tredje fase som sies å være tjenesteorientert. Med tjenesteorientert menes at underliggende arkitektur eller plattform har mindre betydning, bare tjenesten kan aksesseres og brukes av klientene. I tillegg er det ofte ønskelig å aksessere de samme tjenestene fra forskjellige lokasjoner, og med forskjellige typer klienter. Microsoft .NET og Sun Open Net Environment (Sun ONE) initiativene er eksempler på denne type arkitektur.

## 3.3 Nettjenester

Gjennom utbredelsen av netteknologi har man den siste tiden sett at brukerens fokus har forandret seg fra å være applikasjonsbasert til å bli tjenestebasert. Brukeren er ikke lenger så opptatt av hvilke applikasjon som skal brukes for å utføre en tjeneste, men ønsker istedenfor at tjenesten kan brukes når og hvor en selv ønsker det, og gjerne med forskjellige typer klienter.
Lese nyheter, lese/skrive e-post, betale regninger, fjernjobbing, og aksessere bedriftens datanett  er alle eksempler på slike tjenester.

Nye og distribuerte tjenester, samt brukerenes ønske om at flere og flere av de tradisjonelle applikasjonene skal være tilgjengelig som nettbaserte tjenester, har medført at størrelsen og kompleksiteten til datasystemene som tilbyr disse tjenestene har økt kraftig. For å prøve å bedre dette problemet har system- og applikasjonsarkitekturen også utviklet seg i takt med datasystemet kompleksitet. Spesielt har utviklingene innen det som kalles ”mellomvare” (*midleware*) utviklet seg mye det siste tiåret. Den første mellomvarestandarden for fjernkall av funksjoner, het *Remote Procedure Call (RPC)*. Etter hvert som ønsket om sammenkobling og integrering av datasystemene økte, viste dette ”rammeverket” seg imidlertid å være litt for statisk. Ofte måtte systemene gjennom relativt store redesign prosesser, selv for relativt små og enkle utvidelser. For å løse dette problemet utviklet OMG et nytt ”rammeverk” for distribuerte objekter, kalt *Common Object Request Broker Architecture (CORBA)*. Dette ”rammeverket” var helt fra starten av spesielt godt tilpasset distribuerte miljøer, og muliggjorde utvikling og integrering i forskjellige språk, og på forskjellige operativsystem. Dette var et stort steg i utviklingen mot enklere integrering og standardiserte utviklingsprosesser. Etter hvert har man imidlertid sett at det å bruke distribuerte objekter ikke alltid er det ideelle, men at binære (kompilerte) komponenter (byggeklosser) ofte er bedre. Bruk av komponenter og komponentteknologi gir ofte mange fordeler fremfor tradisjonelle objekter, spesielt for gjenbruk av standard løsninger, samt utvidelser av eksisterende systemer. System- og applikasjonsarkitekturen har derfor utviklet seg til nå å mere være tjeneste- og komponentbasert.

## 3.4 Komponentteknologi

### 3.4.1 Innledning

Komponentbasert tjenesteutvikling har som grunn-ide, at ved å bryte store og komplekse problemer ned i mindre små enheter, også kalt komponenter, kan en redusere kompleksiteten til problemet. Slike komponenter kan også være løsninger på mere generelle problemstillinger som forekommer ofte i utviklingsprosjekter. Dermed kan disse løses en gang for alle, for så og bli gjenbrukt i mange utviklingsprosjekter. På denne måten vil utviklingstiden og dermed kostnadene reduseres. Stabiliteten til produktet vil også øke siden mye brukte komponenter vil være bedre testet, og derfor inneholde færre feil. En annen meget stor fordel er at ferdige

komponenter kan kjøpes fra komponentleverandører, noe som kan redusere utviklingstiden betydelig. De mange fordeler som bruk av komponentteknologi gir kan oppsummeres med følgende punkter:

- Redusere kompleksiteten ved å bryte store problemer ned i mindre små.
- Redusere utviklingstiden, og dermed kostnadene, ved å kjøpe ferdige komponenter.
- Enklere å håndtere forandring og utvidelse av systemet
- Økt kvalitet ved gjenbruk av vel prøvde og vel testede løsninger.
- Forenkler utvikling i parallelle og distribuerte prosesser.
- Reduserte vedlikeholdskostnader ved at komponenter enkelt kan byttes eller utvides.
- Muliggjør gjenbruk.

Siden komponentteknologi gir mange fordeler må vi se litt nærmere på hva en komponent er, og hvordan disse brukes til tjenesteutvikling. Men siden komponenter som oftest bygges opp av objekter, og siden komponenter og objekter har mange likhetstrekk, må vi først se litt på egenskapene til objekter og objektorientert utvikling.

### 3.4.2 Objekt

Forskjellen mellom objekt og komponent kan i mange sammenhenger virke ”diffus”. Dette kommer av at begrepene ofte blandes om hverandre i litteraturen. Noen kaller f. eks. distribuerte objekt miljøer, for komponentstandarder. Vi mener imidlertid at objekt og komponent er to forskjellige ting, siden en komponent er ”knyttet” opp mot et kjøretidsmiljø (*runtime environment*) og beskriver en binær (ferdig) software ”byggekloss”, mens objekter som oftest har en ”knytning” mot objektorientert programvareutvikling. Når en snakker om objektorientert programvareutvikling kan ordet objekt som oftest likestilles med en instans av en klasse. En klassen kan betraktes som en ”mal” for objektene som skal opprettes, og har derfor en sterk ”knytning” mot det objektorienterte utviklingsspråket som brukes. Noen av objektets egenskaper er innkapslet implementasjon, tilstandsinformasjon, samt unikt identitet. Andre sentrale begreper innen objektorientert utvikling er grensesnitt, arv og polymorfi.

### 3.4.3 Komponent

Komponenter kan betraktes som programvare-”byggeklosser”. Større systemer bygges ved å sette sammen mindre deler. Hver komponent tilbyr et sett av metoder som kan aksesseres via sine interface[1],[2]. *Unified Modeling Language (UML)* [6] definerer begrepet komponent, og tilbyr terminologi for hvordan en kan beskrive og modellere med komponenter:

> *Component: ”A physical and replaceable part of a system that confirms to and provides the realization of a set of interfaces.”*

Komponentteknologi bruker mange av de samme begreper som objektorientert utvikling, men selv om det er mange likheter mellom komponenter og klasser, er det også noen veldig essensielle forskjeller, som er belyst under:

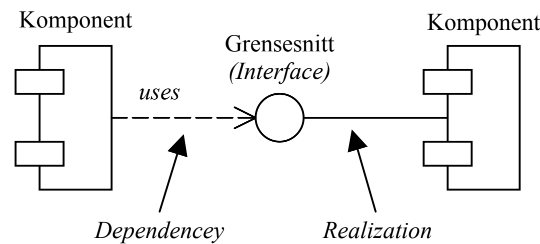
<u>Likheter:</u> Begge har navn, begge kan realisere (*realize*) et sett av grensesnitt (*interface*), begge kan delta i avhengighet- (*dependency*), generalisering- (*generalization*) og assosiasjon- (*association*) relasjoner. Begge kan være nøstete , begge kan ha instanser og begge kan være delaktig i interaksjoner.

<u>Ulikheter:</u> Klasser representerer logiske abstraksjoner, mens komponenter representerer fysiske ting (i et *runtime environemet*). Komponentene representerer den fysiske innpakning til de ellers logiske delene, og må derfor betraktes på et helt annet abstraksjonsnivå. Klasser kan ha attributter og operasjoner direkte, mens komponentene har operasjoner som bare kan aksesseres via sine i tilhørende grensesnitt.

Oppsummert kan en si at komponentteknologien bruker de samme fundamentale egenskaper som objekter: Atskillelse av dataelementer og funksjonene som prosesserer på dem, innkapsling (*encapsulation*) som oppnås ved at klientene kun har avhengighet med spesifikasjonen og ikke med implementasjonen, samt unik identitet som er uavhengig av tilstand. Både objekt og komponent bruker spesifikasjons beskrivelsen til å beskrive spesifikasjons avhengighet, kalt grensesnittsbeskrivelse eller bare grensesnitt (*interface*).

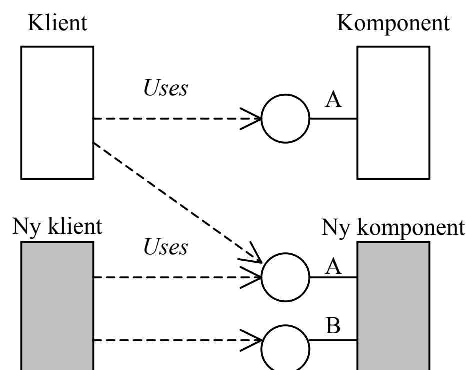
## 3.4.4 Grensesnitt

Grensesnitt (*Interface*) defineres som en samling av operasjoner som brukes for å spesifisere tjenesten som tilbys av en komponent eller klasse. Grensesnittet som en komponent realiserer blir kalt et eksportert grensesnitt (*export interface*), noe som betyr at dette er et grensesnitt som komponenten tilbyr som en tjeneste til andre komponenter. Mens et grensesnitt som en komponent bruker (*uses*) blir kalt et importert grensesnitt (*import interface*), noe som betyr at komponenten er avhengig av at dette finnes for å virke. Et gitt grensesnitt kan bli eksportert av en komponent og importert av en annen. Det er nettopp denne egenskapen som bryter den direkte avhengighet mellom dem. (Grensesnittet ligger mellom). Se figur under:




**Figur 3, Komponent og grensesnitt**

Hovedhensikten med komponentbasert utvikling ligger i muligheten til å lage et system (tjeneste/applikasjon) ved å sette sammen binære gjenbrukbare deler [4]. Dette betyr at det er mulig å lage et større system ved å sette sammen komponenter, for så og utvide eller forbedre systemet ved å legge til nye komponenter, eller ved å bytte ut gamle. Se figur under:




**Figur 4, Utvidelse av eksisterende system**

Av figuren ser en at det lett å utvide eksisterende systemer ved å legge til nye komponenter og/eller klienter. Gamle klienter vil virke sammen med nye komponenter, dersom disse tilbyr samme grensesnitt som den gamle komponenten. Nye klienter kan bruke både gamle og nye komponenter.

### 3.4.5 Rammeverk

Som vi allerede har beskrevet trenger komponentene et ”rammeverk” i form at et kjøretidsmiljø for å virke. Komponentene må klargjøres (instansers og konfigureres) i dette miljøet. De forskjellige leverandørene av tjenere basert på en komponentstandarder, tilbyr som oftest et eget utviklingsmiljø, der forskjellige verktøy er integrert for å effektiviserer og automatiserer prosessene med å installere og klargjøre komponentene. Oppsummert kan en si at det er fem viktige elementer som påvirker komponenter, og hvordan en utfører komponentbasert utvikling [20], [43], [45]:

- **Spesifikasjon:** Med bakgrunn i grensesnittkonseptet, må det finnes en abstrakt beskrivelse av tjenesten som komponenten tilbyr. Denne brukes som en forpliktende ”kontrakt” mellom klient og tilbyder av tjenesten.

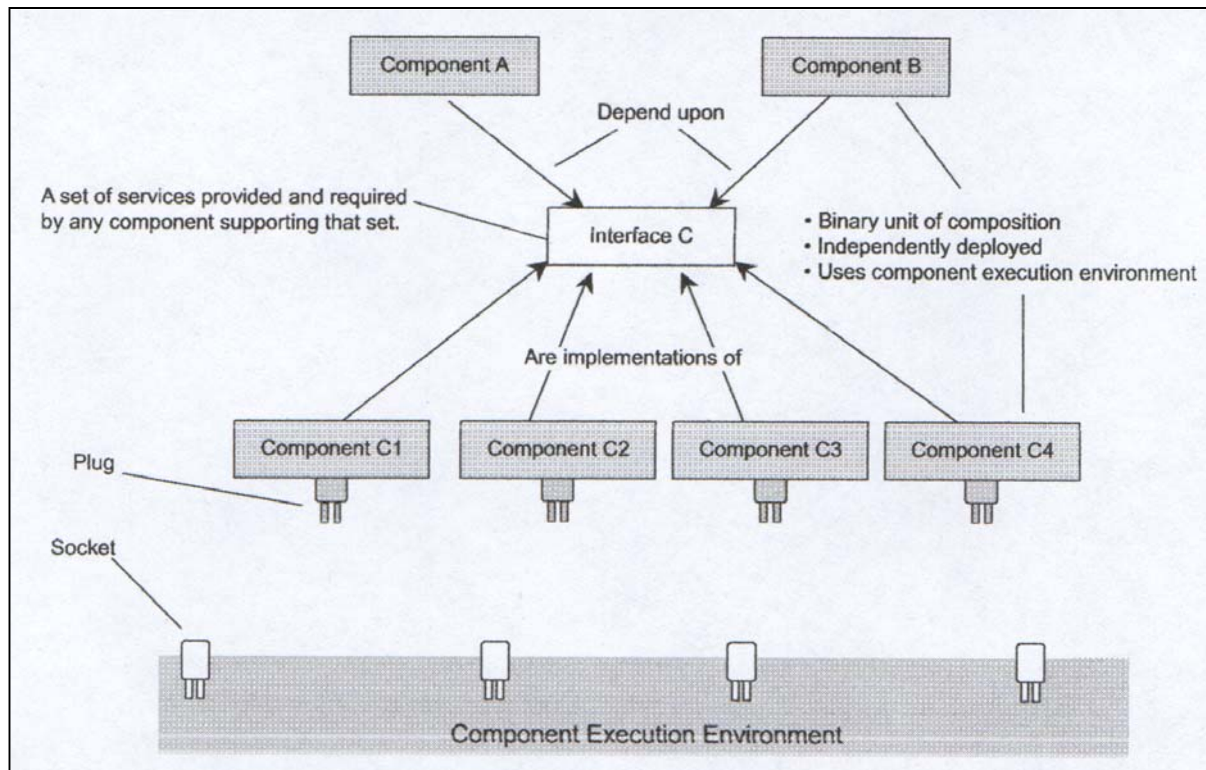
- **Implementasjon:** Komponenten må ha en eller annen form for implementasjon (data og kode), og er forpliktet til å følge grensesnittspesifikasjonen, men kan selv velge best egnet form for hvordan det funksjonelle skal implementeres.

- **Komponentstandard:** Komponenter eksisterer i et gitt (definert) miljø (*environment*) også kalt applikasjonstjeneres komponentmodell. En slik komponentmodell tilbyr komponentene et sett med tjenester, og et sett med lover og regler for hvordan de skal forholde seg til omgivelsene sine. (kjøretids-miljø).

- **Innpakking (*packaging*):** Gjenbrukbare tjenester lages ved at komponentene kan bli innpakket på forskjellige måter. Hver pakke representerer en gjenbrukbar enhet, som før den kan bli tilgjengelig for andre, må registrere seg i applikasjonstjeneren sin komponentmodell.

- **Klargjøring (*deployment*):** Når komponentene er registret i applikasjonstjenerens sin komponentmodell, vil det opprettes en eller flere kjørbare instanser av komponentene. Disse er dermed klare til å motta interaksjoner fra klientene eller andre komponenter.

Det er her spesielt viktig å legge merke til at også de tre siste punktene må være oppfylt for at bruk av komponenter og komponentbasert tjenesteutvikling skal være mulig (Se figur under:).




**Figur 5, Komponent og rammeverk**

Som figuren viser trenger komponentene et kjøretids-miljø (Execution Environment), som de må klargjøres for (Plug/Socket i figuren over). Dette kalles for applikasjonstjenerens-komponentmodell, eller komponentstandard.

Vi vil derfor nå se litt nøyere på hvilke komponentstandarder / komponentmodeller som dagens applikasjonstjenerne bruker (støtter).

# 3.5 Komponentstandarder

## 3.5.1 Innledning

Når en snakker om komponentstandarder skiller en ofte på om komponentmodellen som ligger til grunn for komponentstandarden tilbyr ”rammeverk” og tjenester rettet mot distribuerte miljøer, eller ikke. Med distribuerte miljøer menes da at programvaren kjører på flere sammenkoblede datamaskiner.

De mest brukte ikke distribuerte komponentstandarder som er i bruk i dag er: *Microsoft Component Object Model (COM)*, og *Sun JavaBeans*. De mest brukte distribuerte komponentstandardene som finnes på markedet i dag er *Sun Enterprise JavaBeans (EJB)* og *Microsoft COM+.* Sistnevnte komponentstandard er en sammenslåing og videreutvikling av *Microsoft COM* og *Microsoft Disributed COM (DCOM)*. *Object Management Group (OMG)* [44] har også spesifisert opp en komponentstandard, kalt *CORBA Component Model (CCM).* Denne komponentstandarden bygger på CORBA sine mekanismer for distribuerte objekter, men har også mange likhetstrekk med EJB sin komponentmodell (basert på denne). Siden CCM-standarden er så ny, finnes det ikke kommersielle produkter basert på denne standarden tilgjengelig på markedet i dag. I tillegg til distribuert/ ikke distribuert miljø, snakker en ofte også om hvordan komponentstandarden er realisert, eller om den har en ”knytning” mot en bestemt teknologi eller plattform. Slike ”knytninger” kan f. eks. være mot utviklingsspråk som Sun Java plattform (gjelder JavaBeans og Enterprise JavaBeans), eller operativsystem som Microsoft Windows (gjelder COM, DCOM og COM+). Figuren under viser en oversikt over dagens mest brukte komponentstandarder:.

<table>
<tr><th></th><th>Ikke distribuert</th><th>Distribuert</th></tr>
<tr><td>OMG</td><td>-</td><td>CORBA/CCM</td></tr>
<tr><td>Sun</td><td>JavaBeans</td><td>Enterprise JavaBeans</td></tr>
<tr><td rowspan="2">Microsoft</td><td>COM</td><td>DCOM</td></tr>
<tr><td colspan="2">COM+</td></tr>
</table>

**Figur 6, Dagens mest brukte komponentstandarder**

Alle disse komponentstandardene tilbyr metoder, regler og konvensjoner for hvordan komponentene skal gjøre sine tjenester tilgjengelige, hvordan de blir navngitt (identifisert), samt hvordan de skal bli klargjort for bruk (tilpasset omgivelsene). Disse komponent-standardene er alle konkurrerende standarder (teknologier), og er heller ikke direkte kompatible med hverandre. (selv om f.eks. EJB og CCM har svært mange likhetstrekk). Enkelte utviklingsforum, som *Component Based Development and Integration Forum* (CBDi forum), har imidlertid adressert dette problemet i håp om å få mere åpne og kompatible standarder.

Det finnes imidlertid mange gode løsninger som muliggjør sammenkobling og integrering av de forskjellige systemene, f.eks. i form av ”broer”. Slike løsninger muliggjør at komponenter utviklet for forskjellige komponentstandarder kan ”snakke” med hverandre (bruke tjenester fra hverandre). Det finnes også metoder for hvordan komponentene kan konfigureres og tilpasses for å kunne brukes direkte i andre komponentstandarder (i alle fall i en viss utstrekning). Dette er imidlertid mest aktuelt mellom EJB og CCM, siden disse er så like.

I den videre teoretiske beskrivelsen har vi valgt å legge hovedfokus på distribuerte komponentstandarder, siden dette er mest relevant for problemstillingene diskutert i denne oppgaven. Ikke distribuerte komponentstandarder som JavaBeans og COM, blir derfor bare forklart på et overordnet nivå.

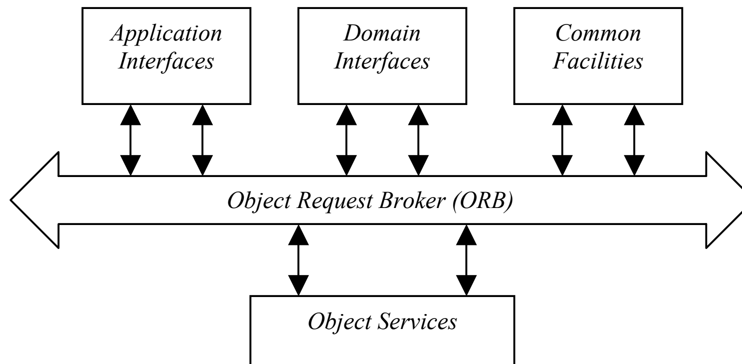
Siden CORBA og CCM viser de fleste mekanismen som ligger til grunn for en objektbasert komponentmodell, har vi valg å beskrive disse med et høyere detaljeringsnivå en de andre. Dette fordi mange av de samme mekanismene, som f.eks. grensesnittsbeskrivelse, lokasjonstransparens, utviklingsprosessene er veldige like i de forskjellige komponentmodellen.

### 3.5.2 OMG CORBA/CCM

På slutten av åttitallet ble sammenkobling av heterogene og distribuerte nettverk mer og mer vanlig. Mange av de store software-selskapene så da at det å utvikle software for denne type nettverk, var en mer kompleks oppgave en tidligere software-utvikling. Mange forskjellige standarder og utviklingsverktøy var i bruk, og ofte kunne ikke de forskjellige systemene brukes sammen. Mye ressurser ble derfor brukt på integrasjon og sammenkobling av de forskjellige systemene. For å bedre på dette problemet dannet en gruppe software selskaper i 1989, *Object Management Group (OMG)*. Dette skulle være en fri og åpen organisasjon, og som kun skulle jobbe med samordning av forslag til nye teknologistandarder. Gjennom delaktighet i OMG kom selskapene frem til en felles standard, arkitektur (rammeverk) og infrastruktur, som muliggjør software utvikling ved hjelp av objekter i distribuerte og heterogene systemer, kalt *Common Objekt Request Broker Architecture (CORBA)* [15]. Dette ”rammeverket” for objektkommunikasjon standardiserte og automatiserte mange av de mest vanlige arbeidsoppgaver innen nettverksprogrammering. Eksempler på dette kan være: registrering, lokalisering og aktivisering av objekter, parameter *marshalling* og *demarshalling* (tilpasse parameterne til en på forhånd bestemt standard for overføringssyntaks), og *operation dispatching* (formidle objektinvokering mellom de involverte parter). Under spesifiseringen av arkitekturen har OMG spesielt hatt fokus på åpenhet, og at invokeringer i distribuerte systemer skulle være mulig uten å måtte ta hensyn til objektenes lokasjon, programmeringsspråk, operativsystem, kommunikasjonsprotokoller eller underliggende maskinvare. Mer en 700 selskaper har etter hvert kommet til, og OMG er forsatt en viktig organisasjon for standardiseringsaktiviteter innen åpne og distribuerte systemer. Siden flere av dagens komponentstandarder bygger på de samme grunnmekanismene som ble utviklet for distribuert objektkommunikasjon, vil vi starte med å forklare grunnmekanismene i CORBA-arkitekturen. Denne arkitekturen betraktes ofte som en referansearkitektur, og mange av de samme grunnmekanismene finner en igjen i Sun Enterprise JavaBeans (EJB) og Microsoft COM objektmodellene. Vi vil også forklare mekanismene til OMG sin nye komponent standard, som bygger på CORBA (utvidelse), kalt *CORBA Component Model (CCM)*.

## CORBA referanse modell

CORBA referansemodellen består av fem hovedelementer: *Application Interfaces, Domain Interfaces, Common Facilities, Object Services* og *Object Request Broker*. [5] Elementene er inndelt etter funkjsonelle egenskaper, karakteristiske egenskaper, samt hvordan disse påvirker hverandre: (Se figur under: ).




**Figur 7, CORBA referanse modell**

***Application Interfaces*** er en samling med grensesnitt som tilbyr tjenester ment for tilpasning av spesifikke applikasjoner. Siden OMG ikke utvikler applikasjoner (ikke til nå i alle fall), er ikke dette grensesnittet spesifisert. (Er beregnet for fremtidige tilpasninger).

***Domain Interfaces*** er en samling grensesnitt som tilbyr tjenester mot bestemte applikasjonsdomener. Eksempler på slike domene-spesifikke tjenester kan være tjenester rettet mot telekommunikasjon, medisin, finans, osv.

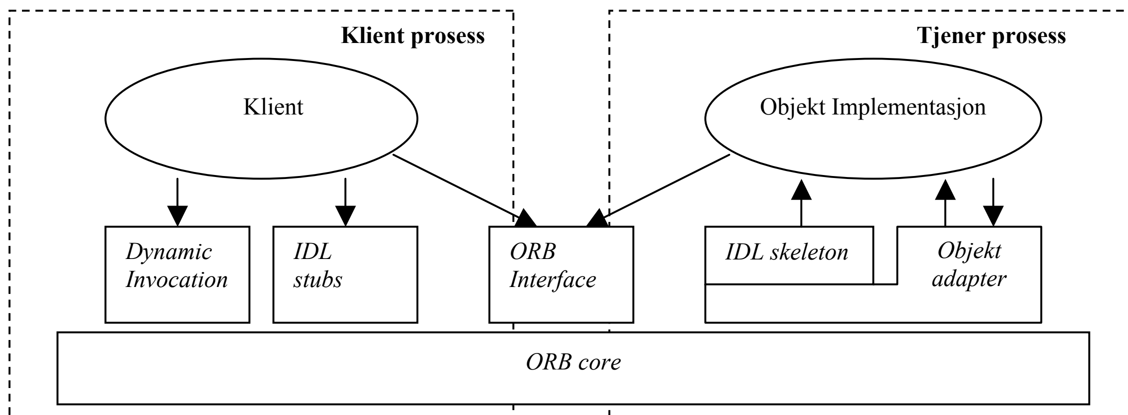
***Common Facilities*** er en samling tjenester beregnet for sluttbruker-applikasjoner. Eksempler på dette kan være dokumentstandarder som spesifiserer hvordan objekter kan utveksles mellom flere typer applikasjoner, f. eks. det å sette inn et regneark objekt inn i et tekstbehandlerdokument. (OpenDoc er et eksempel på dette).

***Object Services*** er en samling tjenester som ikke er domene spesifikke, og som derfor etterspørres av mange typer applikasjoner. Eksempler på slike tjenester er navnetjeneste, oppslagstjeneste (trading), tilstandstjeneste, sikkerhetstjeneste og databaseaksesstjeneste.

***Object Request Broker*:** Dette er kjernedelen i systemet, og har som hovedoppgave å knytte de andre delene sammen (muliggjøre kommunikasjon mellom de forskjellige delene).

## CORBA arkitektur

OMG har spesifisert opp en arkitektur (rammeverk) som muliggjør objekt-kommunikasjon på heterogene og distribuerte systemer. Med kommunikasjon menes da ”mekanismene” som muliggjør at objektene kan forespørre tjenester fra hverandre. (Se figur under: ).




**Figur 8, CORBA arkitektur**

**CORBA objekt** innkapsler tilstandsinformasjon, og tilbyr tjenester, via spesifiserte grensesnitt. Grensesnittene spesifiseres i et språk kalt *Interface Definition Language (IDL)*. Dette er et eget spesifikasjonsspråk, utviklet av OMG for å være uavhengig av programmeringsspråk. IDL beskriver objektenes grensesnitt i form av operasjonene og argumentene som disse tilbyr, samt typen til grensesnittene. IDL definerer et eget sett med typer og tilbyr derfor en metode for konvertering mellom de forskjellige programmeringsspråk sine spesifikke typer, og typer spesifisert i IDL. (gjelder C, C++, JAVA, COBAL, ADA 95, Smalltalk, Perl og Modula-3). Det er utviklet en egen kompilator for slik konvertering, kalt IDL kompilator.

***Object Request Broker (ORB)*** har som hovedoppgave å sørge for kommunikasjonen mellom objektene, dvs. levere klientenes forespørsler til tjenerobjektene (invokering), samt å returnere resultatene tilbake til klientene som gjorde kallene. ORB (sammen med stubs og skeleton) har også som oppgave å skjule objektenes lokasjon. Kommunikasjonen mellom klientene og

tjener objektene blir derfor transparent, sett fra klientens side. Med dette menes at klientene ikke trenger å vite noe om lokasjonen til tjener objektene. Kall-semantikken mot tjenerobjektene er lik uansett om tjenerobjektene ligger på samme prosess, en annen prosess, eller på en helt annen maskin. For å muliggjøre dette, opprettes det en unik id for hvert objekt som finnes tilgjengelig i systemet, dette i form av en objekt-referanse (*object reference*). Klientene bruker denne referansen når de skal gjøre en forespørsel mot et tjenerobjekt, mens ORB bruker denne referansen for å adressere forespørselen mot rett tjenerobjekt.

***Stubs og Skeletons*** blir generert ved at IDL-spesifikasjonen kjøres gjennom en kompilator for å generere type-konvertering; stub på klientside og skeleton på tjenerside. Disse har som oppgave å sørge for transparent invokering. Dette kalles ofte også statisk invokering (*static invocation*), siden alle objektenes grensesnitt er kjent ved kompilering.

***Dynamic invocation:*** I tillegg til statisk invokering gjennom stubs og skeletons, tilbyr CORBA to grensesnitt for dynamisk invokering. Disse kalles *Dynamic Invocation Interface (DII)* og *Dynamic Skeleton Interface (DSI)*. DII brukes på klientsiden, mens DSI brukes på tjenersiden. DII gir klientene muligheten til å gjøre invokeringer mot alle objekter i systemet, også de som de ikke kjenner grensesnittet til under kompilering. Dette gjøres ved at klientene aksesserer ORB sine adresseringsmekanismer direkte, det vil si uten bruk av stubs. Slik adressering muliggjør også ”ikke blokkerende” kall (klienten venter ikke på svar), samt *oneway call* (bare sending, ikke svar). DSI muliggjør at tjenerobjekter kan utvikles uten at det lages statisk skeleton kode for dette (typekonvertering). Via DSI kan ORB levere forespørsler til objekter som selv ikke kjente typene sine ved kompilering, og som derfor ikke har egen skeleton kode.

**Objekt adapter** har som oppgave å tilpasse grensesnittet til et annet objekt, til et grensesnitt forventet av en klient. Det vil si at objektadapteren ligger mellom klient- og tjenerobjekt, og på den måten ”skjuler” grensesnittet til tjenerobjektet. Dermed kan en klient kalle et tjenerobjekt, uten selv å vite hvordan tjenerobjektets grensesnitt ser ut. Objektadapteren har også som oppgave å assistere ORB med å registrere objekter, generere objekt-referanser, aktivere prosessene og/eller objektene, samt å adressere rett objekt. Objektadapter kan derfor betraktes som ”limet” mellom objektenes implementasjon og ORB. CORBA-arkitekturen er laget på en slik måte at det er mulig å tilby flere typer objekt adaptere. I dag brukes to

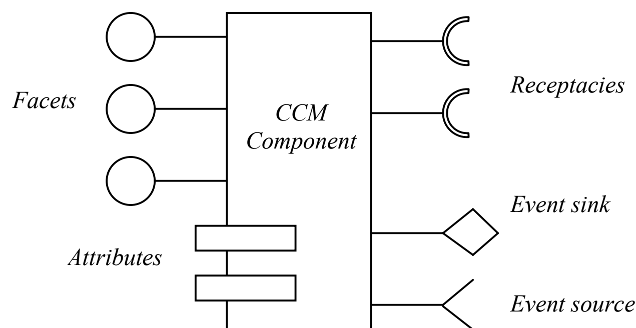
objektadaptere: *Basic Object Adapter (BOA)* og *Portable Object Adapter (POA)*. BOA var den første som ble spesifisert, og tilbyr et relativt lite utvalg av tjenester. Denne er nå nesten erstattet med POA som har et større utvalg av tjenester.

**Interoperabilitet (sammenkobling):** CORBA tilbyr to metoder for sammenkobling av ORB systemer. Disse kalles *Generic Interoperation Protocol (GIOP)* og *Internet Interoperation Protocol (IIOP)*. GIOP spesifiserer overføringssyntaks, samt et sett av meldingsformater for overføring over forbindelsesorienterte nettverk. Mens IIOP spesifiserer hvordan GIOP kan brukes sammen med TCP/IP protokollen.

OMG er nå snart klar til å ferdigstille en ny utgivelse av CORBA-spesifikasjonen, kalt CORBA 3.0. Denne spesifikasjonen inneholder noen svært viktige utvidelser i forhold til forrige CORBA standard: *Asynchronous Method Invocation (AMI)* – asynkron invokering, samt  en nye spesifikasjon (og modell) for utvikling av komponenter, kalt *CORBA Component Model (CCM)*.

## CORBA Component Model (CCM)

CCM  modellen utvider CORBA modellen ved å definere rammeverk og tjenester som muliggjør applikasjonsutvikling ved hjelp av komponenter. CORBA komponentmodellen definerer fire nye begreper, også kalt ”porter”, utfra CCM komponentenes egenskaper:




**Figur 9, Portene (egenskapene) til CCM komponenter**

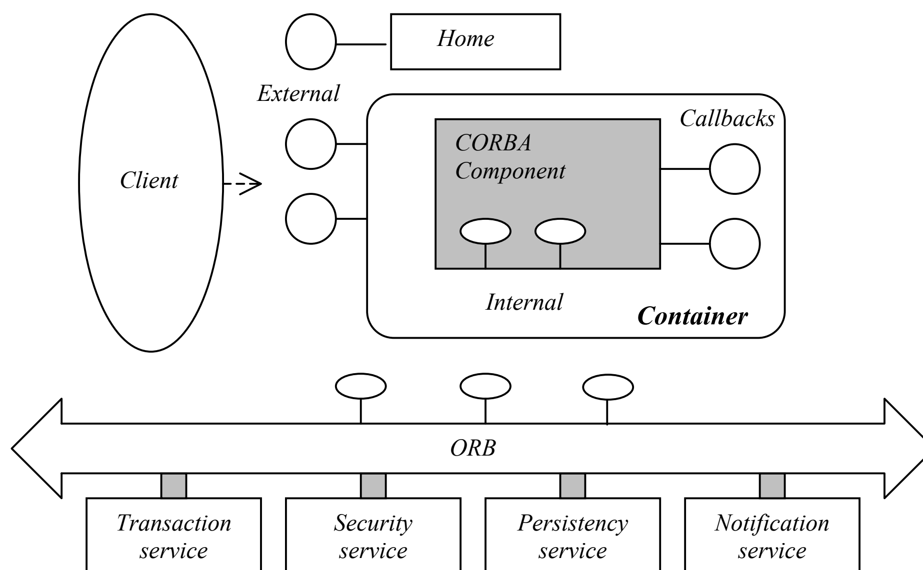
***Facets:*** Dette er grensesnittene som komponenten tilbyr mot klientene (*provided interface*). Disse kan ha forskjellige ”utseende” (egenskaper) mot klientene, og kan invokeres synkront eller asynkront.

***Attributes:*** For å muliggjøre konfigurering utvider CCM-modellen attributt-begrepet, slik at disse kan konfigureres (presettes under klargjøringsprosessen) ved hjelp av det tilhørende konfigureringsverktøyet. Konfigurering av attributter bruker også *exeptions* (unntak), slik at det er mulig for andre komponenter å kontrollere når en attributt konfigureres (de kan abonnere på melding om forandring).

***Receptacles:*** CCM innfører begrepet "*object connections*" for koblingen den gjør mot andre komponentinstanser (referanse). "Portene" som slike koblinger (referanser) kan kobles til, kalles *receptacles*. Portnavnene til disse "portene" (*receptacles*), tilbyr en standard måte for å beskrive grensesnittene som må være på plass, for at komponenten skal virke som foreskrevet. (Avhengighet mot andre komponenter).

***Event sockets:*** Komponenter kan også utveksle informasjon med hverandre ved hjelp av asynkrone hendelse. Dette gjøres ved at komponentene "abonnerer" på hendelser (*publish /subscribe* forhold), ved å beskrive *source/sinks* avtalene i sine respektive definisjoner.

CCM utvider også CORBA arkitekturen med noen nye begreper [18]. (se figur under).



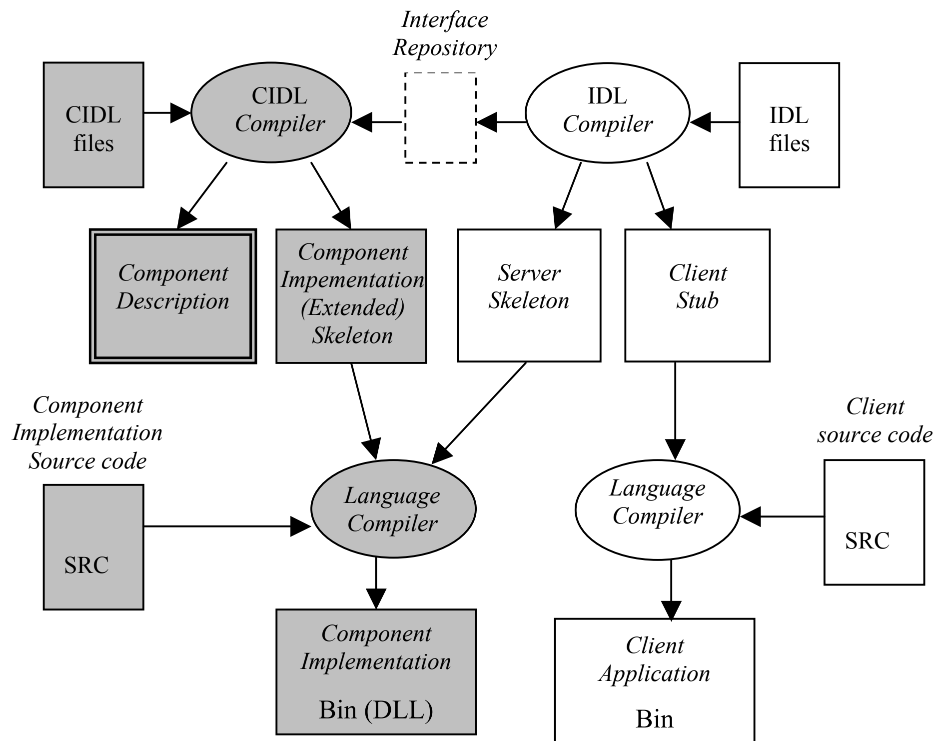

**Figur 10, Container og grensesnitt i CCM komponentmodellen**

Som vi ser av figuren er arkitekturen utvidet med noen standard tjenester: transaksjon, sikkerhet, persistens (tilstand) og notifikasjon. Disse tjenestene var i CORBA-arkitekturen

standardisert som *Object Services* (felles tjenester), men er nå integrert direkte i CCM-arkitekturen. Et eksempel på hvordan disse tjenestene bruks av komponentene og CCM-arkitekturen, er hendelser (*events*). Denne tjenesten er realisert ved hjelp av *notification*-tjenesten, og muliggjør kommunikasjon mellom komponentene ved hjelp av hendelser (*events*). Arkitekturen tilbyr også tjenester for hvordan komponentene skal konfigureres og klargjøres for kjøretidsmiljø. Arkitekturen innfører begrepet ”container” for å beskrive (ved hjelp av et sett API'er) omgivelsene og kjøretidsmiljøet til komponentene. Hovedoppgavene til en container kan oppsummeres i følgende punkter:

- Aktivere og deaktivere komponent-instansene, slik at dette blir tilpasset tilgjengelig prosesseringsressurser og minne.
- Lage et ”rammeverk” som muliggjør bruk av *callbacks* (tilbakekall) grensesnitt, slik at ORB og container kan informere komponentene om hendelser. (brukes blant annet ved transaksjon og notifikasjon tjeneste).
- Adressere standardtjenestene som arkitekturen tilbyr (transaksjon, sikkerhet, persistens og notifikasjon). Klientene slipper dermed å lokalisere disse tjenestene.
- Styre objektadapteren sin beslutning om hvordan referansene til komponentene (instansene) skal lages (gjelder POA).

Klientene aksesserer komponenten direkte via eksterne grensesnitt (*external interface*), mens komponentenes interne grensesnitt (*internal interface*) brukes av komponenten for å forespørre tjenester mot containeren. Tilbakekall grensesnittet (*callback interface*) brukes av containeren for å forespørre tjenester mot komponenten. Grensesnitt kontraktene mellom komponenten og containeren er beskrevet i egen fil, kalt *Component Implementation Framework (CIF)*. CCM arkitekturen har også innført noe som kalles ”*Home*” (en enkel database), som brukes til ”*life cycle*” styring av komponentinstansene. Eksempler på dette kan være *create* (lage) og *remove* (slette) instanser av komponentene. Klientene kan også aksessere *home* for å sjekke ”tilstanden” til en komponentinstans. For å forenkle utviklingen av komponenter har OMG også utvidet IDL beskrivelsen med et deklarativt språk, kalt *Component Implementation Definition Language (CIDL)* [19]. Noen kaller dette også for IDL3. Denne type beskrivelse brukes av CCM for å beskrive komponentenes implementasjon, tilstandsinformasjon, samt komponentenes home (*life cycle*) database. (se figur neste side).




**Figur 11, IDL3 til komponent kompilering**

Som figuren viser vil innføringen av CIDL medføre at utviklingsprosessen utvides til nå også å inneholde en CIDL kompilator. Komponentens implementasjon, generert av CIDL-kompilatoren (via språk kompilatoren), kalles *executors*. Slike *executors* kan igjen pakkes inn i "*assembly files*" (kjørbare filer), som lastes og installeres på komponent-tjeneren. Disse prosessene kalles for pakking (*packaging*) og klargjøring (*deployment*). CMM-arkitekturen tilbyr tjenester for å automatisere disse prosessne, ved at komponentene og komponentenes avhengighet, beskrives i en *Open Software Description (OSD)* beskrivelse (basert på XML). I tillegg har OMG tilrettelagt for bruke av CORBA-script for klargjøring og idriftsetting av komponentinstanser, slik at det å klargjøre, konfigurere og administrere komponentene og komponentenes instanser skal være en "enkel" prosess.
Siden CCM er en helt ny komponentstandard (ikke offisielt utgitt ennå), finnes det ennå ikke noen ferdig implementasjon av denne standarden.

### 3.5.3 Sun JavaBeans/Enterprise JavaBeans

Via programmeringsspråket Java, tilbyr Sun en ikke-distribuert komponentstandard, kalt *JavaBeans*, samt en distribuert komponentstandard kalt *Enterprise JavaBeans (EJB)*. Disse har flere likhetstrekk, men også helt klare forskjeller. JavaBeans sies å være en klientside-komponentstandard, og med dette menes at JavaBeans for det meste brukes til utvikling av elementer for grafiske brukergrensesnitt. EJB derimot er utviklet med tanke på forretningslogikk i distribuerte enterprise miljøer.

**JavaBeans**

JavaBeans sies å være plattformuavhengig (uavhengig av underliggende software og hardware), men siden den er ”knyttet” til Java språket kreves det en implementasjon av en Java virtuell maskin for å kjøre. Sun definerer JavaBeans på følgende måte:

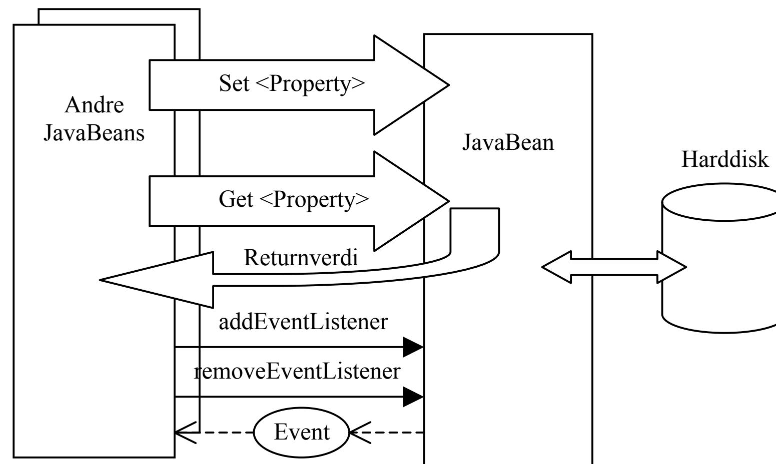
> *JavaBeans: ”A JavaBean is a reusable software component that can be manipulated visually in a builder tool.*

Dette er en veldig ”vag” definisjon som kan ha mange muligheter: En JavaBean kan være en liten gjenbrukbar software komponent, som en ”knapp” eller et ”vindu” i et grafisk brukergrensesnitt, eller en mere komplett funksjonalitet som f. eks. et regneark. En JavaBean vil uansett ha følgende egenskaper:

- *Properties* (egenskaper) gjør det mulig å programmere og tilpasse en JavaBean.
- *Events* (hendelser) muliggjør kommunikasjon mellom JavaBeans.
- *Persistence* (tilstand) gjør det mulig å lagre JavaBean tilstand, for senere bruk.
- *Introspektion* gjør det mulig å analysere hvordan en JavaBean virker.
- *Methods* (metode invokering) virker på samme måte som mot et Java objekt.

JavaBeans kan ha tilstandsinformasjon som leses og settes av *get- / set- properties* metodene. Denne tilstandsinformasjonen gjøres persistent ved hjelp av *Java object serialization* API'en. Komponentene ”snakker” med hverandre ved hjelp av *events* (hendelser). Dette gjøres ved at komponentene som ønsker å bli varslet om hendelser, først må implementere et passende grensesnitt slik at de ”lytter” etter visse type hendelser (blir *Listener*). Deretter må *Listener*

komponentene registrere seg hos de forskjellige komponentene den ønsker hendelser fra, slik at disse vet hvilke Listener komponenter den skal sende hendelser til. (Komponentene abonnere på hendelser fra hverandre). *Properties*- og *event*-mekanismene kan også kombineres slik at komponenten kan abonnere på hendelser om at properties egenskapene til en komponent endres (kalles *PropertyChangeListener*). I tillegg tilbyr JavaBeans API'et metoder for lagring av komponentenes tilstandsinformasjon på disk. (Se figur under: ).

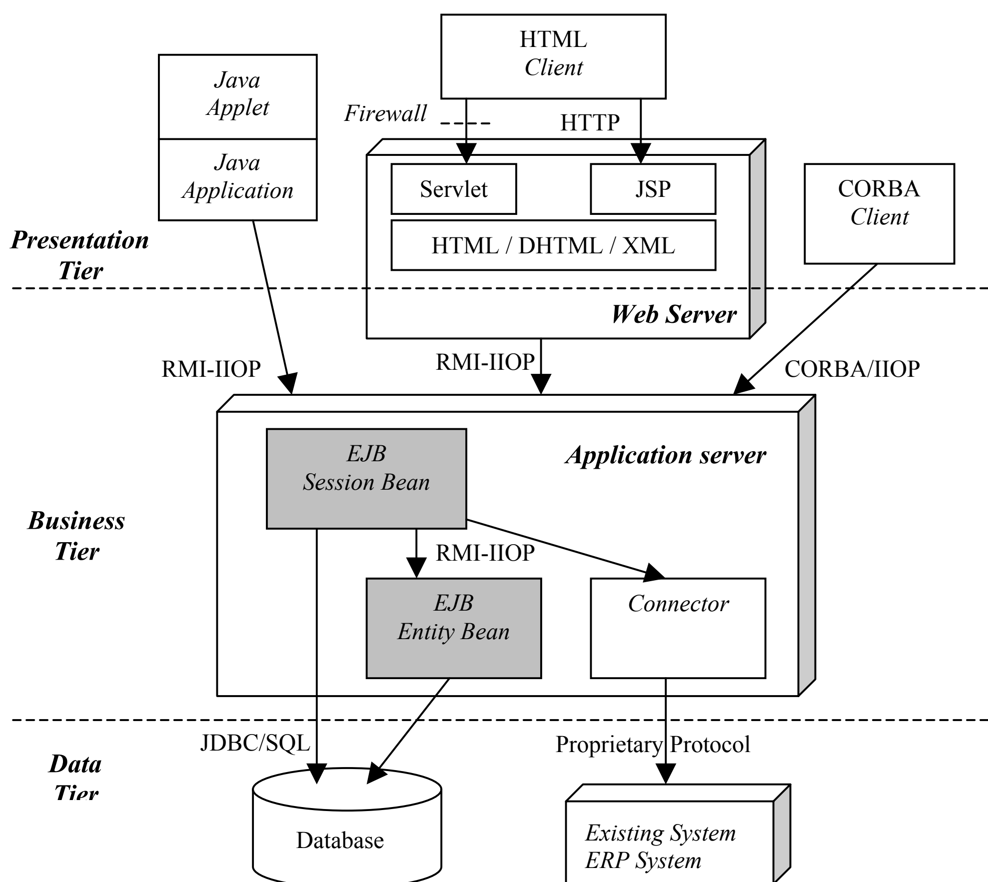



**Figur 12, JavaBeans komponentmodell**

JavaBeans blir vanligvis distribuert ved hjelp av et pakket arkivformat kalt .JAR. I slike .JAR filer legges det klassefiler, seriealiserte versjoner av komponenter, samt  en fil som beskriver innholdet i .JAR filen.

## Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) [21] er Sun sin distribuerte komponentstandard [3]. Denne ble utviklet med tanke på å forenkle utviklingen av forretningskritiske applikasjoner. EJB inngår som komponentmodell og bindeledd mellom de forskjellige teknologier som inngår i Sun sin tjenerside-teknologiplattform, kalt *Java 2, Enterprise Edition (J2EE*) [17]. Dette er en plattform utviklet med tanke på å forenkle prosessene for utvikling og integrering av forretningsapplikasjoner, samt gjøre tilpasninger av eksisterende systemer, slik at tjenesten som disse tilbyr kan aksesseres fra web. Arkitekturen er basert på en trelags arkitektur med presentasjonslag, forretningslogikk lag og data-aksess lag. (Se figur under: ).




**Figur 13, J2EE applikasjonsarkitektur**

**Presentasjonslaget (*Presentation Tier*):** Som vi ser av figuren er det mulig å bruke flere typer klienter i J2EE arkitekturens presentasjonslag, som Java applets, Java applikasjoner, Java servlets, Java Server Pages (JSP), statiske HTTP kode, eller CORBA. Java klientene bruker *Java Naming and Directory Interface (JNDI)* for å lokalisere forretningslagets

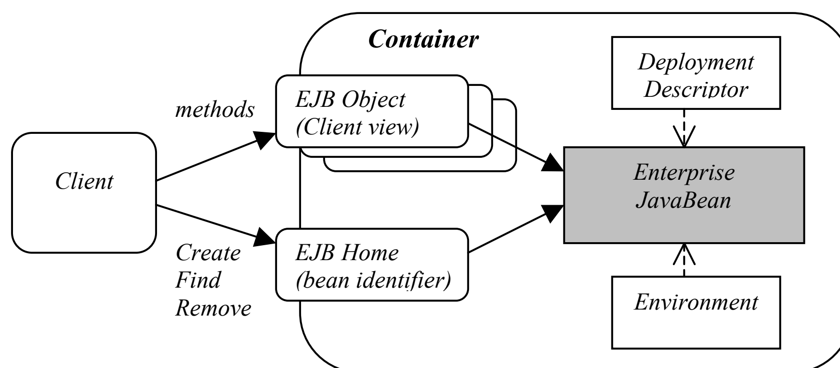
komponenter, og RMI-IIOP til å invokere metoder mot disse. CORBA klientene bruker *CORBA Naming Service* for å lokalisere forretningslagets komponenter, og CORBA/IIOP til å invokere metoder mot disse.

**Forretningslogikklaget (*Business Tier*):** Dette laget består av forretnings- og datalogikk. EJB består i dag av session beans (forretningslogikk komponenter), entity beans (datalogikk komponenter), og Message-driven beans (meldingsdrevet beans). I tillegg kan dette laget tilby flere forskjellige mellomvarelags-funksjoner som f. eks. håndtering av transaksjoner, tilstandsinformasjon, sikkerhet og persistens.

**Dataaksesslaget (*Data Tier*):** For å gjøre aksess mot de forskjellige databasesystem som er i bruk i dag, tilbyr J2EE plattformen *Java Database Connectivity (JDBC)*, eller SQL/J. Integrering mot eksisterende system kan også gjøres via *J2EE Connectors*.

**EJB komponent modell**

EJB sin komponentmodell bruker begrepet "container" som et viktig konsept for å beskrive omgivelsene til komponentene. (Se figur under: ).




**Figur 14, EJB komponentmodell**

*Enterprise JavaBeans* blir innpakket og klargjort for kjøretidsmiljøet ved hjelp av containere. Under klargjøringsprosessen generere containeren automatisk et *EJB Home* grensesnitt, samt et *EJB Object* grensesnitt for hver EJB komponenten. *EJB Home* grensesnittet identifiserer EJB klassen, og brukes til å lage (*create*), identifisere (*find*) og slette (*remove*), EJB komponentinstanser. *EJB Object* grensesnittet tilbyr klientene aksess til EJB komponentens

forretningslogikk. Alle klientforespørsler, uansett om det går via et *EJB Object* eller *EJB Home* grensesnitt, går via containeren for å sikre at transaksjon, tilstand, sikkerhet og persistente egenskaper blir ivaretatt for alle operasjoner. EJB-teknologien tilbyr både transiente og persistente komponentinstanser (komponent objekt). En transient komponentinstans kalles *Session bean*, mens en persistent komponent instans kalles *Entity bean*.

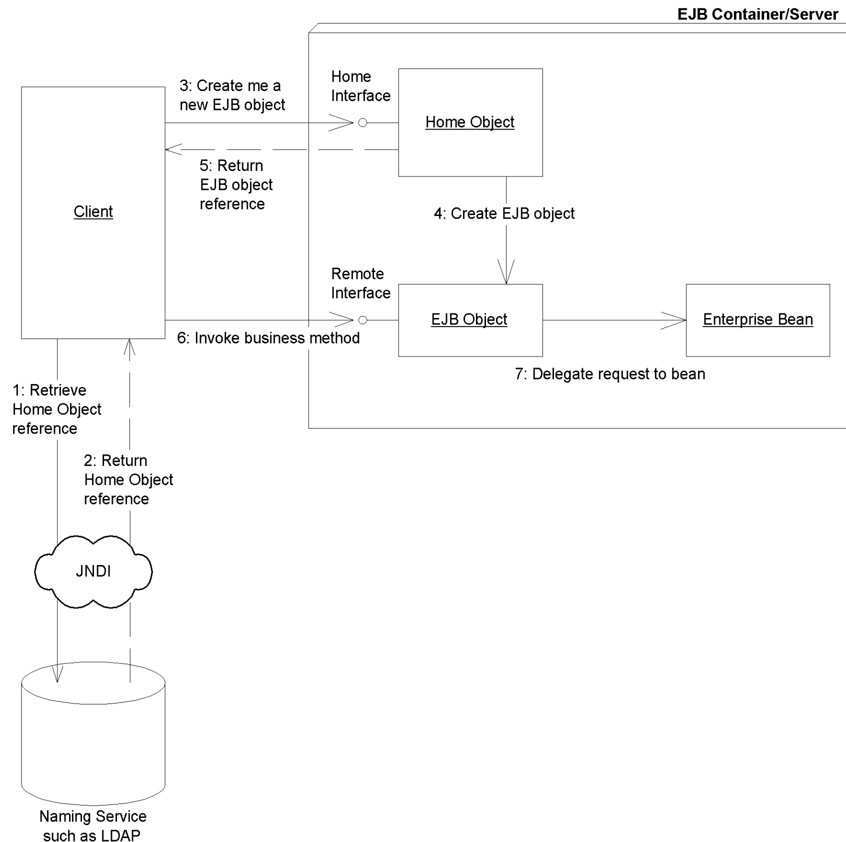
***Session beans*** opprettes av klientene, og varer i de fleste tilfeller like lenge som klient/tjener-sesjonen. En session bean representerer funksjonalitet, ikke data, og utfører operasjoner som databaseaksess eller beregninger på forespørsel fra klientene (kan betraktes som verb, siden de utfører noe). En session bean kan være uten tilstand (klienten får da tilordnet en vilkårlig komponent instans), eller kan via container håndtere tilstandsinformasjon for metoder eller transaksjoner (metodekallene til klienten invokeres da mot samme komponent instans). Session bean kan håndtere transaksjoner, men kan vanligvis ikke tilbakerulles (*recover*) ved systemkrasj.

***Entity beans*** er en komponentinstans (komponent objekt) som representerer persitente dataelementer fra et datalager (database eller ekstern disk). Entity bean komponentinstansene identifiseres av et primærnøkkel objekt. Entity bean kan håndtere sin egen persistens, eller den kan delegere dette ansvaret over til containeren. Entity beans kan håndtere transaksjon, og kan også tilbakerulles (*recover*) ved systemkrasj.

***Message-driven beans*** har mange likhetstrekk med session beans, siden de også kan betraktes som verb som utfører en handling, men message-driven beans kan bare kalles ved at det sendes en melding til disse.

Som allerede nevnt er det mulig å bruke flere typer klienter mot EJB, men uansett om klienten aksesserer EJB ved hjelp av CORBA/IIOP eller RMI-IIOP, vil en klient kode inneholde:

1. Oppslag (Lookup) av Home Objektet sin referanse. (Punkt 1-2 i figur).
2. Opprettelse av et EJB objekt, ved hjelp av Home objektet. (Punkt 3-5 i figur).
3. Kall av forretningslogikk metodene til EJB objektet (instansen). (Punkt 6-7 i figur).
4. Slett EJB objektet (instansen). (ikke tegnet inn i figur under).

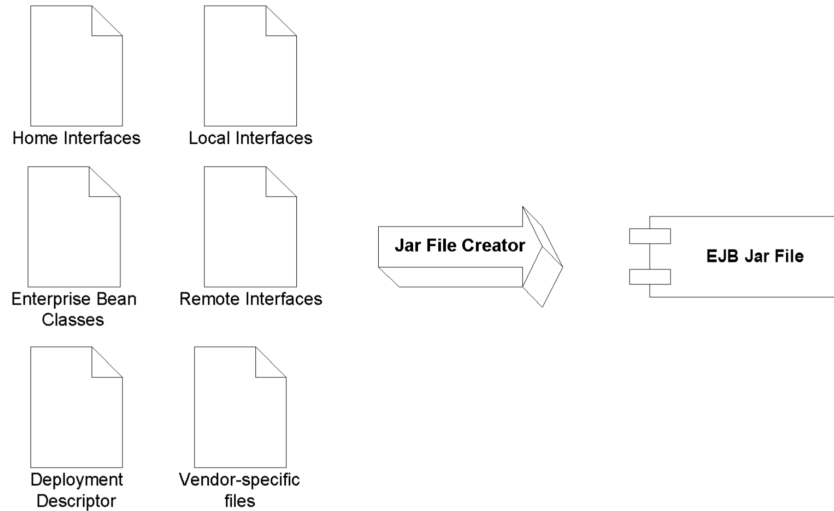



**Figur 15, Sekvensdiagram for en EJB invokering**

Som figuren over viser, starter klienten med å gjøre oppslag av referansen til Home objektet til EJB komponenten (1,2). Klienten bruker så denne referansen til å "fortelle" Home objektet at det skal opprette et EJB objekt (3,4,5). Klienten kan så kalle metodene som EJB komponenten tilbyr, via Remote Interface til EJB objektet (6,7).

Som vist i figuren over, bruker klientene en navnetjeneste for oppslag av referansen til Home-objektet til EJB-komponenten (punkt 1 og 2 i figuren over). En slik navnetjeneste realiseres som oftest ved hjelp av JNDI (*Java Naming and Directory Interface*), og kan enten realiseres som en selvstendig tjeneste, eller som en integrert del av applikasjonstjeneren (BEA WebLogic bruker sistnevnte metode). Oppslaget i navnetjenesten er basert på et "*nickname*" som kommer fra EJB komponentenes descriptor (Velges av utvikleren, og kan enten hardkodes i descriptoren, eller settes automatisk av utviklingsverktøyet). EJB tjeneren sin container tar seg av koblingen "*nickname*" til Home objekt (gjør automatisk "*bind*").

EJB komponentstandarden bruker et eget filformat for "innpakking" av komponenter, kalt EJB-Jar. EJB 2.0 spesifikasjonen definerer også en deployment descriptor i form av XML fil, som beskriver strukturell informasjon om komponentene, konfigurasjonskrav, samt container parametere. Figur under viser hvilke elementene som normalt inngår i en EJB-Jar file:




**Figur 16, Elementer i en EJB Jar file**

Det finne i dag mange leverandører av EJB tjenere (Se appendix B for en oversikt over teknologier som inngår i J2EE plattformen), men vi har nevnt noen av de mest kjente som finnes på markedet i dag er:

- BEA – Weblogic [46] (www.bea.com)
- IBM – Websphere [47] (www.ibm.com)
- Borland – Borland AppServer [48] (www.borland.com)
- Sybase – EAServer [49] (www.sybase.com)
- Oracle – Oracle application server [50] (www.oracle.com)
- Veldig mange flere... ( > 30)

### 3.5.4 Microsoft COM/DCOM/COM+

Microsoft har på lik linje med OMG og SUN utviklet komponentstandarder. *Component Object Model (COM)* [13] var Microsoft sitt første forsøk på å lage en komponentstandard. Denne komponentstandarden ble utviklet med tanke på intern bruk i Windows-operativsystemene, og var i første omgang ment for sammenkobling og integrering av de forskjellige windows-applikasjonene. (Et eksempel på dette er en Microsoft teknologi kalt *Object Linking and Embedding (OLE)*. Denne teknologien er basert på COM komponenter, og muliggjør at f. eks. regneark objekter kan "klippes" ut og "limes" inn direkte i tekstbehandlingsdokumenter). *Distributed COM* eller DCOM standarden ble første gang introdusert med Windows NT 4.0, og var en utvidelse av COM, slik at komponenter også kunne brukes i distribuerte miljøer. Siden har Microsoft utviklet en "ny" komponentstandard ved å integrert COM og DCOM, samt å utvidet med en del ny funksjonalitet. Denne "nye" komponentstandarden som første gang ble introdusert i Windows 2000, har Microsoft valgt å kalle COM+.

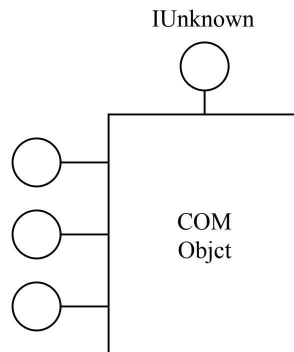
Vi har valgt å beskrive teorien for Microsoft sine komponentstandarder (COM/DCOM/COM+) på et overordnet nivå, siden mange av "mekanismene" som komponentstandarden bruker (f. eks. lokasjonstransparens), er basert på de samme teknologi-konseptene som allerede beskrevet for OMG CORBA modellen.

**Microsoft COM**

COM er som tidligere nevnt Microsoft sin ikke-distribuerte komponentstandard. Denne komponentstandarden er svært integrert og "knyttet" til Microsoft sine Windows operativsystem. Dette har både positive og negative sider. Positive i form av at komponentstandarden er mye utbrett, siden Microsoft har stor markedsdominans, og at komponentstandarden er godt integrert med operativsystemet (optimalisert). Negativt av den samme nære knytning mot operativsystemet, noe som vanskeliggjør bruk av COM komponenter sammen med andre operativsystemer som f. eks. Unix og Linux.

COM er en binær komponentstandard som bygger på en tidligere definert standard for fjernkall kalt DCE RPC. Dette innebærer at alle funksjonelle kall, utføres som om de var lokale kall. Kall mellom de forskjellige komponentene blir utført på to forskjellige måter,

avhengig av om komponentene ”kjører” i samme prosess, eller i forskjellige prosesser men på samme datamaskin. Et COM-miljø kan noe forenklet sies å bestå av COM-komponenter med sine grensesnitt, et ”*Class factory*” objekt, samt en samling med operativsystemfunksjoner. Grensesnittene brukes på samme måte som hos andre komponentstandarder, ved at grensesnittbeskrivelsen sier hvilke tjenester komponentene tilbyr (i form av metoder), mens den skjuler hvordan komponentene realiserer disse funksjonene. *Class factory* objektet brukes for å lage nye objekter (komponentinstanser), og virker stort sett på samme måte som et EJB Home objekt (lage/slette komponentinstanser). COM grensesnittene må altid arve fra et fast grensesnitt kalt *IUnknown*. (Se figur under: ).



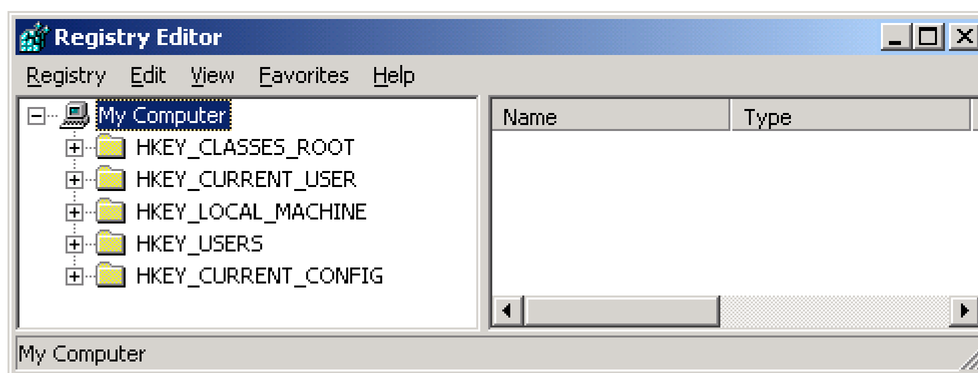

**Figur 17, COM komponent objekt**

*Iunknown*-grensesnittet har tre virtuelle funksjoner kalt *QueryInterface()*, *AddRef()* og *Release()*. Disse funksjonene brukes til å holde rede på hvor mange klienter som til enhver tid ”bruker” en komponent, samt ”mekanismene” som gir klientene mulighet til å oppdage komponentenes grensesnitt dynamisk. Eneste måte å aksessere en COM-komponent på, er via en peker til et grensesnitt (såkalte *virtual table*, eller *V-table*). Det er nettopp bruken av slike pekere som gjør IUnknown grensesnittet så spesielt, siden dette muliggjør at grensesnittene kan brukes (aksessere), helt uten at brukeren (klienten) vet noe om selve grensesnittet. For å identifisere en COM-komponent og alle COM-grensesnitt, bruker COM-komponentmodellen et unikt 128 bits tall, kalt *Globally unique identifier (GUID)*. Når GUID-identitet brukes for å identifisere en COM-klasse, kalles den imidlertid for *ClassID (CLSID)*, mens dersom en GUID-identitet brukes til å identifisere et grensesnitt, kalles den for *Interface Identifiers (IID)*. Disse identitetene registreres i *Windows Registry* (operativsystemet interne databasen). Når en klient ønsker å aksessere en komponent må den først opprett en komponentinstans. Dette gjøres ved å kaller en metode kalt CoCreateInstance().

Som allerede nevnt registreres alle CLSID (class)- og IID (interface)- identitetene i Windows sitt interne register, kalt Windows Registry. Slik registrering gjøres normalt direkte i utviklingsverktøyet som brukes til utvikling av objektene/komponentene, som f. eks Microsoft Visual C++. Windows tilbyr også et program for direkte editering av innholdet i Windows Registry, kalt REGEDIT. Dette programmet kan også brukes til å undersøke hva som allerede er registrert i Windows Registry. Dette gjøres ved hjelp av følgende punkter:

1. Klikk Start button på task bar.
2. Velg Run fra Start menyen.
3. Skriv inn REGEDIT i Run vinduet.
4. Klikk på OK button.

REGEDIT startes da opp og viser følgende skjermbilde:

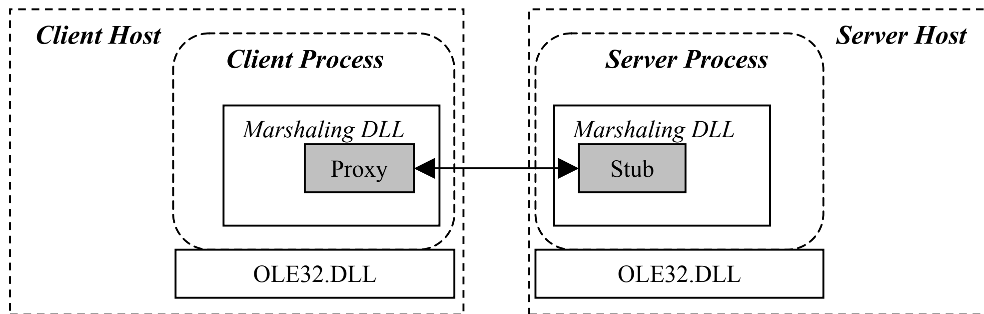



**Figur 18, Registry Editor**

Ved så å gå under HKEY_CLASS_ROOT, og ned til CLSID vil en få opp en oversikt over alle klassene som er registret i Windows Registry. Under CLSID feltene vil en videre finne masse viktig informasjon som gjelder for hver CLSID. Tilsvarende informasjon kan en også finne for grensesnittene (IID), men da under *Interface*-feltet i REGEDIT.

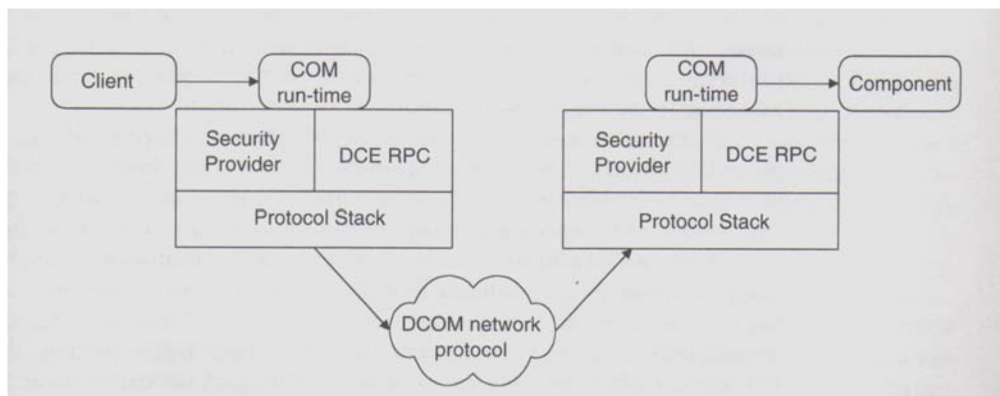
## Microsoft DCOM

Som allerede nevnt er DCOM en utvidelse av COM, for distribuerte miljøer. Utvidelsen består i korte trekk av mekanismer som muliggjør lokasjonstransparens. Dette er oppnådd ved at det brukes *marshaling* (felles format på overført informasjon ved funksjonskall), samt proxy- og stub-objekter. (se figur under).




**Figur 19, Lokasjonstransparens i DCOM**

I tillegg er det lagt stor vekt på sikkerhetsmekanismer i DCOM siden komponentstandarden muliggjør kall mot komponenter som ligger på andre datamaskiner (Se figur under for prinsippskisse av teknologien som inngår i DCOM).




**Figur 20, DCOM: kommunikasjon mellom komponenter på forskjellige maskiner**

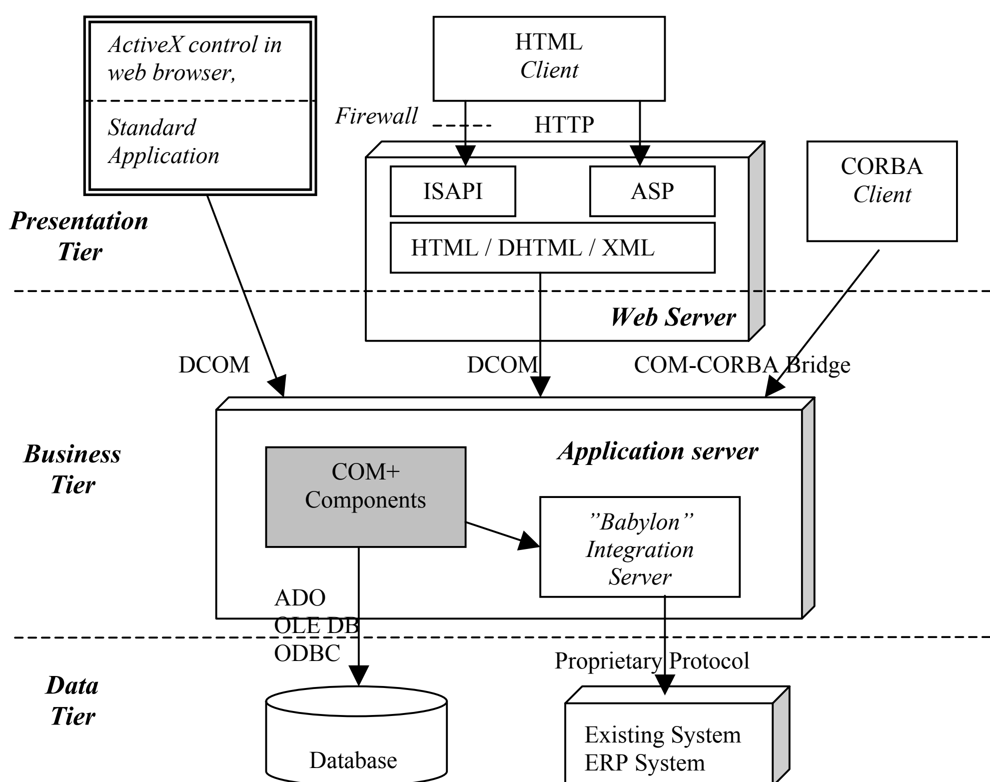
Som figuren over viser innfører DCOM en protokoll-stakk, for overføring mellom forskjellige datamaskiner. I Microsoft sammenheng sies DCOM å være "TCP/IP-protokollen" til objektene. Teknologikonseptene introdusert med DCOM brukes fortsatt, også nå etter at Microsoft har kommer med en ny komponentstandard, kalt COM+.

## Microsoft COM+

På lik linje med Sun J2EE plattform, tilbyr Microsoft en egen plattform for utvikling tjenerside forretningskritiske applikasjoner, kalt *Microsoft Windows Distributed interNet Application Architecture (Windows DNA)* [8] . Det er imidlertid en stor forskjell på Sun sin J2EE plattform og Microsoft sin Windows DNA plattform, og det er at mens Sun tilbyr åpne teknologispesifikasjoner som kan brukes av andre tjener leverandører, holder Microsoft seg til

proprietær standarder, og som kun er tilgjengelig gjennom Microsoft sine Windows operativsystem. En bør allikevel være klar over at Microsoft har endret noe på sitt syn på åpen arkitekturer, og spesielt gjelder dette innen Microsoft DNA arkitekturen, der mange av ”konseptene” er hentet fra åpne standardiseringsorganisasjoner som World Wide Web Consortium (W3C) og Internet Engineering Task Force (IETF). COM+ brukes som komponentmodell, mens DCOM brukes som bindeledd mellom de forskjellig deler som inngår i Windows DNA plattformen.

Windows DNA plattformen er også basert på en trelags arkitektur. (Se figur under: ).




**Figur 21, Windows DNA applikasjonsarkitektur**

**Presentasjonslaget:** I Windows DNA kan presentasjonslaget bestå av forskjellig typer klienter, som *ActiveX controls* i en web browser, selvstendig applikasjoner, *Internet Server API (ISAPI)*, *Active Server Pages (ASP)*, eller statiske web sider. I tillegg er det mulig å bruke CORBA-klienter via en COM-CORBA bro. Klientene bruker Microsoft Active Directory for å lokalisere komponentene i forretningslogikklaget, og DCOM for å invokere metodene til komponentene.

**Forretningslogikklaget:** Dette laget består av forretningslogikk og data, innkapslet i COM+ komponenter. All invokering mot COM+ komponenter går via COM+ kjøretidsmiljøet, som på den måten kan utføre mellomvarefunksjoner som transaksjons- og sikkerhetshåndtering.

**Dataaksesslaget:** Windows DNA plattformen tilbyr flere forskjellige teknologier for aksess mot datakilder, som *Active Database Objects (ADO)*, *OLE DB*, og *Open Database Connectivity (ODBC)*. Microsoft *Babylon Intergration Server* muliggjør kobling mot eksisterende enterprise informasjonssystemer.

Se appendix C for en oversikt over teknologier som inngår i Windows DNA plattformen.

# 4 Dynamisk Konfigurering

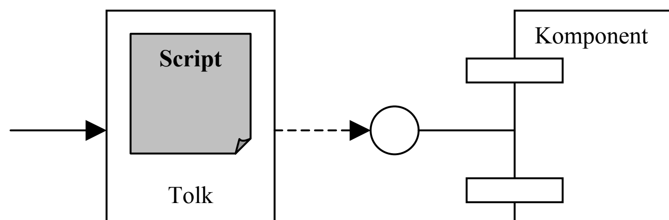
## 4.1 Innledning

Hypotese:
Ved å gjøre komponenter eller hele tjenester dynamisk konfigurerbare, vil man kunne tilpasse tjenestene dynamisk til hver enkelt klients type, behov, bruksmønster og kjøretidsomgivelser.

I forrige kapitel så vi at nettbaserte tjenester ofte er basert på komponentteknologi, og at hovedideen bak komponentteknologi er gjenbruk. Det vil da være naturlig å stille følgende spørsmål:

1. **Er det mulig å lage en eller annen form for konfigurering av komponenter?**

Siden hovedideen med komponenter er gjenbruk må det vel være "smart" å lage en eller annen form for konfigurering av komponentene. På den måten vil komponentene enklere kunne tilpasses nye situasjoner, krav eller ønsker. Komponentene blir mere fleksible og muligheten for gjenbruk øker ytterligere. En kan si at ved å innføre konfigurering, vil komponentene bli noe mere generelle, ved at tjenestespekteret som komponentene tilbyr, tilpasses ønsket bruk. En måte å implementere konfigurering kunne på f. eks. være ved bruk av script. (se figur under).




**Figur 22, Konfigurering av komponent ved hjelp av Script**

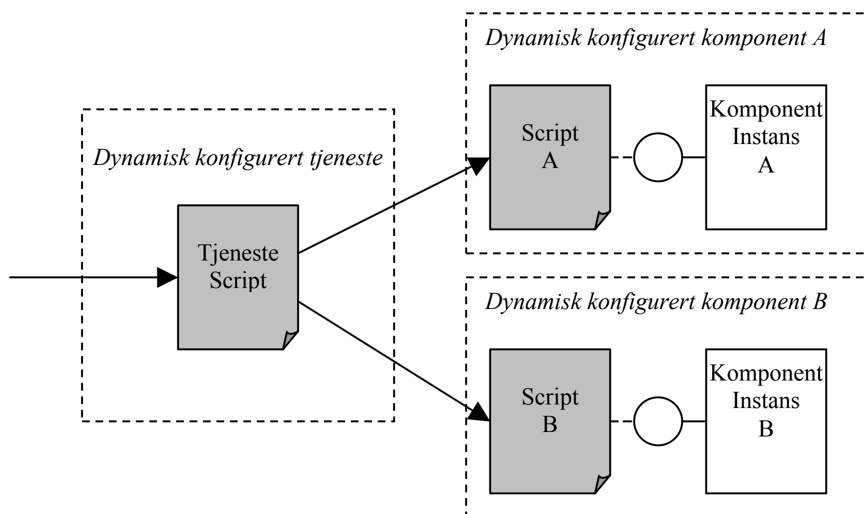
2. **Er det mulig å lage dynamisk konfigurering av komponenter?**

Dersom det lar seg gjøre å konfigurere komponenter, så må det vel også være "smart" å la en slik konfigurering være "dynamisk" i kjøretidsmiljøet (*runtime environment*); direkte i sanntid, ved tjenesteforespørslene mot en komponent. En slik dynamisk konfigurerbar komponent vil ha mulighet til å tilpasse sine funksjonelle tjenester, på bakgrunn av informasjon den mottar fra klientene under selve invokeringen. Siden selve konfigureringen

skal være så dynamisk som mulig, antar vi at et tolket script-språk vil være godt egnet for dette. Script er ofte enkle å forandre, og kan dessuten ofte genereres automatisk av et verktøy. (f. eks vil *deployment* prosessen til komponentstandardene være et godt egnet sted for dette).

3. **Vil det være hensiktsmessig å la hele tjenesten være dynamisk konfigurerbar?**

Med dynamisk konfigurerbar tjeneste menes da at tjenesten "bygges opp" direkte i kjøretidsmiljøet, ved at allerede ferdige komponenter velges ut. Komponentene igjen kan selvfølgelig også (dersom ønskelig), være konfigurerbare. (Se figur under: ).




**Figur 23, Dynamisk konfigurert tjeneste**

Ved en slik løsning vil det være mulig å lage tjenester som tar hensyn til klientens tilstand, klientens ønsker, og behov. Siden script brukes for konfigurering, må alle invokeringer gå via en script-tolk (*interpreter*). Dette medfører selvsagt en liten forsinkelse i tid, men vi tror ikke denne blir større en at fordelene med økt gjenbruk oppveier for dette.

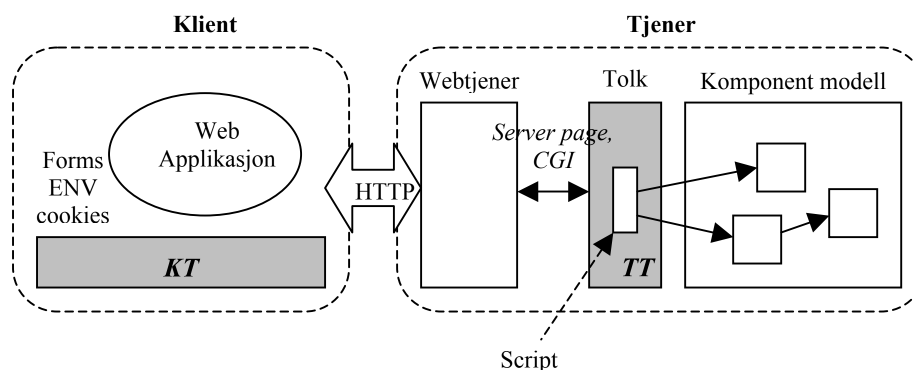
Det er allerede utført en del arbeid innen konfigurering av komponenter (problemstilling 1), eller *Adaptable Software Components*, som det også blir kalt. Spesielt har Dr. George T. Heineman og hans forskningsgruppe publisert artikler angående dette emnet (http://www.cs.wpi.edu/~heineman) [22]. De har utviklet et eget komponent-spesifikasjonsspråk som beskriver komponentenes grensesnitt, samt hvordan disse kan tilpasses nye krav (konfigureres).

## 4.2 Dynamisk konfigurering av komponenter/tjenester (vår idé)

Vi ser for oss tre innfallsvinkler og mulige generelle løsninger, når man skal prøve å utvikle dynamisk konfigurering av komponenter og/eller tjenester:

**Løsningsforslag 1: Web-basert konfigurering**

Gjennom å utvide arkitekturen til også å inneholde en klienttilpasning (KT) og en tjenertilpasning (TT), vil en kunne lage en "generell" løsning. (se figur under:)

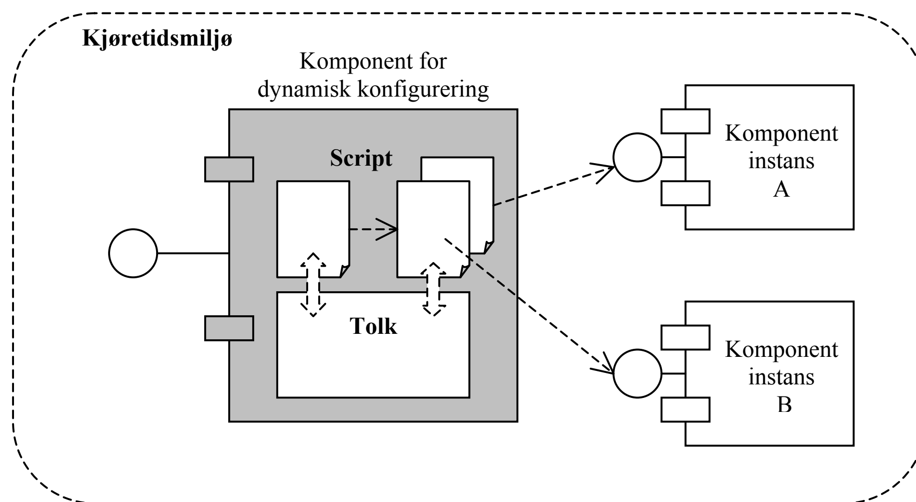



**Figur 24, Webtjener og CGI**

Tilpasningsleddet på klient-side(KT), har som hovedoppgave å samle inn "nyttig" informasjon om klienten og dens omgivelser, for så å sende denne med i tjenesteforespørsler mot tjeneren. Tilpasningsleddet på tjener-side (TT), har som oppgave å gjøre konfigurering av tjeneste eller komponenter, på bakgrunn av informasjon mottatt fra klienten. Denne løsningen vil være uavhengig av komponentenes "rammeverk" (komponentstandard). En slik løsning legger derfor ikke begrensinger i valg av applikasjonsarkitektur, og vil være svært fleksibel med hensyn på anvendelse.

**Løsningsforslag 2: Egen komponent for Konfigurering**

En annen mulighet vi være å realiser den dynamiske konfigureringen som en egen komponent, ved å legge inn både tolk og script i denne komponenten. Konfigureringskomponenten vil da virke som en slags utvelger (*dispatcher)* mot de andre komponentene.
(Se figur under: ).




**Figur 25, Prinsippskisse for egen komponent for dynamisk konfigurering**

Siden en slik komponent vil være en helt vanlig komponent, vil det ikke være nødvendig å modifisere dagen komponentstandarder.

**Løsningsforslag 3: Spesifikk tilpasning av komponentmodellen**

Gjennom å utvide tjenestespekteret som komponentstandarden tilbyr, slik at disse også støtter dynamisk konfigurering. Denne løsningen medfører at dagens komponentstandarder må tilpasses, slik at en eller annen form for konfigureringsscript kan legges inn i tjenerens komponentmodell, sammen med komponentene. Et stort problem med denne løsningen er at dette bare lar seg gjennomføre dersom tjenerleverandørene implementerer en slik oppdatering av applikasjonstjenerens komponentmodell. Siden Sun og Microsoft i dag bruker proprietære komponentstandarder, kan en slik modifisering av tjenestespekteret til komponentmodellen være svært vanskelig å få realisert i praksis.

Siden vi i vår oppgave har valgt å ha fokus på hvordan dynamisk konfigurering kan realiseres (flere forskjellige løsningsscenarios), og dette enklest lar seg implementere og teste basert på prinsippene i løsningsforslag 1, har vi i det videre arbeidet valgt å basere våre praktiske løsninger på løsningsforslag 1.

Til å utføre selve konfigureringen (den dynamiske skreddersømmen) ser vi for oss at bruk av scripts vil være effektivt og gunstig. Det finnes flere ulike scriptspråk å velge imellom som f.eks. Python og Perl, og fordelen er at disse i utgangspunktet er plattformuavhengig. I web-sammenheng vil det være naturlig å se for seg at kommunikasjonen med disse scriptene foregår via web-tjenerens CGI (*Common Gateway Interface*) eller i form av en "*Server Pages*"-implementasjon. Sistnevnte alternativ fungerer på den måten at man kan blande HTML-kode og script-kode i samme dokument, og her kommer også ASP (*Active Server Pages*), JSP (*Java Server Pages*) og PSP (*Python Server Pages*) inn som aktuelle implementasjoner.

Til innsamlingen av informasjon om klienten (brukerprofil), vil teknologier som JavaScript, Cookies og miljø-variabler (*environment variables*) være aktuelle. JavaScript er et scriptspråk som kan brukes på klientensiden enten for å kommunisere med brukeren eller for å trekke ut informasjon om klienten på egenhånd. Cookies er en teknikk for å lagre informasjon om klienten (brukeren), slik at tjeneren kan bruke denne informasjonen ved senere anledninger. Når det gjelder miljø-varibl ene (*environment variables*), så beskriver disse ting som klienttype, operativsystem, adresse-informasjon o.s.v.

## 4.3 Informasjonsinnsamling (klient)

### 4.3.1 Innledning

I dette kapitlet vil vi ta for oss forskjellige aktuelle teknikker for informasjonsinnsamling på klienten, og vi vil starte med å definere begrepet brukerprofil.

### 4.3.2 Brukerprofil

Med begrepet brukerprofil mener vi informasjon som på en eller annen måte beskriver noe om brukeren av tjenesten (klienten). Dette kan være personlige ting om brukeren, som f.eks. navn, alder, yrke, stilling og bruksmønster, eller det kan være mer teknisk informasjon om klienten, som f.eks. klienttype, operativsystem, kjøretidsomgivelser o.s.v. De personlige opplysningene vil brukeren i første omgang bli bedt om å oppgi selv, og i denne prosessen vil bruk av forms og javascript, iallefall i web-sammenheng, være gunstig teknologi. Oppbevaring av denne inntastede informasjonen kan videre enten lagres på tjeneren i f.eks. en database eller i cookies hos klienten selv. Sistnevnte løsning vil nok være den beste, i og med at tjenesten vil skalere bedre. Den mer tekniske informasjonen vedrørende klienten (type, operativsystem, kjøretidsomgivelser o.s.v) kan man f.eks. finne utifra sesjonens miljø-variabler (environment variables). For å inkludere klienter som ikke baserer seg på web-teknologi, vil man måtte utvikle andre former for teknisk informasjonsinnsamling, f.eks. vha av (innsamlings)script på klienten.

### 4.3.3 HTML Forms

HTML Forms [14] er en metode man kan bruke i browseren for å gi brukeren mulighet til å taste inn informasjon og/eller velge blant et eller flere predefinerte valgalternativer. På grunn av at brukeren får anledning til å velge og spesifisere ulike ting (avhengig av tjenesten), brukes forms i stor grad når det gjelder dynamisk konfigurering av tjenester. En form kan inneholde er rekke ulike objekter som f.eks. tekstbokser, listebokser og radiobuttons, og den har i de aller fleste tilfeller også en submit-knapp som brukes for å sende

informasjonen/valgene (forespørselen) til web-tjeneren. Generell HTML-kode for forms er vist og kommentert nedenfor:

```
<FORM NAME="myform" ACTION="/cgi-bin/myscript.cgi" METHOD="POST">
<INPUT TYPE="type" NAME="name" VALUE="value">
...
...
<INPUT TYPE="submit" value="Send">
</FORM>
```

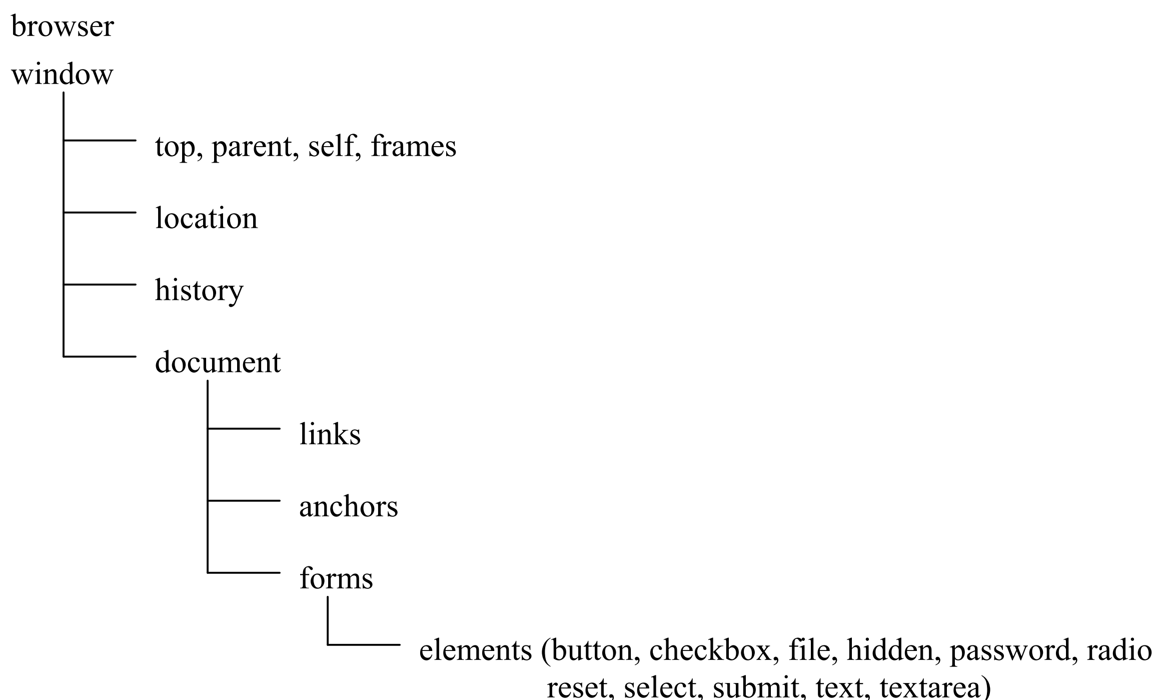
Den første linjen i koden introduserer formen, gir den et navn (NAME) og tilordner et program (ACTION) som skal motta informasjonen fra brukeren når han/hun har trykket på submit-knappen. METHOD-argumentet sier hvordan informasjonen skal oversendes (ved POST sendes url og data hver for seg (anbefalt), mens ved GET sendes url og data sammen).

De neste linjene plasserer ut de forskjellige objektene formen skal bestå av. Man må si hva slags type objekt det skal være (TYPE), gi objektet et navn (NAME) og eventuelt sette en default verdi (VALUE). Det finnes per i dag 11 forskjellige forms-objekter tilgjengelig:

- Button Knapp
- Checkbox Avkrysningsboks
- File Brukes for å velge og laste opp en fil til tjeneren
- Hidden Skjult felt (vises ikke)
- Password Tekstfelt hvor inntastet informasjon blir skjult av *'er
- Radio Avkrysningsboks (flere gruppert sammen, kun en kan være valgt)
- Reset Tilbakestiller formobjektene (sletter inntastet informasjon)
- Select Listeboks (enkelt valg, eller flervalg)
- Submit Knapp som sender form-dataene til tjeneren (spesifisert i ACTION)
- Text Tekstfelt (en linje)
- Textarea Tekstfelt (flere linjer)

### 4.3.4 JavaScript

JavaScript [9],[10] er et ”lightweight” interpretert (tolket) programmeringsspråk med objektorienterte egenskaper. JavaScript har til tross for navnet ingen ting med Java å gjøre, og er heller ikke så objektorientert. Det vil derfor være mer riktig å kalle JavaScript for objektbasert fremfor objektorientert, da språket ikke omfatter ordentlige klasser (abstraksjon) og arv (polymorfi). Når man programmerer JavaScript må man allikevel forholde seg til objekter og hierarkier av objekter [15]. Siden JavaScript først og fremst brukes i forbindelse med web-browsere, har de fleste browserne en JavaScript interpreter (tolk) innebygget, og objekthierarkiet starter med browseren på topp:



### 4.3.5 Cookies

Cookies [23] er en teknikk som kan brukes til å lagre og hente forskjellig informasjon om en klient. Denne informasjonen kan i utgangspunktet være hva som helst, men har som oftest til hensikt å på en eller annen måte beskrive klienten personlig (navn, alder, yrke, stilling, tjenestebehov, bruksmønster o.s.v) og/eller teknisk (klienttype, operativsystem, kjøretidsomgivelser o.s.v). Det kan være lett å anta at slik informasjonen ligger lagret på tjeneren, men det er ikke tilfelle. Informasjonen ligger lagret lokalt hos den enkelte klient, i en fil som heter cookies.txt. Grunnen til dette er at tjenestene ville skalere dårlig dersom mye slik informasjon ble lagret på tjeneren (det ville ta for mye plass, og bli tregt å gjøre oppslag i). Det er selvfølgelig tjeneren (tjenesten) som bestemmer hva slags informasjon den ønsker å lagre og lese, men det er som nevnt altså klienten selv som oppbevarer informasjonen. Cookies-teknologien er også laget slik at en klient har mulighet til å reservere seg mot slik aktivitet. En tjener kan lagre informasjon hos en klient (sette en cookie) ved å inkludere en Set-Cookie linje i HTTP-headeren på HTML-dokumentet som sendes til klienten, og kan ved senere anledninger hente ut informasjonen igjen via environment-variabelen HTTP_COOKIE i klientens HTTP-request.

Formatet på en cookie er som følgende:

```
NAME=VALUE;NAME=VALUE...;expires=DATE;path=PATH;domain=DOMAIN;secure
```

- `NAME=VALUE` Sett av variabel-navn og tilhørende verdi. Kan ha flere slike par.
- `expires` Angir levetid til cookien. Format: `Wdy, DD-Mon-YYYY HH:MM:SS GMT`
- `domain` Angir hvilken tjener denne cookien tilhører
- `path` Angir hvilke dokumenter (i hvilken katalog) denne cookien tilhører
- `secure` Hvis satt, sendes bare cookien over sikre (HTTPS) forbindelser.

### 4.3.6 Miljø-variabler (environment variables)

Miljø-variabler (environment variables) i web-sammenheng er et sett med variabler hvis verdier beskriver diverse teknisk informasjon om klienten, tjeneren og forbindelsen. Miljøvariablene blir satt for hver forespørsel til tjeneren, og er automatisk tilgjengelig for eventuelle CGI-scripts som måtte befinne seg på tjeneren.

Nedenfor vises et utdrag av tilgjengelige miljø-variabler som beskriver klienten:

| | |
|---|---|
| REMOTE_USER | Klientens brukernavn |
| REMOTE_ADDR | Klientens IP-adresse |
| REMOTE_HOST | Klientens maskinnavn (domene) |
| HTTP_USER_AGENT | Klient-type (browser) |
| QUERY_STRING | Parametre innbakt i url'en |
| HTTP_REFERER | Beskriver hvilken side forespørselen kommer fra (url'en) |
| HTTP_COOKIE | Klientens (brukerens) eventuelle cookie |

**Tabell 1, miljø-varibler**

## 4.4 Konfigurering (tjener)

### 4.4.1 Innledning

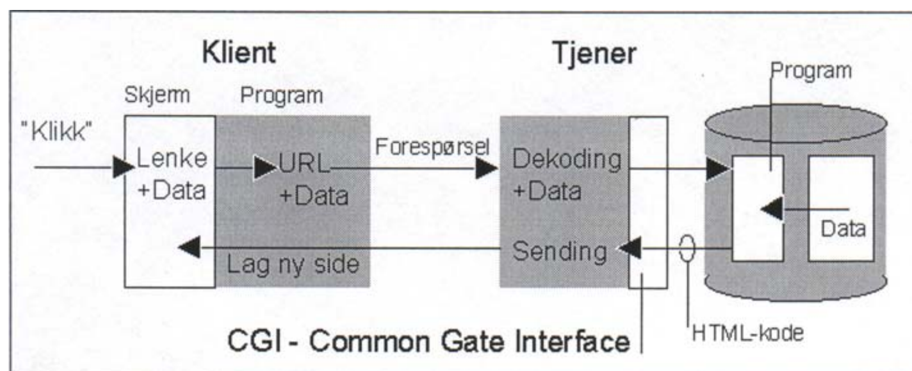
Som nevnt i innledningen til kapittel 4 (4.1), så mener vi at bruk av scriptteknologi er det mest hensiktsmessige i forbindelse med den direkte dynamiske konfigureringen (skreddersømmen) av komponenter og tjenester. Script er enkle å oppdatere, muligheten for klient-tjener-script kommunikasjon over Internett/intranett er allerede definert gjennom CGI eller ved en "Server Pages"-løsning, og mange av de mest brukte scriptspråkene har en moderat læreterskel. Det finnes nok flere aktuelle og anvendbare scriptspråk, men til denne type konfigurering vil vi generelt foreslå å bruke enten Python eller Perl. Begge språkene er plattformuavhengige, hvilket innebærer at man kan bruke samme script (kode) på flere forskjellige tjener-plattformer. Disse språkene er dessuten vel etablert i programmeringsmiljøer over hele verden, så det eksisterer derfor en global interesse for utvikling og implementasjon i og av språkene.

### 4.4.2 Common Gateway Interface (CGI)

Common Gateway Interface (CGI) [24] er et standard grensesnitt for kommunikasjon mellom en web-tjener og resten av maskinen. I et litt større perspektiv kan man si at dette grensesnittet åpner for avansert kommunikasjon mellom en web-klient (*browser*) og en maskin som kjører en web-tjener. Common Gateway Interface (CGI) er et grensesnitt som de aller fleste web-tjenere har innebygd, så man behøver bare å "slå det på" (på tjeneren). I utgangspunktet kan dette grensesnittet åpne for full tilgang til resten av maskinen, men dette styres selvfølgelig av brukerrettigheter og annen form for tilgangskontroll. CGI kan med andre ord være et sikkerhetshull, men ikke hvis det konfigureres riktig.

Hensikten med å bruke Common Gateway Interface (CGI) er for å få tilgang til informasjon lagret andre steder på maskinen (f.eks. i databaser), og for å kunne generere dynamiske web-sider basert på valg eller informasjon fra klienten. Som nevnt ovenfor er CGI som oftest ferdig implementert i web-tjeneren. Det eneste man behøver å gjøre som programmerer er derfor å lage programmene (scriptene) som skal kunne ta imot og behandle valg/informasjon fra klienten, og trekke ut og sette sammen ønsket informasjon for så å presentere denne for

klienten (generere html-kode). Disse såkalte CGI-scriptene kan i utgangspunktet være skrevet i et hvilket som helst programmeringsspråk så lenge de er i stand til å motta input (STDIN) og generere output (STDOUT). For å få scriptene til å fungere må disse ofte plasseres i en egen katalog på web-tjener maskinen (gjerne kalt cgi-bin), og klientene kaller scriptene direkte derfra via HTTP-protokollen (f.eks. http://www.mydomain.com/cgi-bin/myscript.cgi). Et standard CGI-kall kan beskrives ved hjelp av figuren nedenfor:



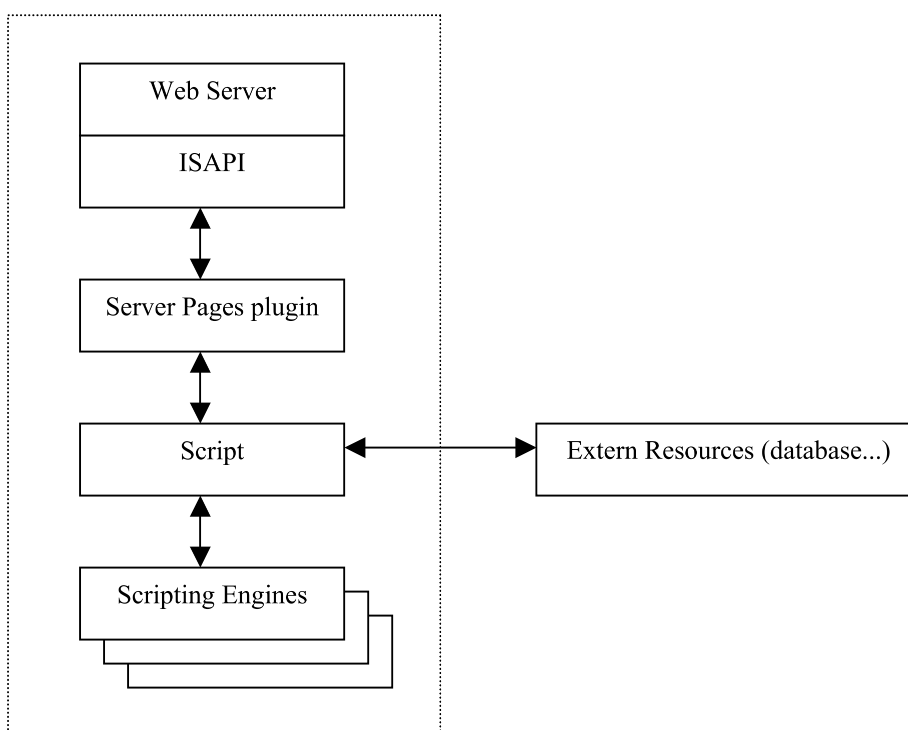

**Figur 26, CGI - Common Gate Interface [15]**

Brukeren trykker en link (eller form-button) i browseren (klienten) og sender dermed en forespørsel med tilhørende informasjon (query-string eller form-data) via CGI-grensesnittet på web-tjeneren til et cgi-script (program). Cgi-scriptet analyserer den mottatte informasjonen, foretar sine logiske valg, henter inn ønsket ekstern informasjon (f.eks. fra en database) og leverer (printer) HTML-kode til web-tjeneren. Deretter sender web-tjeneren HTML-koden tilbake til klienten (browseren) som viser informasjonen til brukeren.

### 4.4.3 Server Pages

Server Pages teknologien [25] er en annen løsning man kan bruke for å implementere dynamisk konfigurering. Som ved CGI, så er også hensikten med å bruke en Server Pages teknologi det å få tilgang til informasjon lagret andre steder på maskinen (f.eks. i databaser), og for å kunne generere dynamiske web-sider basert på valg eller informasjon fra klienten. Det finnes en rekke ulike Server Pages teknologier å velge mellom, som f.eks. Active Server Pages (ASP), Java Server Pages (JSP) og Python Server Pages (PSP). Det som er likt for alle teknologiene, og noe av det som også skiller løsningen fra det å bruke cgi-scripts, er at man kan blande HTML-kode og script-kode i samme dokument. Som ved cgi-scripts, så er det en

tolk (*interpreter*) som tolker og utfører scriptkoden, men i Server Pages teknologien er denne tolken mer direkte knyttet til webtjeneren (som en slags plugin), enn tilfellet er ved bruk av cgi-scripts. Grensesnittet mellom webtjeneren og tolken kalles her for ISAPI (*Internet Server Application Programming Interface*). ISAPI er i utgangspunktet definert av Microsoft, men dette grensesnittet implementeres også på andre plattformer/webtjenere enn Microsoft sine. Siden tolken er så nært integrert med webtjeneren (kjører i samme prosess) og tolken kun kobles inn ved behov, er denne løsningen ofte tidsmessig mer effektiv enn ved bruk av CGI og ekstern tolk. Allikevel kan det være begrensninger i de tilgjengelige scriptspråkene i Server Pages teknologien, som gjør at man heller vil velge en cgi-løsning. Det som ofte er tilfelle, er at man har anledning til å velge mellom flere forskjellige scriptspråk innenfor en og samme Server Pages teknologi. Derfor inneholder løsningene ofte flere såkalte "*Scripting Engines*", som tar seg av hvert sitt scriptspråk. Nedenfor vises en skisse som setter sammen de forskjellige begrepene vi har snakket om i dette avsnittet.




**Figur 27, Server Pages teknologi**

## Active Server Pages

Active Server Pages (ASP) [7],[26] er utviklet av Microsoft, og er nå ute i versjon 3. Ved installering av Microsoft sin webtjener Internet Information Server (IIS) installeres og konfigureres ASP automatisk. For andre plattformer/webtjenere må ASP-støtte som regel installeres separat. ASP installeres default med to komplette ”scripting engines”: VBScript og JScript. VBScript bruker et subsett av programmeringsspråket Visual Basic, mens JScript bruker programmeringsspråket JavaScript. Flere ”scripting engines” er også tilgjengelige, slik som TCL og PerlScript. Det mest utbredte i forbindelse med ASP-programmering er allikevel VBScript. Grunnen til dette er nok at Visual Basic er et ganske intuitivt og lettfattelige programmeringsspråk, også for personer som ikke har så mye programmeringserfaring. ASP er derfor etterhvert blitt en meget utbredt teknologi blant webdesignere. Som nevnt tidligere, så kan man blande HTML-kode og script-kode i samme dokument. Filen må ha endelsen .asp, som er et signal til webtjeneren om at filen kan inneholde script-kode. Øverst i filen må man gi beskjed om hvilket scriptspråk (scripting engine) man bruker, og dette gjøres ved følgende setning: <%@LANGUAGE=”language_name”%>, hvor language_name for eksempel kan være VBScript. All kode som plasseres innenfor <%....%> blokker vil nå bli sendt til den angitte ”scripting engine”. For å håndtere henvendelser fra klienter definerer også ASP et sett med objekter og operasjoner man kan benytte seg av. De mest brukte objektene er beskrevet nedenfor:

| Request | Via dette objektet får man tak i all informasjon som klienten har oppgitt i forbindelse med henvendelsen (formdata, querystring, cookies etc) |
|---|---|
| Response | Dette objektet bruker man i forbindelse med responsen som skal sendes tilbake til klienten. Objektet tilbyr f.eks. metoder for output |
| Session | Session objektet opprettes unikt for hver bruker, og lever så lenge sesjonen pågår. Inneholder mekanismer for å bevare ønsket data gjennom hele sesjonen. |
| Server | Objektet brukes i forbindelse med ekstern informasjonsinnhenting, f.eks. fra databaser. |

**Tabell 2, ASP objekter**

**Java Server Pages**

Java Server Pages (JSP) [27],[28],[29],[30] bygger på det samme prinsippet som Active Server Pages (ASP), d.v.s at man kan blande HTML-kode og script-kode i samme dokument, for å oppnå dynamisk innhold. Forskjellen er at JSP bruker Java som programmeringsspråk (default), og at koden kompileres til servlets (automatisk) før kjøring. For å få JSP til å fungere, må derfor webtjeneren støtte servlets. Det er få webtjenere som kommer ferdig konfigurert med støtte for servlets, men det finnes i de fleste tilfeller såkalte plugins eller støtteprogrammer man kan installere i etterkant, for å legge til denne funksjonaliteten. I vårt oppsett har vi brukt webtjeneren Internet Information Server fra Microsoft, samt programmet JRun som gir IIS mulighet til å kjøre JSP og servlets (mer om JRun i kapittel 5). På grunn av at scriptkoden (Java) må kompileres til servlets før kjøring, kan JSP i visse tilfeller oppleves som noe tregere enn ASP. JSP (Java) har allikevel fordeler fremfor ASP med hensyn til funksjonalitet og portabilitet. JSP er utviklet av Sun Microsystems.

**Python Server Pages**

Python Server Pages (PSP) [31] er en server scripting teknologi på lik linje med Active Server Pages (ASP) og Java Server Pages (JSP). Den største forskjellen mellom ASP og PSP, er at PSP er skrevet 100% i Java, og derfor er lang mer portabel. Programmeringsspråket (scriptspråket) som brukes i PSP heter Jython. Jython er en Java-implementering av programmeringsspråket Python, som gir brukeren (programmereren) anledning til å bruke både Python-kode og Java-kode i samme program (script), noe som har vist seg å være en meget vellykket kombinasjon (mer om Jython og Python i neste delkapittel). I og med at både PSP og Jython er skrevet i Java, krever PSP (som JSP) at webtjeneren har installert støtte for servlets. En komplett installasjonsveiledning for Python Server Pages, samt nedlasting, finnes på sidene til CIOBriefings LLC (http://www.ciobriefings.com/psp).

### 4.4.4 Scriptspråk

Et scriptspråk kjennetegnes gjerne som et interpretert (tolket) programmeringsspråk som ofte brukes til å utføre mindre, rutinemessige oppgaver. Den første delen av denne beskrivelsen er en mer eller mindre felles oppfattet definisjon, mens bruksområdet nok kan omfavne litt mer

en nevnt her. Det at et programmeringsspråk er interpretert (tolket) betyr at det ikke forgår noen kompilering av kildekoden. Den tolkes og utføres linje for linje av en tolk (interpreter). Hvilken tolk (interpreter) som skal tolke og utføre koden, bestemmes enten av filens endelse (type), p.g.a mapping, eller utifra en referanse til tolken øverst i kode-filen. En slik referanse kan f.eks. se ut som følgende: `!#/usr/bin/perl` , som i dette tilfellet betyr at koden skal tolkes av Perl.

**Perl**

Perl (Practical Extraction and Report Language) [12],[32] er kanskje det mest utbredte scriptspråket i verden idag, og utviklingen av dette språket ble startet allerede i 1986 av Larry Wall. Siden den gang har språket utviklet seg mye, og interessen for språket er fremdeles stor både blant systemadministratore og web-utviklere. Som navnet også sier noe om, så er språket spesielt beregnet på ekstrahering og mønstergjenkjenning av data i f.eks. filer, databaser eller annen form for input, og er derfor vel egnet for rapportering og konfigurering. Disse kjerneegenskapene er fremdeles bevart, selv om det nå også finnes moduler for å gjøre alle mulig andre ting. Andre typiske egenskaper ved Perl er at det er et tolket, plattformuavhengig språk som utvikles Open Source (fri kildekode, gratis)

**Python**

Python [11],[33] er et interpretert (tolket), objektorientert programmeringsspråk med dynamisk semantikk. Språkets høynivåstrukturer, kombinert med dynamisk typing og binding, gjør det til et attraktivt språk for applikasjonsutvikling, så vel som til scripting og for å lime eksisterende komponenter sammen. Pythons pseudokode-lignende syntaks gjør også at dette er et enkelt og lettfattelig programmeringsspråk å bruke. Utviklingen av Python startet i 1990 ved CWI (Centrum voor Wiskunde en Informatica) i Amsterdam, and fortsetter idag ved CNRI (The Corporation for National Research Initiatives) i Reston, Virginia, USA. Internett og Intranett er sannsynligvis det mest populære utviklingsområdet for Python for tiden, men språket brukes også, i likhet med Perl, til mer systemadministrative oppgaver og applikasjoner. Python er også plattformuavhengig, og utvikles Open Source

**Jython**

Jython [35] er en implementasjon av programmeringsspråket Python, skrevet 100% i Java. Jython er sømløst integrert med Java plattformen, hvilket innebærer at man kan kjøre Python på en hvilken som helst Java plattform. Den kanskje største fordelen er at man fra sitt Jython-program har full tilgang til alt av Java-klasser og Java-objekter som måtte finnes på systemet, slik at man faktisk er istand til å programmere Python og Java om hverandre, i samme applikasjon. På denne måten blir man altså i stand til å benytte seg av det beste fra hvert språk, hvilket må sies å være meget fordelaktig. På grunn av den tette tilknytningen til Java, regner vi med at Jython (Python) er et bra alternativ i forbindelse med konfigurering av java-komponenter, og har derfor tenkt å se på mulighetene for å benytte Jython (Python) som et scriptspråk for å konfigurere enkeltkomponenter såvell som hele tjenester i vårt videre arbeid. Jython er i skrivende stund ute i versjon 2.0, og videreutvikles derfra som et Open Source prosjekt hos SourceForge (http://sourceforge.net)

**Java**

Språket Java fra Sun Microsystems [36] er ikke noe man generelt vil karakterisere som et scriptspråk, men siden både JSP-teknologien og Jython (beskrevet ovenfor) bruker Java som sitt programmeringsspråk, vil vi allikevel si noen ord om dette. Java er et generelt, objektorientert høynivåspråk på lik linje med f.eks. C++, som kan brukes til å utvikle både applikasjoner, applets og servlets. Det som først og fremst skiller Java fra språk som C++, er at kildekoden kompileres til et format (bytekode) som kan kjøres på en virtuell maskin (JVM, Java Virtual Machine) istedenfor på en spesifikk maskinplattform. Siden det finnes JVM'er tilgjengelig for de fleste maskinplattformer, har Java et stort fortrinn når det gjelder portabilitet.

**JScript / JavaScript**

JScript, eller JavaScript [9],[10],[37] som det også kalles, har lite til felles med Java. Navnene er derfor nokså villedende, og selv om JScript og JavaScript brukes om hverandre er det allikevel ikke helt det samme. Begge språkene bygger på spesifikasjonsspråket ECMAScript, som foreløpig er det eneste standard scriptspråket for web. ECMAScript brukes i liten

utstrekning i seg selv (bortsett fra i browseren Opera), men både Netscape og Microsoft følger denne standarden i sine implementasjoner av henholdsvis JavaScript og JScript. Forskjellene mellom JavaScript og JScript er derimot minimale, og det er nok grunnen til at begrepene ofte blandes. JScript kan f.eks. brukes i forbindelse med ASP (Active Server Pages), selv om det nok er mer vanlig å bruke VBScript.

**VBScript**

VBScript (Visual Basic Scripting Edition) [38] er et subsett av språket Visual Basic fra Microsoft. Visual Basic er et fullverdig fjerdegenerasjons programmeringsspråk, mens VBScript er et raskt, portabelt, lightweight interpretert (tolket) språk til bruk i websammenheng. Active Server Pages (ASP) teknologien bruker VBScript som standard scriptspråk, og har derfor etterhvert fått en ganske stor utbredelse. En fordel med dette språket er at det, etter vår oppfatning, er meget intuitivt og krever liten programmeringserfaring for å lese/skrive.

**CorbaScript**

CorbaScript [39] er et interpretert (tolket) objektorientert scriptspråk dedikert til CORBA miljøer. Et CorbaScript kan invokere en hvilken som helst operasjon og få tak i eller sette alle tilgjengelige attributter i et hvert Corba objekt gjennom Corba DII (Dynamic Invocation Interface). Videre kan et hvert OMG IDL grensesnitt implementeres i CorbaScript, hvilket betyr at man ikke trenger å generere hverken stubs eller skeletons eksplisitt. OMG IDL beskrivelsene kan ekstraheres fra ”Interface Repository” og gjøres direkte tilgjengelig for scriptene. CorbaScript ligner både syntaksmessig og funksjonelt en del på java. Det er stor fokus på objekter og kildekoden kompileres til en slags pseudokode som kjøres av en virtuell maskin (Virtual Object Oriented Machine). CorbaScript er et akademisk prosjekt tilgjengelig i full kildekode, og er gratis for en hver bruk.

# 5 Praktiske løsninger (eksperimentell del)

## 5.1 Innledning

Dette kapitelet beskriver praktiske løsninger for dynamisk konfigurering med hensyn på komponentbaserte tjenester. Vi har satt sammen og testet ulike teknologier fra de to foregående kapitlene (komponentstandarder og dynamisk konfigurering), hvorpå vi har kommet frem til såkalte løsningsscenarios som beskriver prinsipper for oppsett, arkitektur og virkemåte. Løsningsscenarioene beskriver også metoder for innsamling av klientdata og aksessering/konfigurering av komponenter.

## 5.2 Avgrensninger

I den eksperimentelle delen har vi valgt å ha fokus på JavaBeans og COM som komponentstandarder. Videre valgte vi å teste ut løsninger basert på webtjeneren Internet Information Server fra Microsoft under Windows 2000. Grunnen til disse avgrensningene er først og fremst tilgjengelighet. Vi mener imidlertid at valg av plattform og webtjener som vi bruker i det eksperimentelle arbeidet er underordnet. Felles for scenarioene (5.4) er at de ikke inneholder kodelogikk (*if-then-else*), da kodelogikken vil variere kraftig fra tjeneste til tjeneste.

## 5.3 Systembeskrivelse og Konfigurering (tjener)

Det praktiske arbeidet (analyse, test og implementasjon) i forbindelse med dette prosjektet har blitt utført på følgende plattform og programvare:

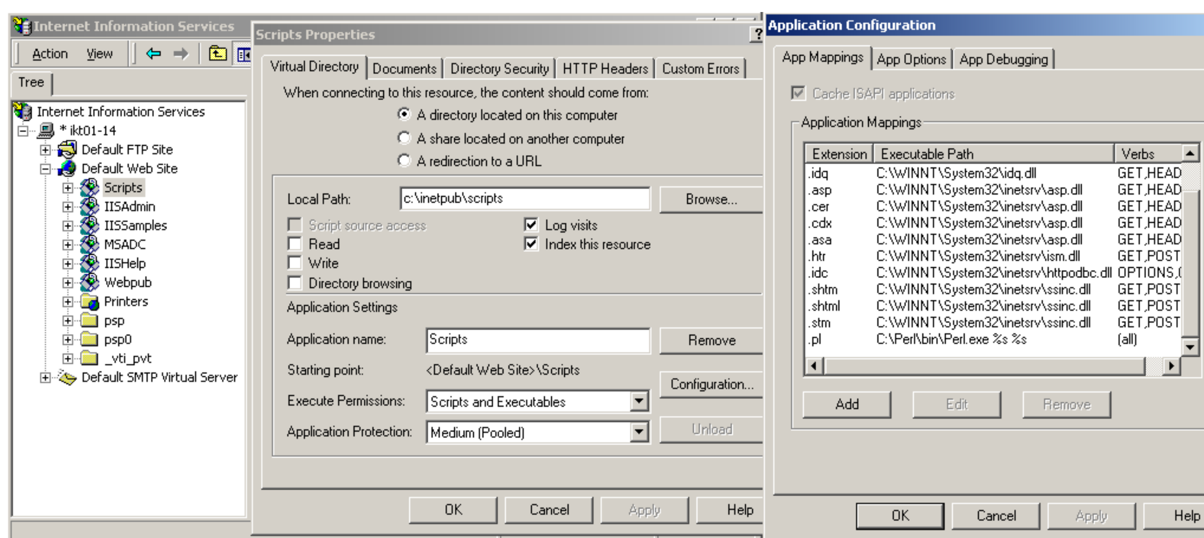
| | |
|---|---|
| Hardware: | Pentium II 266Mhz med 128Mb Ram |
| Operativsystem: | Microsoft Windows 2000 Professional (5.00.2195 Service Pack 1) |
| Web-server: | Microsoft Internet Information Services 5.0 |
| Programvare: | Java 2 SDK, Standard Edition 1.3.0_02 |
| | Perl 5.7.1 med JPL-modul |
| | JRun Server 3.0, Developer Edition |
| | Python Server Pages 1.0.3 |
| | Jython 2.0 |
| | Perl Dev Kit 2.1 (PerlCOM) |

### 5.3.1 Microsoft Internet Information Services 5.0

Siden vi til daglig jobber i et Microsoft miljø, med Microsoft Windows 2000 som plattform, ble vårt naturlige valg av webtjener Microsoft Internet Information Server (IIS). Microsoft Internet Information Server er en del av Microsoft Internet Information Services 5.0, som også inneholder tjenerprogrammer for ftp, epost osv. Installering og oppsett av programmet var stort sett rett frem (følge henvisningene i installasjonsprogrammet). I ettertid måtte vi allikevel konfigurere enkelte innstillinger manuelt, ettersom vi trengte å knytte til ekstra funksjonalitet og programmer (beskrevet individuelt nedenfor). Microsoft Internet Information Server (IIS) har innebygget støtte for Active Server Pages (ASP).

## 5.3.2 Perl 5.7.1 med JPL-modul

Vi ønsket å gjøre et eksperiment med IIS, CGI, Perl og JPL mot JavaBeans (scenario 5.4.1), og trengte derfor å laste ned, kompilere og installere Perl og JPL-modulen [41] for Windows. JPL-modulen er inkludert i Perl 5.7.1, og muliggjør at man kan importere og bruke Java-klasser i sitt Perl-script (forutsatt at man har installert Java). Perl, inkludert JPL-modulen, kan lastes ned fra http://www.perl.com. For å få webtjeneren (IIS) til å kjøre Perl-scripts (gjennom CGI), måtte vi sette opp en "application-mapping" i Internet Information Services, slik at alle henvendelser angående Perl-filer (.pl) blir behandlet av Perl-tolken.




**Figur 28, Konfigurering av Internet Information Services**

## 5.3.3 Jython 2.0

På samme måte som beskrevet ovenfor for Perl og JPL, ønsket vi å teste ut kommunikasjon mot JavaBeans vha IIS, CGI, og Jython (scenario 5.4.2). Jython er et scriptspråk som inkluderer språkene Python (lignende Perl) og Java, og Jython er avhengig av et Java-miljø (Java Runtime Environment) for å kjøre. Nedlastingen og installasjonen av Jython gikk helt fint og det fungerte greit å kjøre Jython-script i shellet, men vi hadde problemer med å få webtjeneren (IIS) til å kommunisere med Jython. Vi prøvde, på tilsvarende måte som beskrevet ovenfor for Perl, å lage en "application-mapping" i IIS mellom Python-filer (.py) og Jython-tolken, men vi fikk bare en rekke "internal server errors" ved kjøring. Feilmeldingene var ikke så veldig informative, men det kunne se ut som om det hadde noe med

rettigheter/sikkerhet å gjøre. Vi prøvde derfor å sette fulle rettigheter til alle brukere på Jython-tolken, men det gav dessverre ingen merkbar effekt. Videre prøvde vi også å hardkode mappingen direkte i systemregisteret (Windows Registry), men det hadde heller ingen effekt. Vi tror mest sannsynlig at dette problemet ikke direkte har med Jython å gjøre, men at det er mer relatert til Java-miljøet på maskinen (som kjører Jython). Det er mulig at den virtuelle java-maskinen (JVM, Java Virtual Machine) er avhengig av enkelte systeminnstillinger og kjøremiljø (paths osv), som ikke er tilstede når forespørselen går via webtjeneren (IIS). Det kan også ha med rettigheter/sikkerhet knyttet til Java å gjøre.

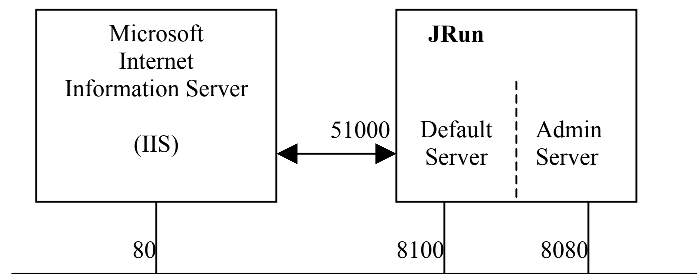
### 5.3.4 Java 2 SDK, Standard Edition 1.3.0_02

Siden Jython er avhengig av et Java-miljø (Jython er en Python/Java-tolk skrevet i Java), måtte vi også installere Java. Vi valgte å installere Java 2 SDK (Software Development Kit) standard edition 1.3.0_02, som i tillegg til Java-miljøet (Java Runtime Environment) også inneholder verktøy for kompilering (javac), dokumentering (javadoc) osv.

### 5.3.5 JRun Server 3.0, Developer Edition

For å kunne ta i bruk teknologiene PSP (Python Server Pages), JSP (Java Server Pages) og Servlets (scenario 5.4.3 – 5.4.5), som er avhengig av at webtjeneren (IIS) har støtte for java-servlets, installerte vi programmet JRun Server 3.0 [42], Developer Edition. JRun Server er en frittstående servlet-tjener (med innebygget støtte for Java Server Pages), men den kan enkelt knyttes opp mot eksterne webtjenere og dermed tilføre disse servlet-funksjonalitet. JRun består egentlig av to tjenere: default-server og admin-server. Administrasjonstjeneren (admin-server) tar seg av oppsett og konfigurering, mens applikasjonstjeneren (default server) inneholder og kjører selve servlet-applikasjonene. Begge tjenerne har egne webgrensesnitt, men applikasjonstjeneren (default server) kan også opprette grensesnitt mot eksterne webtjenere. I vårt tilfelle ønsket vi å bruke Microsoft Internet Information Server (IIS) som webtjener, og satte derfor opp et grensesnitt mellom IIS og JRun's applikasjonstjener. Videre måtte vi angi hvilken katalog under webtjeneren (IIS) som skulle inneholde script, og lage en mapping av filtypene vi ønsket at JRun skulle betjene for IIS (.jsp og .psp). All konfigurering foregikk i JRun's administrasjonstjener. En sammenheng mellom Microsoft Internet

Information Server (IIS) og JRun Server er illustrert i figuren nedenfor (tallene representerer portnummer).




**Figur 29, Microsoft Internet Information Server og JRun Server**

## 5.3.6 Python Server Pages 1.0.3

Python Server Pages (PSP) er en server pages teknologi med Jython som scriptspråk. Jython er en java-implementasjon av språket Python, og gir brukeren (programmereren) anledning til å benytte språkene Python og Java om hverandre. Python Server Pages krever at webtjeneren støtter servlets, samt at språket Jython allerede er installert på maskinen. Python Server Pages distribueres som en JAR-fil (Java Archive) og installeres med følgende kommando:

```
jython -jar psp103.jar
```

Videre må den installerte servleten (Python Server Pages er en servlet) registreres i servlet-tjeneren (JRun Server), og det må opprettes en filmapping som sier at psp-filer skal behandles av Python Server Pages.

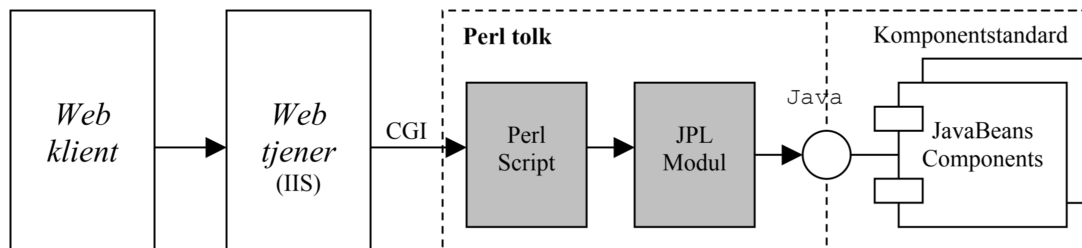
## 5.3.7 Perl Dev Kit 2.1 (PerlCOM)

Perl Dev Kit [40] inneholder en rekke nyttige verktøyer for Perl-programmerere på Windows-plattformen. Ved å installere denne pakken får man også installert PerlCOM, som er en Perl-interpreter (tolk) innekapslet i en COM-komponent. PerlCOM inngår da som en integrert del av komponentstandarden (operativsystemet) til Microsoft. PerlCOM brukes i scenario 5.4.7.

## 5.4 Scenarios

### 5.4.1 IIS, CGI, Perl, JPL, JavaBeans

Dette scenarioet beskriver en metode for dynamisk konfigurering av JavaBeans komponentbaserte nett-tjenester, med Perl som scriptspråk. Forespørselen fra klienten (browseren) går direkte til Perl-scriptet (via webtjenerens CGI-grensesnitt), som har hovedansvaret for konfigureringen. Perl-scriptet baserer konfigureringen på data (formdata, miljøvariabler og cookies) som mottas fra klienten (browseren). For å kunne kalle opp, konfigurere og bruke JavaBeans komponenter, må Perl-scriptet importere modulen JPL. Sammenhengen vises i figuren nedenfor:




**Figur 30, Prinsippskisse: IIS, CGI, Perl, JPL, JavaBeans**

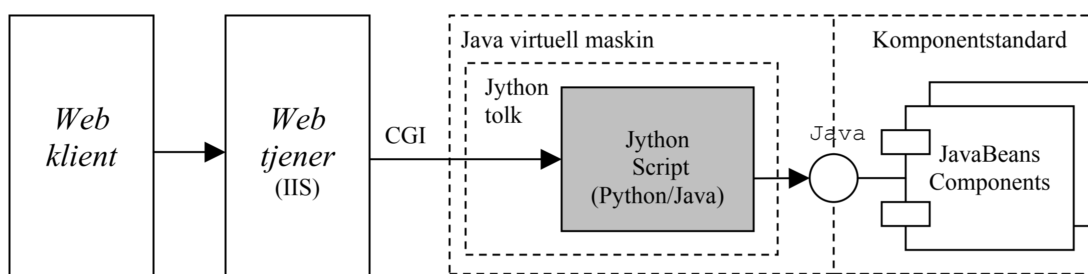
Perl-scriptet vil automatisk motta dataene (formdata, miljøvariabler og cookies) fra klienten (browseren), såfremt scriptet starter med å lese fra standard input (`$in = <>;`). Verdiene til miljøvariablene (environment variables) finnes i hash-tabellen `%ENV` (automatisk), og de enkelte variablene kan aksesseres ved `$ENV{NAME}`, hvor `NAME` er navnet på miljøvariabelen. Innholdet av eventuelle cookies finnes i `$ENV{HTTP_COOKIE}`. Formdataene ligger lagret etter hverandre som en lang tekststreng i variabelen `$in`, og må formateres før bruk. Dette kan gjøres med funksjonen ReadParse (`&ReadParse($in)`), som genererer en hash-tabell (`%in`) der hvert element inneholder eksakt input fra et formobjekt. Har HTML-siden f.eks. en tekstboks som heter Fornavn, kan inputen i dette feltet aksesseres ved variabelen `$in{Fornavn}`.

For å aksessere/konfigurere Javaklasser og JavaBeans komponenter, må disse importeres (`use JPL::Class "MyClass";`), og man må lage interne referanser til alle funksjonene man ønsker å bruke (`$new = getmeth("new", [argtype1], [argtype2], ....);`). Følgende

linje oppretter en instans av klassen: `$MyClass = MyClass->$new(arglist);`. Når man videre ønsker å kalle klassens funksjoner, gjøres dette slik: `$MyClass->$MyFunc(arglist);` (man må først lage interne referanser til alle funksjonene).

### 5.4.2 IIS, CGI, Jython, JavaBeans

Dette scenarioet beskriver en metode for dynamisk konfigurering av JavaBeans komponentbaserte nett-tjenester, med Jython (Python og Java) som scriptspråk. Forespørselen fra klienten (browseren) går som i forrige scenario direkte til scriptet (via webtjenerens CGI-grensesnitt), og scriptet er hovedansvarlig for konfigureringen. Jython-tolken som utfører scriptet er implementert i Java, og lever derfor innenfor en Java virtuell maskin. Siden Jython takler Java-kode (i tillegg til Python-kode), får man her en direkte åpning mot JavaBeans komponentene. Sammenhengen mellom de forskjellige elementene vises i figuren nedenfor:

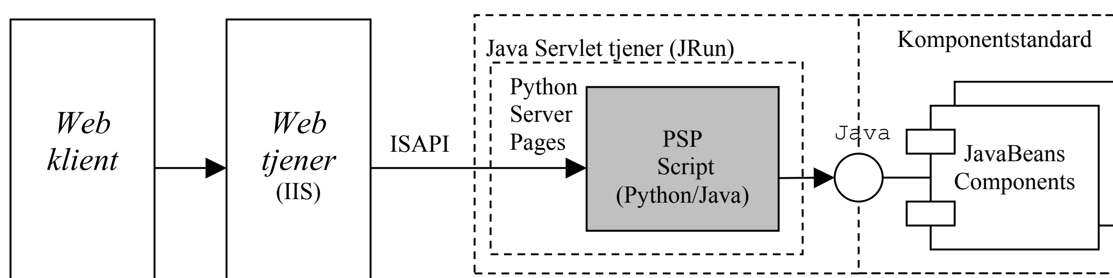



**Figur 31, Prinsippskisse: IIS, CGI, Jython, JavaBeans**

Som i forrige scenario, vil scriptet basere mye av konfigureringen på mottatt data fra klienten (formdata, miljøvariabler og cookies). Formdataene skaffes til veie med følgende kode: `import form = cgi.FieldStorage()`, og verdiene til de enkelte form-objektene aksesseres ved `form[name].value`, hvor `name` er navnet på ønsket form-objekt. Miljø-variablene (environment variables) importeres og aksesseres omtrent på samme måte, henholdsvis: `import env = os.environ()` og `env[name]`, hvor `name` er navnet på en enkelt miljøvariabel. Eventuelle cookies finnes også innenfor miljøvariablene, mer konkret i `env[HTTP_COOKIE]`.

## 5.4.3 IIS, JRun, PSP, JavaBeans

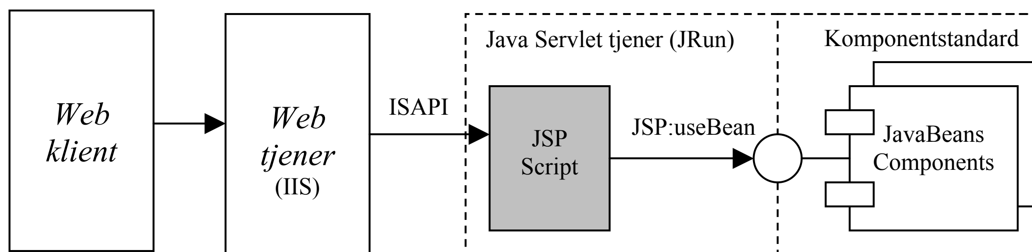
I dette scenarioet har vi realisert konfigurering av JavaBeans komponentbaserte nett-tjenester ved hjelp av Python Server Pages (PSP) teknologien. Python Server Pages er i virkeligheten en Java-servlet, og må derfor kjøre innenfor en servlet-tjener (JRun). Grensesnittet som brukes mellom webtjeneren (IIS) og servlet-tjeneren (JRun) er ISAPI (Internet Server Application Programming Interface). Som i de to forrige scenarioene, er det scriptet (her PSP-script) som er ansvarlig for konfigureringen, og forespørselen fra klienten (browseren) går derfor til scriptet. Python Server Pages (PSP) bruker også Jython (Python/Java) som scriptspråk, så metodene for aksessering av klientdata og JavaBeans blir på samme måte som beskrevet i scenario 3.4.2. En oversikt over systemet vises i figuren nedenfor:




**Figur 32, Prinsippskisse: IIS, JRun, PSP, JavaBeans**

## 5.4.4 IIS, JRun, JSP, JavaBeans

I dette scenarioet er konfigureringen av JavaBeans komponentbaserte nett-tjenester realisert ved hjelp av Java Server Pages (JSP). Web-tjeneren (IIS) har ikke innebygget støtte for Java Server Pages, så vi må derfor knytte til et eksternt program (JRun) som kan håndtere JSP-script på vegne av webtjeneren (IIS). Grensesnittet som brukes mellom webtjeneren (IIS) og servlet-tjeneren (JRun) er ISAPI (Internet Server Application Programming Interface). Språket som benyttes til scriptingen er Java. I tillegg til Java-språket kan man benytte seg av predefinerte objekter (tilsvarende som for Active Server Pages) og såkalte JSP-tags for å få utført de mest vanlige operasjonene knyttet til aksessering og konfigurering av klientdata (formdata, miljøvariabler og cookies) og JavaBeans.

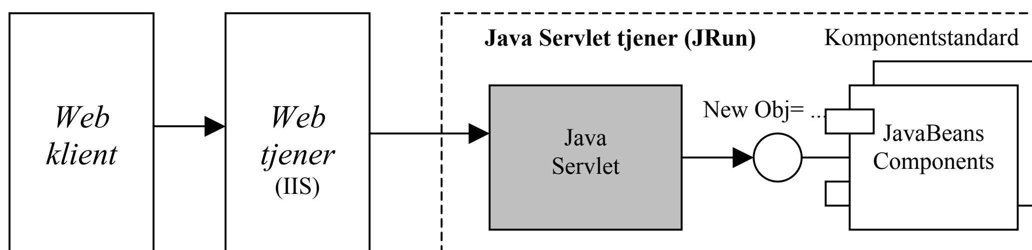



**Figur 33, Prinsippskisse: IIS, JRun, JSP, JavaBeans**

Formdata kan hentes inn med getParameter-funksjonen til Request-objektet (`Request.getParameter("name")`), og miljøvariablene aksesseres med ServerVariables-funksjonen (`Request.ServerVariables("name")`). `name` symboliserer henholdsvis navn på formobjekt og navn på miljøvariabel. Eventuelle cookies finnes i miljøvariabelen `HTTP_COOKIE`. Det å aksessere/konfigurere JavaBeans er sett på som så vanlig og nyttig, at det er laget egne JSP-tagger for dette. En komponentinstans kan opprettes med taggen `<JSP:useBean>`, og egenskapene til komponenten kan f.eks settes med taggen `<JSP:SetProperty>`.

## 5.4.5 IIS, JRun, Servlets, JavaBeans

Dette scenarioet beskriver en metode for dynamisk konfigurering ved hjelp av servlets. Scenarioet tar utgangspunkt i et Java miljø med JavaBeans/Enterprise JavaBeans som komponenter, og konfigurering utføres av et Java program (servlet) som kjører på en tjener. Siden webtjeneren (IIS) ikke støtter servlets direkte, må vi knytte til et eksternt program (JRun) som kan håndtere servlets på vegne av webtjeneren (IIS). I dette scenarioet er det servleten som tar seg av den videre konfigureringen. Det er ikke noen stor prinsipiell forskjell på dette fremfor det å bruke et script, men forskjellen ligger i det at koden må (re)kompileres hos servlet-tjeneren før kjøring. En skisse av oppsettet vises i figuren under:




**Figur 34, Prinsippskisse: IIS, JRun, Servlets, JavaBeans**

Java-servleten vil også basere konfigureringen på data fra klienten (browseren). Formdata kan skaffes til veie ved å opprette et objekt av typen `HttpServletRequest`, og kalle objektets `getParameter`-funksjon slik:

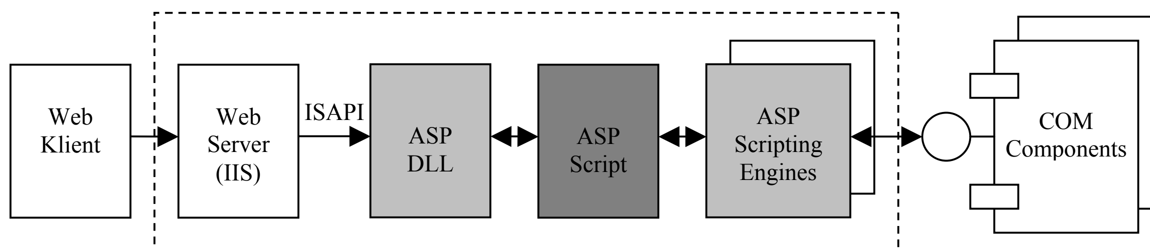
```
HttpServletRequest request;
request.getParameter("name");        // name = formobjekt
```

Miljøvariablene kan også aksesseres via request-objektet, og det finnes egne funksjoner for hver av variablene. `request.getRemoteAddr()` vil f.eks gi deg innholdet av miljøvariabelen `REMOTE_ADDR`.

Aksessering og konfigurering av JavaBeans komponenter er også enkelt, siden man er i et Javamiljø (servlet) og kan bruke språket Java direkte.

## 5.4.6 IIS, ASP, COM

Dette løsningsscenarioet tar i bruk Active Server Pages (ASP) teknologi [7] for aksessering og konfigurering av Microsoft COM komponenter. ASP består av en enkel DLL-fil (ASP.DLL), samt en eller flere ASP scripting motorer (*Scripting Engines*). ASP.DLL filen har som oppgave å gjenfinne (via *parsing*) all script-kode i en ASP-fil, for deretter å videresende script koden til rett scripting motor. ASP tilbyr som standard to scripting motorer, VBScript og JScript. ASP tolken skiller på disse ved at scriptene begynner med hver sin identifikasjon: `<%@LANGUAGE="VBScript"%>` for VBScript (subsett av Visual Basic), og `<%@LANGUAGE="JScript"%>` for JScript (JavaScript). Uansett hvilken scripting motor og scriptspråk man velger, er ideen også her at det er scriptet som er ansvarlig for aksesseringen/konfigureringen av objektene. Sammenhengen vises i figuren nedenfor.




**Figur 35, Prinsippskisse: IIS, ASP, COM**

Som i de tidligere scenarioene, vil det være naturlig for scriptet å basere konfigureringen på data fra og om klienten. Innhenting av klientdata kan gjøres på samme måte som i JSP: Formdata hentes inn med getParameter-funksjonen til Request-objektet (`Request.getParameter("name")`), og miljøvariablene aksesseres med ServerVariables-funksjonen (`Request.ServerVariables("name")`). `name` symboliserer henholdsvis navn på formobjekt og navn på miljøvariabel. Eventuelle cookies finnes i miljøvariabelen `HTTP_COOKIE`.

Både VBScript og JScript har innebygget funksjonalitet for å opprette, samt lage koblinger mot COM komponenter. Koblingen script -> komponent gjøres litt forskjellig i de to scriptspråkene. I VBScript er det mulig å bruke metodene `CreateObject` eller `GetObject`, for å opprette et object, eller en komponentinstans. `CreateObject` metoden bruker PROGID, eller CLASSID for adressering av objektene eller komponentene, mens `GetObject` brukes mest dersom man ønsker å opprette et objekt eller en komponentinstans av en bestemt type, f .eks. et excel-objekt, og som derfor er ment brukt i en bestemt applikasjon. JScript støtter også `GetObject`, men ikke `CreateObject`, istedenfor tilbyr JScript en metode kalt `ActiveXObject` som tilbyr samme tjenesten.

I tillegg til ovennevnte metoder er det også mulig å bruke et standard HTML-tag, kalt <OBJECT> for å opprette objekter eller komponentinstanser på tjeneren. Objektene eller komponenten adressere enten ved at en PROGID streng angis (eksempel 1), eller ved at CLASSID til objektet/komponent angis (eksempel 2).

**Eksempel 1**

```
<OBJECT ID= ”objThis” RUNAT=”SERVER”
       PROGID=”Vendor.objectname”>
   <PARM NAME=”parm1” VALUE=”value1”>
   <PARM NAME=”parm2” VALUE=”value2”>
</OBJECT>
```

**Eksempel 2**

```
<OBJECT ID= ”objThis” RUNAT=”SERVER”
       CLASSID=”clsid:892D6DA7-E0F9-
       11D2-B2E9-00105A42AF30”>
   <PARM NAME=”parm1” VALUE=”value1”>
   <PARM NAME=”parm2” VALUE=”value2”>
</OBJECT>
```

**Figur 36, Opprettelse av en komponentinstans**

I dagens ASP løsninger er det mest vanlig at PROGID brukes (eksempel 1). Dette kan kanskje virke noe overraskende, siden man ved bruk av CLASSID (som er garantert unik i hele verden), aldri vil få en navnekonflikt. PROGID derimot vil aldri være en unik og sikker identifisering av

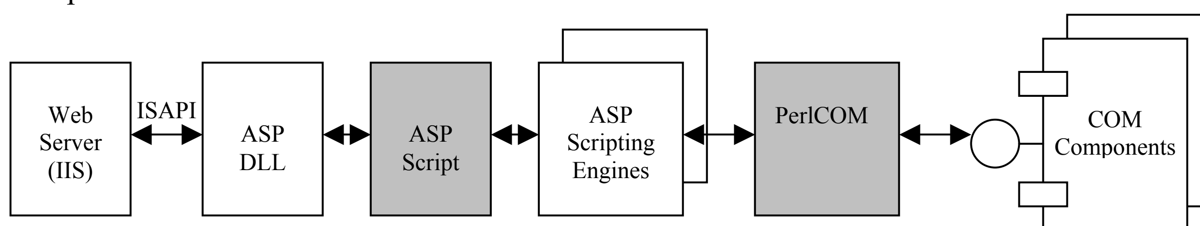
et objekt eller en komponent, siden denne bare består av en "*vendor.component*" streng. Et selskap (*vendor*) i Norge kan godt ha samme navn som et i Danmark osv., og navnene på komponentene velges helt fritt.

Det er også mulig å sette såkalt "*scope*" under opprettelsen av et objekt- eller en komponentinstans. Standard innstilling for "*scope*" er "*page scope*". Med dette menes at objekt- / komponentinstans eksistere like lenge som .ASP siden blir prosessert, og at objekt- / komponentinstansen automatisk blir slettet når ASP-siden er ferdig og resultatet returneres til klienten. Andre mulige innstillinger for "*scope*" er "*session scope*" og "*application scope*".

Selv om ASP tilbyr to forskjellige scriptspråk, advares det mot at script beregnet for forskjellige script motorer, blandes/mikses på samme side. Dette fordi effektiviteten til en slik løsning er mye dårligere en om bare et scriptsråk brukes per side.

### 5.4.7 IIS, ASP, PerlCOM, COM

Dette løsningsscenarioet utvider forrige scenario (5.4.5) ved at ASP ikke instansierer eller kaller komponentene direkte, men istedenfor går via en scripting komponent kalt PerlCOM. PerlCOM er en Perl tolk (*interpreter*) innpakket i en COM komponent. Ved å bruke en slik PerlCOM komponent er det mulig å kjøre Perl kode integrert med (inni i) alle språk, operativsystem, komponentstandarder eller applikasjoner som støtter Microsoft COM komponenter.




**Figur 37, Prinsippskisse: IIS, ASP, PerlCOM, COM**

Før PerlCOM komponenten kan brukes, må det opprettes en instans av PerlCOM komponenten. I vårt eksempel gjøres dette via et ASP metodekall (CreateObject), som beskrevet i forrige scenario. Etter at PerlCOM komponentinstansen er opprettet, kan alle former for Perl scripting brukes. Dette gjelder også alle Perl tilleggsmoduler (*modules*) tilrettelagt for Microsoft-plattformen. Et viktig eksempel på en slik tilleggsmodul for Perl er

Win32.pm. Denne modulen tilbyr en samling av funksjoner som blant annet viser status informasjon til Win32 kjøretidsmiljøet, samt funksjoner for registrering og avregistrering av COM komponenter. PerlCOM kan også brukes sammen med Microsoft DCOM, noe som muliggjør distribuert aksess til Perl. PerlCOM kjører Perl kode (script) via en metode kalt `EvalScript`. Eksempelet under viser hvordan dette kan gjøres (fra ASP):

**Eksempel**

```
ObjPerl.EvalScript ”$greet=’Hello, World.\n’;”
MsgBox objPerl.greet
```

**Figur 38, ASP og PerlCOM**

Dette ekseplet tilordner en tekst streng (Hello, World) til en variabel (greet), og skriver ut variabelens innhold i en tenkt meldingsboks. PerlCOM komponentens instans er kalt objPerl.

En fordel som oppnås ved å bruke PerlCOM for scripting, fremfor ASP alene, ligger i Perl språkets fleksibilitet og styrke. Via moduler tilbyr Perl fleksibilitet og styrke til miljøer som ellers ikke ville hatt dette.

Når det gjelder metoder for å samle inn og behandle klientdata (formdata, miljøvariabler og cookies), kan dette gjøres på samme måte som beskrevet i scenario 5.4.5 eller 5.4.1

# 6 Drøfting

## 6.1 Innledning

Som nevnt i kap. 4.2, mener vi at det finnes tre forskjellige generelle løsningsmetoder (prinsipielle måter) for hvordan dynamisk konfigurering av komponentbaserte tjenester kan realiseres. Vi vil i dette kapitelet belyse positive og negative sider ved disse generelle løsningsmetodene (kap. 6.2 Generelle løsningsmetoder), samt prøve å sette løsningene opp mot hverandre. Videre vil vi se på positive og negative sider ved de praktiske løsningsscenarioene beskrevet i kap. 5, og i den utstrekning det lar seg gjennomføre, prøve å sette disse opp mot hverandre. Gjennom å gjøre en slik sammenstilling av løsnings-scenarioene (kap. 6.3 Praktiske løsninger), ønsker vi å prøve å finne svar på om et bestemt løsningsscenario, eller anvendt teknologi, er det mest egnede med hensyn på realisering. Vi vil også belyse til hva vi mener dynamisk konfigurering er godt egnet, samt positive og negative sider med det å innføre dynamisk konfigurering (kap. 6.4 Anvendelse).

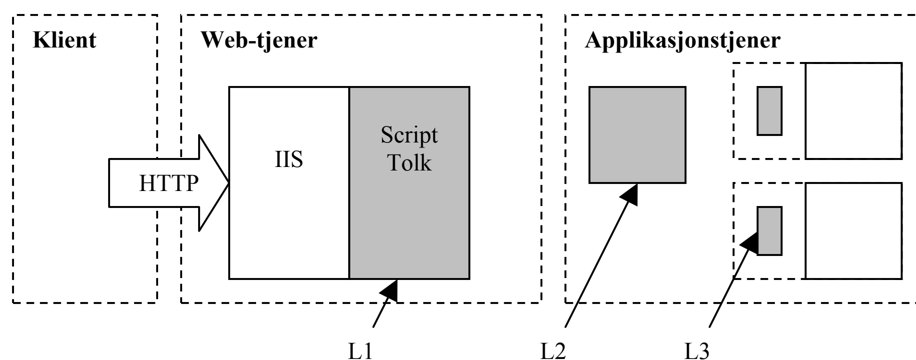
For å evaluere/drøfte anvendt teknologi, samt antatte løsninger og løsningsscenarios, har vi satt opp noen evalueringskriterier som vi mener er viktige: Kompleksitet – d.v.s hvordan de forskjellige delene av systemet er satt sammen, hvor god integrasjon det er mellom disse delene, hvordan en får tilgang til scriptsråk, samt hvordan ”knyttingen” script - komponent utføres. Effektivitet - oppnår vi det vi ønsket gjenom våre løsninger, eller lager løsningene isteden ”flaskehalser” i systemet. Er valgte scriptsråk godt egnet, og finnes det kombinasjoner script/komponentstandard som er spesielt godt egnet. Arkitektur - med arkitektur menes overordnede refleksjoner rundt valg av arkitektur, hvordan arkitekturen påvirker løsningene, samt om en bestemt arkitektur (eller utvidelse av arkitektur) er best egnet. Gjenbruk - vil våre løsninger øke gjenbrukbarheten til tjenestene/komponentene i nevneverdig grad, og går i såfall dette på bekostning av noe annet. Praktisk anvendelse - hvor anvendelige er våre løsninger for praktiske problemstillinger.

## 6.2 Generelle løsningsmetoder

Som beskrevet i kap. 4.2 har vi kommet frem til tre forskjellige prinsipielle- eller generelle måter for hvordan dynamisk konfigurering av komponentbaserte tjenester kan realiseres:

- Løsning 1 (L1): Bruk av scriptspråk i forbindelse med web-tjeneren.
- Løsning 2 (L2): Bruk av egen komponent for scripting.
- Løsning 3 (L3): Utvide tjenestespekteret til komponentstandardene.

Løsningsprinsipp L1 går ut på at script og tolk integreres, så godt det lar seg gjøre, sammen med web-tjeneren. Enten ved at konfigurering utføres av web-tjenerens allerede eksisterende scripting-muligheter (Server Pages teknologi), eller ved at egen script motor ”kobles” til web-tjeneren via CGI grensesnittet. Løsningsprinsippet L2 går ut på at script og tolk legges i en egen komponent som dermed kan instansers og brukes på applikasjonstjeneren som en helt vanlig komponent. Løsningsprinsipp L3 går ut på at tjenestespekteret til komponentstandarden som applikasjonstjeneren bruker, utvides slik at *deploy* (klargjøring) prosessen til komponentene også muliggjør bruk av scripting. En slik løsning vil medføre at hver komponent har sitt eget script integrert, mens scriptspråkets tolk kan være en integrert del av applikasjonstjeneren. Dermed vil det å lage konfigurering av komponentene bli en like naturlig prosess som f. eks. å skrive en XML *descriptor* for klargjøringsprosessen. Dersom datasystemet (tjenesten) har et relativt lavt antall klienter, kan selvfølgelig web- og applikasjonstjener være integrert på samme tjenermaskin, men for relativt store systemer (Internett) er det mer vanlig at web- og applikasjonstjener er adskilt på separate datamaskiner. Skissen under viser prinsippet for plassering/integrering av de ulike løsningene:




**Figur 39, Prinsippskisse av generelle løsninger**

Siden vi i vår oppgave har valgt å legge hovedfokus på L1 er det i første omgang denne løsningen som er uttestet i praksis. Allikevel mener vi at det er mulig å vurdere L1,L2 og L3 opp mot hverandre, spesielt siden L2 gjenom PerlCOM, allerede er uttestet av andre, mens L3 kan betraktes som en logisk utvidelse av en XML *descriptor* (er ment å virke på tilsvarende måte som en *descriptor*, men funksjonelt utføre dynamisk konfigurering).

L1 har en stor fordel fremfor de to andre løsningene ved at script og scripting motor ofte er godt integrert med web-tjeneren (kjører ofte på samme prosess). Dette gir god effektivitet ved kall av metodene som tjenestene tilbyr (dersom antall klienter ikke blir for stort). En annen stor fordel med L1 er at i mange praktiske tilfeller tilbyr web-tjeneren allerede tilstrekkelige scripting muligheter for å realisere dynamisk konfigurering, gjenom f. eks. Server Page teknologi. Dersom bruk av Server Page teknologi ikke gir tilstrekkelig konfigurerings-muligheter (svært avansert konfigurering), kan web-tjeneren også relativt enkelt utvides med en ekstern scripting motor (f. eks. Perl), gjennom CGI grensesnittet. En slik løsning vil imidlertid gå på bekostning av den gode effektiviteten som Server Page teknologien gir. Dessuten kan L1 sin arkitektur virke noe kompleks dersom ekstern scripting motor brukes, spesielt siden klientenes tjenesteforespørsel må gå via flere "ledd" før de kommer til komponentene som utfører selve tjenesten.

L2 sin store fordel fremfor L1 er at ved bruk av egen komponent for dynamisk konfigurering, kan dynamisk konfigurering realiseres og integreres sammen med komponentene på applikasjonstjeneren istedenfor på web-tjeneren. En slik løsning vil dermed flytte logikken for dynamiske konfigurering, over fra web-tjeneren til applikasjonstjeneren, noe som vil bedre effektiviteten til systemer med mange klienter (store distribuerte systemer). Dette forutsetter imidlertid at web- og applikasjonstjener er adskilt på separate tjenermaskiner. L2 muliggjør også konfigurering av allerede eksisterende komponenter, og kan brukes på alle typer av eksisterende komponentstandarder. En annen stor fordel ved dette løsningsprinsippet er at det allerede er relativt godt uttestet på Microsoft Windows plattformen, gjennom en eksisterende COM komponent, kalt PerlCOM. Denne komponenten tilbyr bruk av Perl script, gjennom en Perl-tolk integrert i en separat COM komponent.

L3 antar vi er den mest fleksible og skalerbare av alle alternativene for dynamisk konfigurering. På lik linje med L2 vil dynamiske konfigurering foregå på

applikasjonstjeneren, noe som gir god effektivitet. Men en stor forskjell mellom L2 og L3 ligger i det at komponentenes script ligger sammen med egen komponent (den komponentene den lager konfigurering for). Script-tolk kan enten være integrert med applikasjonstjeneren eller i separate komponenter. Denne løsningen ville ikke laget flaskehalser i systemet, og heller ikke utvidet kompleksiteten til systemet i  nevneverdig grad. Det å lage script for dynamisk konfigurering mener vi at burde vært en naturlig del av prosessen med å utvikle komponenter, og komponentmodellen som applikasjonstjeneren støtter, burde tilby tjenester for hvordan slike script lastes og klargjøres (en del av *deploy* prosessen til komponentmodellen).

Men hensyn på arkitekturbetraktninger vurderer vi L3 til å være det beste, siden dette gir god effektivitet uten å utvide kompleksiteten til arkitekturen i nevneverdig grad. Dette løsningsprinsippet vil også skalere bra i store og distribuerte systemer. Nest best med hensyn på arkitektur er L2, siden script og tolk også i dette eksemplet ligger på applikasjonstjeneren. L1 er det dårligste av alle alternativene med hensyn på skalerbarheten, distribuering og lastbalansering. Dersom vi tar hensyn til praktisk anvendelse vil derimot L1 være en klar vinner, siden dette løsningsprinsippet er basert på en relativt enkel utvidelse av dagens webtjenere, og i enkelte tilfeller tilbyr webtjeneren allerede tilstrekkelig funksjonalitet for å realiser dynamisk konfigurering. L2 er også relativt enkelt å realiser og bruke i praktisk anvendelse, mens L3 er svært vanskelig å realisere i praksis siden komponentmodellenes tjenestespekter må oppdateres. Sammenstilling av de generelle løsningsmetodene:

| | + (Positivt) | - (Negativt) |
|---|---|---|
| L1 | • Kan utnytte scripting egenskapene som web-tjeneren allerede tilbyr (*Server page* teknologi).<br>• Enkelt å realiser sammen med web-tjenere (via CGI grensesnittet). | • Noe kompleks løsning dersom egen script motor brukes, spesielt med hensyn på koblingen script - komponent.<br>• Kan påvirke effektiviteten til systemet. |
| L2 | • Muliggjør enkel integrering mot eksisterende komponentstandarder.<br>• Allerede godt testet for COM komponenter og Windows miljøer. | • Ikke så enkel å fleksibel med hensyn på forandring (noe som gjøres relativt ofte for web løsninger). |
| L3 | • Blir en enkel løsning, der utvikling av script for konfigurering kan inngå som en del av *deploy* prosessen til komponentene. | • Vanskelig å få realisert i praksis.<br>• Kan medføre forskjellige metoder for hvordan dette gjøres på de forskjellige standardene. |

**Tabell 3, Oppsummering av drøfting av generelle løsninger**

## 6.3 Praktiske løsninger (scenarios)

Som beskrevet i kap. 5.4 har vi gjennomført praktiske tester av 7 forskjellige scenarios for hvordan dynamisk konfigurering av en komponentbasert tjeneste kan realiseres. Alle disse testoppsetene er basert på generelt løsningsprinsipp 1, men gir i en viss utstrekning også svar på hvordan generelt løsningsprinsipp 2 ”oppfører seg” under praktisk anvendelse. Scenario 1-5 gjelder testoppsett for konfigurering av Java komponenter (JavaBeans eller Enterprise JavaBeans), mens scenario 6 og 7 gjelder konfigurering av Microsoft COM komponenter. I vårt praktiske testoppsett valgte vi å integrere web- og applikasjonstjener på sammen datamaskin, noe som imidlertid ikke er avgjørende for testresultatene. I tillegg har vi valgt å ha fokus på webtjenerens evne til å realisere konfigurering, f. eks via Server Page teknologi, og webtjenerens evne til å tilby tilleggstjenester, f. eks. i form av ekstern script-motor (tolk).

Dersom tjenesten/komponentene kun trenger en relativ enkel form for konfigurering, gir scenario 3,4,6 og 7 meget gode resultat med hensyn på effektivitet. Dette kommer av Server Page teknologiens gode integrasjonsevne med web-tjeneren, og gjelder i like stor utstrekning for både PSP og JSP, såvell som ASP teknologi. Alle tilbyr gode måter for å opprette instanser av komponentene, samt gode og effektive måter for kall av komponentenes tjenester. Men dersom tjenesten/komponentene trenger mer avanserte former for konfigurering vil et scenario som bruker et mer avansert scriptspråk være et bedre alternativ. Eksempler på sistnevnte kan være scenarioene 1, 3 og 7, der språkene Perl og Python brukes for konfigurering. En annen god egenskap (og som reduserer kompleksiteten), er at hele miljøet på tjeneren integreres i samme miljø, f. eks. Java. Scenario 3, 4 og 5 er eksempler på slike. Alle disse scenarioene bruker servlet teknologien til Java for utførelse av kode. Scenario 3 (PSP) og 4 (JSP) trenger imidlertid ingen kompilering, siden dette blir gjort automatisk av Server Page teknologien (under kjøring).

Når det gjelder konfigurering av COM komponenter har vi sett på to praktiske eksempler, scenario 6 og 7. Begge disse gir god effektivitet siden ASP teknologi brukes, men scenario 7 tilby bedre scripting muligheter gjennom PerlCOM komponenten og Perl språket. Dette går selvfølgelig noe på bekostning av effektiviteten til scenarioet, siden tiden som en tjenesteforespørselen bruker, øker. Både scenario 6 og 7 er oversiktlige og greie å realiser i praksis (ikke for komplekse), og gir begge gode muligheter for dynamisk konfigurering.

Sammenstilling av de praktiske løsningsscenarios kan oppsummeres i følgene tabell:

| | + (Positivt) | - (Negativt) |
|---|---|---|
| S1 | • Gode scripting muligheter gjennom Perl | • Kompleks, Dårlig effektivitet i JPL modulen.<br>• CGI grensesnittet mindre effektivt en Server Page. |
| S2 | • God integrering Jython/ JavaBeans. | • Vanskelig å sette opp miljøet<br>• CGI grensesnittet mindre effektivt en Server Page. |
| S3 | • Tilbyr både Server Page teknologi for enkel konfigurering, og Python script for avansert konfigurering. | • Krever ekstern servlet-tjener |
| S4 | • Server Page teknologi for enkel konfigurering | • Dårlig egnet dersom avansert konfigurering er ønskelig.<br>• Krever ekstern servlet-tjener |
| S5 | • God effektivitet gjennom integrert Java miljø. | • Mindre fleksibelt en ved bruk av scriptspråk. |
| S6 | • Bra effektivitet og god integrasjon med COM komponenter. | • Tilbyr bare enkel konfigurering gjennom Server Page teknologi.<br>• Krever ekstern servlet-tjener<br>• ”Scriptet” må (re)kompileres ved hver endring |
| S7 | • Meget gode scripting muligheter gjennom Server Page teknologi for enkel konfigurering, og Perl for avansert konfigurering. | • PerlCOM komponenten kan bli en flaskehals i store systemer. |

**Tabell 4, Oppsummering av drøfting av praktiske løsninger (scenarios)**

## 6.4 Anvendelse

Gjennom teoristudie av scriptspråk, komponentstandarder, samt våre praktiske test-scenarios, har vi identifisert noen tjenester som vi mener dynamisk konfigurering er anvendelig for. Vi har imidlertid prøvd å begrense dette delkapitelet, siden vår oppgaven har fokus på teknologi som brukes, samt hvordan dynamisk konfigurering kan realiseres, fremfor praktisk anvendelse av dynamisk konfigurering. Vi har derfor begrenset oss til kun å beskrive tre anvendelsesområder: tjenestetilpasning, presentasjonstilpasning og variert sikkerhet /autentisering.

**Tjenestetilpasning:** Med tjenestetilpasning mener vi at ved å innføre dynamisk konfigurering på tjeneste/komponent, vil den funksjonelle virkemåte til en tjeneste/komponent variere fra situasjon til situasjon. En slik tjeneste/komponent som tilbyr situasjonsavhengig funksjonell

tjeneste, oppnås ved at det innføres logikk (if-then-else, case, osv.) i et konfigureringsscript som utføres visse ”valg” på bakgrunn av klient informasjon.

**Presentasjonstilpasning:** Med presentasjonstilpasning mener vi at ved f. eks. å utvikle en komponent som står for tilpasning (formatering) av informasjonen som skal presenteres for klientene, for så å la denne være dynamisk konfigurert, vil en slik komponent ha mulighet til å tilpasse innholdet som skal presenteres til et format som passer klienten. Et eksempel på en slik presentasjonstilpasning kan f. eks. være at dersom en type klienter kun støtter .JPG bilder (f. eks. PDA), mens en annen type klient (f. eks. en bærbar PC) også støtter det nye og bedre .PNG formatet, kan en presentasjonstilpasningskomponent på tjeneren tilpasse bildet som skal sendes til klientene, til et format som klienten støtter. En slik løsning kan derfor i visse situasjoner redusere anvendt båndbredde, siden presentasjonstilpasningen sørger for at informasjonene som skal overføres til klienten, tilpasses til klientens beste mulig presentasjonsformat. Tjeneren trenger heller ikke å lagre bildene i forskjellige format, (noe som er vanlig i dag), men istedenfor tilpasse (under kjøring) både format og oppløsning på bildene, etter klientenes ønsker.

**Variert sikkerhet/autentisering:** Med variert sikkerhet/autentisering tenker vi spesielt på det å bruke klientinformasjon (identifikasjonsparametere og environment variable) til å utføre dynamisk konfigurering av en sikkerhets- tjeneste/komponent. En slik tjeneste/komponent kan dermed tilpasse ”kravene” til sikkerhet ut fra informasjon den mottar fra klienten. Dette kan f. eks. være svært praktisk for web-applikasjoner som tilbyr sine tjenester både på Inter- og intranett, siden kravet til sikkerhet er på et helt annet nivå (strengere), dersom klienten aksesserer en tjeneste fra Internett, enn dersom klienten aksesser den samme tjenesten fra et intranett.

Vi vil nå prøve å sannsynliggjøre hvor godt/dårlig egnet dynamisk konfigurering er i praktiske anvendelser, samt belyse positive og negative sider ved det å innføre konfigurering av tjenester eller komponenter. Den mest positive siden med det å innføre konfigurering av tjeneste/ komponent ligger i det at tjenesten eller komponenten blir mer generell, og dermed tilbyr mere generelle tjenester. Med mere generelle tjenester mener vi da at den funksjonelle virkemåte til tjenesten kan tilpasses og brukes ved flere anledninger enn den ellers ville ha gjort, altså tilby et bredere funksjonelt tjenestespekter. Dermed kan tjenesten eller

komponenten sies å få økt gjenbruksverdi, siden de gjennom konfigurering vil kunne tilpasses til nye krav, og brukes til å løse nye problemstillinger. (Gjenbruk: løse nye problemstillinger ved å tilpasse en gammel løsning, slik at denne kan brukes til å løser den nye problemstillingen). En annen meget stor fordel med det å innføre konfigurering av tjeneste/komponent, er at ved å la konfigurering av tjeneste eller komponent være dynamisk i kjøretidsmiljøet, vil det være mulige å lage såkalte ”smarte” tjenester. Med ”smarte” tjenester mener vi da at den funksjonelle virkemåte til en tjeneste/komponent kan variere fra situasjon til situasjon. En tjeneste/komponent med situasjonsbetinget funksjonell virkemåte, kan i mange praktiske tilfeller være svært anvendelig for å realiser differensierte web-tjenester, der klientens type, environment, profil og tilstand påvirker den funksjonelle virkemåte til tjenesten. Den mest negative konsekvens av det å innføre dynamisk konfigurering av tjeneste/komponent ligger i at arkitekturen til slike løsning ofte blir noe mer kompleks, og at utviklingsprosessen til en slik løsninger må utvides til også å inneholde utvikling av konfigureringsscript for tjeneste/komponent (altså noe mere arbeid, og dermed noe lenger utviklings- tid og kostnad). Disse negative konsekvensene må imidlertid sies å være relativt små, siden økt gjenbruk av gamle løsninger, selvfølgelig også reduserer nødvendigheten av nyutvikling (og dermed også utviklings- tid og kostnader).

Sammenstilling av positive/negative refleksjoner rundt det å innføre dynamisk konfigurering av tjenester/komponenter kan oppsummeres i følgene tabell:

| + (Positivt) | - (Negativt) |
|---|---|
| • Tjeneste/komponent blir mere generell.<br>• Økt mulighet for gjenbruk av tjeneste/komponent.<br>• Muliggjør ”smarte” tjenester/komponenter. | • Øker kompleksiteten til arkitekturen.<br>• Utviklingstiden øker for nyutvikling. |

**Tabell 5, Oppsummering av eksempler på anvendelser**

# 7 Konklusjon

I dag er det ofte vanlig at tjenestene til et datasystem er bygget opp av byggeklosser, kalt komponenter. Tjenestene/komponentene er som oftest utviklet med tanke på å løse spesifikke problemer. Ved å innføre konfigurering av komponentene, eller for hele tjenesten, har vi sett at muligheten for gjenbruk av kode/ferdige løsninger øker. Konfigurering av tjenesten eller komponentene kan realiseres ved hjelp av script. Slike script kan enten installeres sammen med web-tjenere (løsningsprinsipp 1), i egen komponent på applikasjonstjeneren (løsningsprinsipp 2), eller som en integrert del av komponentmodellen som applikasjonstjeneren støtter (løsningsprinsipp 3). Vi har også sett at ved å la scriptene gjøre dynamiske valg under kjøring (dynamisk konfigurering), er det fult mulig å realiser tjenester eller komponenter, der funksjonell virkemåte er situasjonsbetinget. (valg blir gjort på bakgrunn av informasjon den mottar fra klienten).

Gjenom vårt arbeid med denne oppgaven, har vi også kartlagt teknologier og løsningsmetoder for hvordan dynamisk konfigurering av web-tjenester, kan realiseres i praksis (scenario 1-7). Vi gjennomførte også noen relativt enkle arkitekturbetraktninger av testede løsningsscenarios. Disse testene viste at scenarioene som var basert på Server Page teknologi, hadde meget god effektivitet, og var meget godt egnet for realisering av relativt "enkel" form for konfigurering (scenario 3,4, 6 og 7). Dersom tjenesten trengte en noe bedre (mer avansert) form for konfigurering, var imidlertid scenarioene der et eksternt scriptspråk (f. eks. Perl) ble brukt, bedre egnet (scenario 1,3 og 7).

Når det gjelder anvendelse av dynamisk konfigurering, mener vi at tjenestetilpasning, presentasjonstilpasning og autentisering, er områder som vil ha høy nytteverdi av dette.

Ut fra våre erfaringer med anvendt teknologi, og våre praktiske eksperimenter/tester, mener vi å kunne anbefale at konseptet "dynamisk konfigurering av komponentbaserte tjenester", tas i bruk i reelle utviklingsprosjekter.

Vi mener også å se at nye arkitekturkonsepter som er på vei, som f. eks. Microsoft .Net og Sun ONE [51], der web-tjenester har høy fokus, tilbyr noen av de samme egenskapene som vårt "dynamisk konfigurering" konsept, men da med støtte for dette direkte i arkitekturen.

# Referanser

## Bøker

[1] Szyperski C. (1998), Component Software, Beyond Object-Oriented Programming, s.21-48
Addison-Wesley, ISBN: 0-201-17888-5

[2] Brown A.W. (2000), Large-Scale, Component-Based Development, s.71-84
Prentice-Hall Inc, ISBN: 0-13-088720-X

[3] Roman E. (1999), Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition
John Wiley & Sons, ISBN: 0-471-33229-1

[4] D'Souza D.F. (2000), Objects, Components, and Frameworks With UML : The Catalysis Approach, s.1-24
Addison-Wesley, ISBN: 0-201-31012-0

[5] Slama D., Garbis J., Russell P. (1999), Enterprise CORBA, s.35-38
Prentice-Hall Inc, ISBN: 0-13-083963-9

[6] Booch G., Rumbaugh J., Jacobsen I. (1999), The Unified Modeling Language User Guide, s.343-358
Addison-Wesley, ISBN: 0-201-57168-4

[7] Homer A., Sussman D., Francis B. (1999), Professional Active Server Page 3.0, s.1-596
Wrox Press, ISBN: 1-8610002-6-10

[8] Blexrud C. m/flere (2000), Professional Windows DNA, s.5-83
Wrox Press, ISBN: 1-861004-45-1

[9] Kent P., Multer K. (1997), Official Netscape JavaScript 1.2 Programmer's Reference
Ventana Communications Group, ISBN: 1-56604-757-9

[10] Flanagan D. (1998), JavaScript: The Definitive Guide 3rd edition
O'Reilly & Associates Inc, ISBN: 1-56592-392-8

[11] Van Laningham I. (2000), Teach yourself Python in 24 Hours
Sams Publishing, ISBN: 0-672-31734-4

[12] Wall L., Christiansen T, Schwartz Randal L. (1996), Programming Perl 2nd edition
O'Reilly & Associates Inc, ISBN: 1-56592-149-6

[13] Gorden A. (2000), The COM and COM+ Programming Primer, s.77-119
Prentice-Hall Inc, ISBN: 0-13-085032-2

## Artikler

[14] Borgesen P. (1996), Publisering på Internett (LV372D), leksjon 7,8
Høgskolen i Sør-Trøndelag, avdeling for ingeniør- og næringsmiddelfag

[15] Maribu G. (1996), Web-teknikker (LV375D), leksjon 7
Høgskolen i Sør-Trøndelag, avdeling for ingeniør- og næringsmiddelfag

[16] Vinoski S., (1997), CORBA: Integrating Diverse Application Within Distributed Heterogeneous Environment, IEEE Communications Magazine, February 1997.

[17] Roman E., Øberg R. (1999), The technical Benefits of EJB and J2EE technologies over COM+ and Windows DNA, The Middleware Company (http://www.middleware-company.com)

[18] Wang N., Schmidt D., O'Ryan C. (2000), Overview of the CORBA Component Model

[19] Marvie R., Merle P., Geib J. (2000), Towards a Dynamic CORBA Component Platform Laboratoire d'Informatique Fondamentale de Lille- France

[20] Sprott D. (1999), Component Based Development – Using Componentised Software The Forum for Component Based Development and Itegration, May 1999

[21] Thomas A. (1998), Enterprise JavaBeans Technology – Server Component Model for the Java Plaatform Patricia Seybold Group for Sun Microsystem Inc. December 1998

[22] Heineman G. (1997), Composing Software System from Adaptable Software Components Worcester Polytechnic Institute, (http://www.cs.wpi.edu/~heinman)

# Web-sider

[23] Heslop B. (2000), Authors publishing on the Internet, Programming Cookies http://www.authors.com/Navigator4/cookies.html

[24] NCSA HTTPd Development Team (1998), The Common Gateway Interface http://hoohoo.ncsa.uiuc.edu/cgi/overview.html

[25] The Resource Directory for Server Pages http://www.serverpages.com

[26] Haneng A. (2001), Haneng.com – A different ASP developer site http://www.haneng.com

[27] The Esperanto Group, The Web JSP Book http://www.esperanto.org.nz/jspbook/jspbook.pdf

[28] Burridge Brian N., Web Development with Java Server Pages http://www.burridge.net/jsp

[29] Hall M. (2000), Servlets and Java Server Pages – A tutorial http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial

[30] Sun Microsystems Inc (2001), Introduction to Java Server Pages http://developer.iplanet.com/viewsource/kuslich_jsp/kuslich_jsp.html

[31] Angell Enterprises (1999), Python Server Pages http://www.ciobriefings.com/psp

[32] O'Reilly & Associates (2001), The Source for Perl http://www.perl.com

[33] Python Language Website http://www.python.org

[34] Gruet R. (2000), Python 1.52 Quick Reference http://www.wag.caltech.edu/home/rpm/python_course/python_quick.html

[35] Source Forge, Jython Home Page http://www.jython.org

[36] Sun Microsystems, The Source for Java Technology
http://java.sun.com

[37] Netscape Communications Corporation (1999), DevEdge Online – Documentation & Information
http://developer.netscape.com/docs/manuals/index.html?content=javascript.html

[38] Microsoft Corporation (2000), Microsoft Scripting Technologies
http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript/techinfo/vbsdocs.htm

[39] The CorbaScript Language
http://corbaweb.lifl.fr/CorbaScript

[40] O'Reilly & Associates (1998), Perl Resource Kit, Utillities Guide chapter 3
http://www.oreilly.com/catalog/prkunix/excerpt/UGch03.html

[41] O'Dea J., JPL – Java/Perl Lingo
http://www.seas.smu.edu/~jodea

[42] Allaire Jrun
http://www.livesoftware.com/Products/JRun

[43] The forum for Component Based Development and Itegration
http://www.cbdiforum.com/

[44] The Object Management Group
http://www.omg.com/

[45] TheServerSide.com
http://www.theserverside.com/home/index.jsp

[46] BEA WebLogic – Application Server
http://www.bea.com/index.shtml

[47] IBM Websphere – Application Server
http://www.ibm.com/websphere

[48] Borland AppServer – Application Server
http://www.borland.com/appserver/

[49] Sybase EAServer – Application Server
http://www.sybase.com/products/applicationservers/easerver

[50] Oracle 9i Application Server – Application Server
http://www.oracle.com/ip/deploy/ias/index.html

[51] Sun Open Net Environment (Sun ONE)
http://www.sun.com/software/cover/2001-0205/

# Appendix

## Appendix A: Opprinnelig oppgavebeskrivelse

Mange av dagens klient-tjener systemer er basert på forholdsvis statiske modeller, dvs at tjeneren er nøye tilpasset en bestemt klients type, behov og bruksmønster. Oppgaven skal undersøke om det er hensiktsmessig å trekke ut tjenerens funksjonalitet i mindre separate komponenter, og gjøre disse konfigurerbare ved hjelp av f. eks script. Målet med dette vil være å kunne velge ut, tilpasse og bruke hver enkelt komponent ut fra klientens type, behov, bruksmønster og kjøretidsomgivelser. Ved å la komponentene være dynamisk konfigurerbare, vil det sannsynligvis være enklere å tilby tjenester som er tilpasset "systemets tilstand" og klientens behov.

Oppgavens eksperimentelle del (prototype) skal ha hovedfokus på web-teknologi, med browser som klient og webserver som tjener. Det skal undersøkes om man kan sette sammen og konfigurere en tjeneste (Java-applet) dynamisk ved hjelp av script (f.eks Perl, Python, PHP). Det skal undersøkes om tjenesten (appleten) kan bygges opp av ulike separate Java-komponenter (f.eks Enterprise JavaBeans) på tjeneren. Valg av komponenter og tilpasning av disse skal gjøres ved hjelp av script på tjeneren. Det skal også belyses ulike metoder på klienten for å beskrive behov, bruksmønster og brukerprofil, gjerne ved bruk av cookies, parametre og ulike klientscript.

I lys av den eksperimentelle delen (prototypen), skal det identifiseres hvilken type funksjonalitet som er hensiktsmessig å plassere i separate komponenter.

# Appendix B: Teknologier i Sun J2EE

| Teknologi (API) | Funksjon |
|---|---|
| Enterprise JavaBeans (EJB) | Arkitektur for å bygge/bruke tjener-side- og gjenbrukbare-komponenter. |
| Java Naming and Directory Interface (JNDI) | Tjeneste for å lokalisere resurser på et nettverk, som f. eks. brukerprofil, tjenermaskiner og EJB-komponenter. |
| Java Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP) | Tjeneste som muliggjør metode-invokering på tvers av "Java virtual machines". IIOP muliggjør også at J2EE kan brukes sammen med CORBA-kode skrevet i andre programmeringsspråk. |
| Java Database Cennectivity (JDBC) | API som tilbyr grensesnitt for aksisering av relasjonsdatabaser, som f. eks. Oracle og SQL Server. |
| Java Message Service (JMS) | Muliggjør asynkron kommunikasjon. Punkt til punkt, eller publish/subscribe utveksling av meldinger. |
| Java Interface Definition Language (Java IDL) | Java-basert tjeneste som tilbyr et subsett at CORBA ORB spesifikasjonen. |
| Connectors | Tjeneste som muliggjør aksess mot eksisterende enterprise informasjonssystemer, som CICS, Tuxedo og SAP R/3. |
| Java Servlet | Servlet er request/respons orienterte programvare deler som kjører på en web-tjener (Mange likhetstrekk med JSP). Tilbyr Sessions-tilstand, of dynamisk generert HTML-kode. |
| JavaServer Page (JSP) | Server Page teknologi som muliggjør at web-sider kan genereres dynamisk. |
| Java Transaction API (JTA) | En sammenbinding av programvare nødvendig for å tilby en tjeneste som muliggjør bruk av transaksjoner. |

# Appendix C: Teknologier i Microsoft Windows DNA

| Teknologi (API) | Funksjon |
|---|---|
| Windows 2000 | Windows DNA applikasjoner bruker/trenger Windows 2000 som underliggende operativsystem. |
| COM+ | En arkitektur for bygging/bruk av tjener-side- og gjenbrukbare-komponenter. |
| Distributed COM (DCOM) | En teknologi som muliggjør bruk av komponenter på distribuerte tjenere. Separerer spesifikasjon og implementasjon (grensesnitt). |
| Microsoft Transaction Server (MTS) | En blanding av en TP-monitor og en "Object Request Broker" som administrerer og styrer bruken av tjener-side komponenter. |
| Microsoft Message Queue (MSMQ) | Et "message-queuing" produkt for asynkron kommunikasjon mellom tjenester eller komponenter (transaksjoner). |
| Microsoft Active Data Object (ADO) og OLE DB | API'er som tilbyr tjenester for aksisering av relasjonsdatabaser, som f. eks. Oracle og SQL Server. |
| Microsoft "Babylon" Integration Server | Tjeneste som muliggjør aksess av eksisterende enterprise informasjonssystem, som CICS og MVS. |
| Microsoft Internet Information Server (IIS) | En web-tjener som tilbyr Internet Server API (ISAPI) for effektiv integrering av web-applikasjoner. |
| Active Server Pages (ASP) | Server Page teknologi som muliggjør at web-sider kan genereres dynamisk. |
| Microsoft Active Directory | Tjeneste for å lokalisere resurser på et nettverk, som brukere, datamaskiner, komponenter osv. |