Göteborgs Universitet
Institutionen för Informatik
Magisteruppsats IA7400
Vårterminen 1999

TUNNA ELLER TJOCKA KLIENTAPPLIKATIONER

- EN STUDIE I CLIENT/SERVER-TEKNIK

Torbjörn Ericson & Anna Tykesson

Abstrakt

Denna magisteruppsats i informatik behandlar valet av tunna respektive tjocka klienter i client/server-applikationer. Problemområdet behandlas utifrån utvecklarens synvinkel. Syftet med uppsatsen är att ta fram riktlinjer för val av arkitektur vid utvecklingen av framtida client/server-applikationer. För att få fram konsekvenser av arkitekturval och skillnader i utvecklingsarbetet har empiri i form av fallstudie med prototypkonstruktioner kombinerats med litteraturstudie och intervjuer. Som underlag för jämförelsen tas grundläggande teori kring skiktad arkitektur och kommunikationsteknologier upp. Det framkommer att utveckling av tunna klienter är mer resurskrävande än utvecklingen av tjocka samt att den ökade flexibiliteten med tunna klienter står att väga mot den ökade komplexiteten. Det finns en rad aspekter att beakta vid valet av tunn eller tjock klient och man kan inte ensidigt säga att den ena lösningen är att föredra.

Handledare:

Per Dahlberg	Viktoria Institutet, Göteborg
Björn Birk	Astrakan Strategisk Utveckling AB, Göteborg

*Vi vill tacka **Björn Birk**, vår handledare på Astrakan Strategisk Utveckling AB för allt stöd under arbetet med vår magisteruppsats. Framförallt har han varit till stor hjälp vid utvecklingen av våra prototyper.*

***Per Dahlberg**, vår handledare på Viktoria Institutet har bidragit med många idéer och kommentarer kring vårt arbete. Vi tackar Per för att han fungerat som bollplank och med stort tålamod granskat vår uppsats.*

*Vi vill också särskilt tacka **Thomas Yregård**, **Petter Wolf**, intervjupersonerna samt alla på Astrakan som intresserat sig för vårt examensarbete och fått oss att trivas under vår tid på Astrakan.*

19:e Maj, 1999

1 Inledning

1.1 Bakgrund

Ett problem som utvecklare ställs inför idag är valet av arkitektur vid konstruktionen av client/server-applikationer. Inga klara riktlinjer finns tillgängliga för när det är lämpligt att välja den ena eller andra typen av arkitektur. Majoriteten av de rekommendationer som ges styrs av trendsättning, dvs antingen skall man applicera den ena eller den andra arkitekturen, utan hänsyn taget till vilken miljö som aktuell applikation skall verka i. Astrakan är ett av många företag som ställs inför just berörd fråga vid utveckling av client/server-applikationer.

Astrakan är ett konsultföretag som har kunskap inom management, kommunikation och informationsteknologi. Kunderna är medelstora och stora företag, med vilka man skapar långsiktiga relationer. Inblandningen sker tidigt i kundens förändrings process. Registret av branscher som kunderna tillhör är brett; bl.a. sjukvård, media, industri och försvar. Verksamheten har delats upp i fem affärsområden; IT i verksamheten, Produkten och Affären samt Kundens förändrings- och kunskapsprocess. Var kommer vi in? På utvecklingsavdelningen hos Astrakan producerar man huvudsakligen applikationer i arkitektur med feta klienter. När man fattar ett beslut om vilken arkitektur, som är ändamålsenlig har man inga strikta regler. Många gånger efterfrågar kunden tunna klienter, p g a att det anses lättare och mer kostnadseffektivt att underhålla. På andra sidan står utvecklaren och leverantören, som har många faktorer att ta hänsyn till, kanske är det snabbare och billigare att utveckla en arkitektur med feta klienter, kanske finns det vissa faktorer som kräver användningen av tunna klienter t ex krav på specifika funktioner. Att ha tillgång till svaren på många av dessa frågor skulle underlätta utvecklingsprocessen och kanske t o m påskynda utvecklingen. Ur ett akademiskt perspektiv är det intressant att ta reda på skillnader mellan olika typer av klienter p g a att skrivelser på området är sparsmakade, åsikterna rörande skillnader mellan de båda klientstrukturerna anges i korthet utan någon grundligare redogörelse för vad som ligger bakom dessa påståenden. En mer ingående undersökning som knyts till existerande åsikter för att se om de har en djupare grund torde komplettera den information som idag finns på området. Vidare är de rent praktiska erfarenheterna på området inte tillräckligt omfattande för att regler eller modeller för utveckling skall ha haft en möjlighet att slå rot [Loosley & Douglas, 1998, s35], m a o skulle även ett bidrag i form av utvecklingserfarenhet ge en nyttoeffekt för konceptet, genom att man får ytterligare referens vid konstruktion av en eventuell framtida modell.

Bo Dahlbom skriver i sin artikel om synen på informatik i Göteborg, att man definierar informatikämnet som framförallt studier i användningen av informationsteknologin[1]. Han tilläger att man är intresserad av att förändra och förbättra den användningen. Vi anser att detta ligger väl i linje med avsikten i vår uppsats då våra intentioner är just att förbättra och förändra användningen av existerande tekniker inom informationsteknologin. Vår ansats överensstämmer väl även med de tankegångar som Bo Dahlbom tar upp i sin artikel om den nya informatiken, där han lägger vikt vid just användningen av informationsteknologin[2].

[1] Dahlbom, B., ***"Göteborg Informatics",*** Institutionen för Informatik, 1995
[2] Dahlbom, B., ***"The New Informatics",*** Institutionen för Informatik, 1997

1.2 Problemområdet

När client/server-konceptet tog fart var det början till en helt ny era för uppbyggnaden av företags IT-infrastruktur. Decentralisering var nyckelordet, användarna skulle få mer kontroll, genom att ha datan i direkt anknytning till sin bordsdator. Sedan dess har begreppet ändrat form ett flertal gånger och utvidgats till att inkludera än fler termer. Man talar om enkel client/server-arkitektur, distribuerad data, tredelad client/server-arkitektur mm. För begreppsdefinitioner se Appendix A. Distribuerad databas och distribuerade program möjliggör en uppdelning av både program och databas på flera plattformar. Vidare talar man om olika lösningar för respektive arkitektur; en-, två-, tre- och till och med fyrskiktslösningar[3], som berör frågan om hur man delar upp de olika komponenterna i arkitekturen. Den senaste trenden är en återgång till centralisering, kontrollen förflyttas till administratören av servern. Många hävdar idag att en ny era har inletts, en tidsålder där man rör sig från pc datorer, fulladdade med logik till enklare terminaler som bara kräver ett fåtal resurser [Pettersson, 1996; Neoware Systems, 1998; Sandred, 1998]. Huvudargumentet för att övergå till tunna klienter är stora kostnadsbesparingar i underhåll, administrationen av centraliserade system blir näst intill obefintlig, då den koncentreras till servern [Neoware Systems, 1998]. Men nog måste det finnas fler faktorer att ta hänsyn till, även om kostnader kan tänkas vara en tungt vägande faktor? Visserligen omtalas andra faktorer, såsom prestanda och utveckling men dessa kommer något i skymundan jämfört med ovanstående argument, som lyfts fram skarpast. Vidare är åsikterna många om vad man bör använda, men erfarenheterna av att både bygga och använda olika former av arkitekturer få [Loosley & Douglas, 1998]. För att få till stånd ett bredare perspektiv på användningen av respektive teknik, skulle det vara ändamålsenligt att dra fram de mindre uppmärksammade aspekterna i ljuset, t ex utvecklingen av tunna respektive tjocka klienter.

1.3 Problemformulering

Den centrala frågan, som behandlas i denna uppsats är som följer;

Vilka effekter får valet av tunna respektive tjocka klienter för utvecklingen av en client/server-applikation?

1.4 Syfte

Ändamålet med uppsatsen är att, genom litteraturstudie och utveckling av prototyper, upprätta ett förslag på vad man bör tänka på när man står inför att utveckla en client/server-arkitektur. Idéerna är ämnade att användas som vägledning vid utveckling av framtida client/server-applikationer hos Astrakan samt andra som har behov av en oberoende undersökning och utredning av de aktuella teknikerna. Förslagen kan tillsammans med kraven på en applikation, samt andra undersökningar på området, ligga till grund för valet mellan tunn och tjock klient.

[3] Lewandowski S. M., ***"Frameworks for Component-Based Client/Server Computing"***, ACM Computing Surveys, Vol. 30, No. 1, March 1998.

1.5 Avgränsning

Studien är gjord ur ett utvecklarperspektiv. M a o läggs huvuddelen av koncentrationen på de erfarenheter som görs rörande skillnader i konstruktionen av de olika arkitekturerna. Fokus har även lagts på skillnader i egenskaper hos de båda arkitekturerna och vad de kan innebära för en applikation. Sistnämnda undersökningsaspekt betyder att man också kan sägas gå in på användarens område, vilket kan ses som ett naturlig inslag p g a att utvecklären gör en produkt för olika användare och naturligtvis måste ta hänsyn till deras intresse vid utvecklingen.

Vi har inte för avsikt att inom ramen för vår studie undersöka för och nackdelar med 2-skikts och 3-skiktslösningar i client/server-applikationer. Någon jämförelse mellan olika kommunikationstekniker för distribuerade system såsom RMI (Remot Method Invocation), CORBA (Common Object Request Broker) och DCOM (Distributed Component Object Model) kommer inte heller att göras. Vi ämnar inte att föra en diskussion om mottagandet av feta respektive tunna klienter, detta p g a att en sådan jämförelse skulle vara högst tidskrävande och resultaten i stor grad subjektiva. Vi anser att en jämförelse som lyfter fram skillnader, som val av den ena eller andra klientstrukturen kan innebära för utvecklingen, kan utgöra en bidragande hjälp för att fatta beslut om arkitektur, förutsatt att man från början vet vilka krav som finns från olika intressenter av applikationen i fråga.

1.6 Rapportstruktur

Uppsatsen är upplagd på följande sätt:

I avsnitt 2, **”Metod”**, behandlar vi olika vetenskapliga forskningsmetoder som vi använt för att få fram information om problemområdet. Här diskuteras också vårt val av metoder, för anpassning till aktuell studie.

Avsnitt 3, **”Teori”**, innehåller material som skall fungera som ett referensverk för resterande sektioner av uppsatsen.

I avsnitt 4, **”Resultat”**, redovisar vi resultatet av vår litteraturstudie, våra intervjuer och de erfarenheter som vi gjort under utvecklingen av prototyperna. Vi har även valt att inkludera en beskrivning av prototypen i anknytning till redogörelsen för utvecklingserfarenheter.

Med avsnitt 5, **”Diskussion”,** vill vi förmedla de slutsatser som kan dras genom tolkning av vad vi kommit fram till i resultatavsnittet. Här tar vi upp våra egna synpunkter och vill även bidra med värdering av resultatet. Vilka områden och frågeställningar som skulle kunna ligga till grund för framtida studier inom området behandlas också.

2 Metod

Detta stycke syftar till att ge läsaren en uppfattning om hur vi gått tillväga för att nå fram till resultatet. Syftet med avsnittet är också att delge de tankar som ligger bakom de olika angreppssätten och knyta dessa till vetenskapsteoretiska aspekter. De praktiska momenten inleder avsnittet, detta är en redogörelse över varför vi tagit med aktuell metod och i vissa fall även vad man bör tänka på vid användning av en sådan metod. De praktiska styckena följs av en diskussion berörande centrala filosofiska begrepp som kan appliceras på vår metod, vidare ställs kvantitativa metoder mot kvalitativa i en diskussion om vad som är tillämpligt i vår studie, för att avslutas med ett avsnitt om vilken form av forskning som vår metod kan sägas följa. Inledande ges redogörelsen över litteraturstudien, vilken företogs som stöd för att kunna besvara frågan om skillnader i utveckling. Litteraturstudien följdes upp av intervjuer för att ta reda på intressen som kan påverkar utvecklarens val. Slutligen genomfördes en fallstudie och prototypkonstuktion i syfte att möjliggöra den eftersträvade jämförelsen.

2.1 Litteraturstudie

Vi ville inleda med att göra en litteraturstudie dels för att ta del av andras erfarenheter inom problemområdet, klargöra centrala begrepp samt få fram faktorer som skulle ligga till grund för jämförelsen av de två teknikerna. Ur litteraturstudierna utvinns också idéer och uppslag, vilka påverkar arbetsgången i resterande del av undersökningen. Det kan röra sig om tidigare undersökningar och erfarenheter, vilka gör att man undviker ett visst förfaringssätt eller försöker lägga extra uppmärksamhet på olösta frågor. Böcker, tidningsartiklar och internetpublikationer har använts som material vid studien. Vad gäller material från internet försökte vi vara noga med att välja ut skrivelser av icke-kommersiell natur, som t ex uppsatser från utbildningsenheter, chansen är annars större att materialet blir färgat och därmed också vår studie.

2.2 Intervjuer

För att få fram de faktorer som påverkar valet av arkitektur, har vi tagit hjälp av personer med erfarenhet av berörda områden. Vad gäller slutanvändare har vi m a o intervjuat individer som har jobbat med slutanvändare. Anledningen till valet av detta tillvägagångssätt är att det hade varit alltför tidskrävande att sätta sig med användare och kanske inte heller alls ändamålsenligt, då användare inte alltid kan sätta ord på vilka egenskaper ett system bör ha. Är man dessutom intresserad av generella krav på system så kan det vara svårt för slutanvändare att formulera sådana krav, då de enbart kan anknyta till de specifika system som de har erfarenhet av. Någon som har längre erfarenhet av att jobba med slutanvändare har dock kontroll över vilka olika krav som kan existera och översätta kraven till tekniska termer. Intervjuerna beslutades genomföras informellt och semistrukturerat, dvs vissa frågor skulle vara förberedda medan andra skulle tas upp under intervjuns gång. Valt förfaringssätt grundar sig på att vi ville fånga upp så mycket kunskap som möjligt och ansåg att en fri konversation på bästa sätt tillgodoser detta, på så sätt kan nya infallsvinklar komma upp under intervjuns gång. I tillägg ansåg vi att personer som har mycket erfarenhet inom ett specifikt område äger stora mängder kunskap som kan tas fram. Ställs några inledande frågor räcker detta för att kunna utveckla diskussionen vidare. Efter redogörelse av vilken information vi var intresserade av, fick vi rekommendationer inom företaget och från handledare om vilka personer som skulle kunna vara lämpliga att intervjua. Vi valde att intervjua tre personer för att få fram de olika intressentgrupperna (för en sammanställning av de olika

intressentgrupperna se 4.2.2.5). Det låga antalet intervjupersoner berodde huvudsakligen p g a tidsramarna. Vi anser ändå att man får en tillräcklig bild över vilka krav som kommer från skilda håll, eftersom intervjuerna var till främst för att kontrollera den information vi redan erhållit men också för att komplettera bilden. Vi har inte tagit hänsyn till aspekter som kön eller ålder, utan som tidigare nämnt enbart sett till den erfarenhet som personerna besitter.

2.3 Fallstudie

På grund av den tidsrymd vi hade till vårt förfogande, kan det ses som en omöjlighet att företa en heltäckande studie av teknikerna, dvs en studie som kan appliceras oavsett vilken slags applikation man använder. Vi har därför valt att göra en studie av ett par olika typer av applikationer, för att kunna uppnå substans i resultatet.

Fallstudie benämns som en metod för att undersöka processer på djupet. Innebörden av begreppet är att man undersöker ett fåtal objekt i en rad avseenden [Wiedersheim-Paul & Eriksson, 1991]. Att man kan applicera detta på vår studie påvisas genom att vi undersöker två applikationer ur utvecklings och funktionalitetsaspekter.

Användningen av fallstudie kan ske i olika syften [Wiedersheim-Paul & Eriksson, 1991]. I vårt fall är studien fokuserad på att ge en teori om när en viss arkitektur är tillämplig. Man kan påvisa att vår fallstudie använts i två syften, dels kan man se den som en metod vid aktionsforskning, dels som hjälpmedel, att skapa nya tankar kring området. Tanken är att på något sätt påverka organisationens sätt att fatta beslut om arkitektur, detta görs genom att, genom jämförelsen ta fram en uppsättning riktlinjer som kan tjäna som beslutsunderlag. Fallstudien kan vidare ses som hjälpmedel i den bemärkelsen att man får en grund till jämförelse, när man vill komma fram till en teori om i vilken situation en viss arkitektur är lämplig.

Frågan om generaliserbarhet är central vid en studie av denna typ. Syftar utredning och forskning till förståelse av speciella situationer lämpar sig denna metod bra, generaliserbarheten kan dock vara svår att hävda eftersom man bara berör enskilda situationer [Wiedersheim-Paul & Eriksson, 1991]. Om man däremot som vi, väljer att studera två fall innebär detta att jämförelsemöjligheten ökar. Koncentrationen på det enskilda fallet minskar dock, vilket kan resultera i lägre förståelse för detta. Detta torde dock inte påverka vår studie i högre grad, då skillnaden mellan fallen är av intresse, snarare än varje enskilt fall, isolerat.

2.4 Prototypkonstruktion

För att få fram jämförelsen mellan de båda teknikerna ville vi konstruera två prototyper av exempelapplikationerna, som realiserar respektive teknik, dvs en representation av vardera arkitektur. Genom arbetet med konstruktionen av applikationerna erhålls erfarenheter som hjälper till att framförallt besvara frågan om skillnader i utvecklingsarbete men också frågor berörande funktionalitet.

2.5 Val av exempelapplikation

När man väljer applikation bör ändamålet, vad man vill visa med hjälp av applikationen, ligga i fokus. Konstruktionen av våra prototyper syftade bl.a. till att kunna besvara frågor kring funktionalitet och utvecklingsarbete. Man måste dessutom titta på om applikationerna kan ge en snedvridning av resultatet, en applikation kanske lämpar sig väldigt väl för en viss

arkitektur. Vad man skall göra är att försöka hitta en applikation som ligger neutralt, dvs som inte verkar mer eller mindre lämplig för en viss arkitektur, alternativt bygga en applikation som innehåller egenskaper som kan tänkas lämpade för vardera typ av arkitektur. Man måste försöka hitta en applikation som inte ligger i någon av extremerna vad det gäller databasinteraktion, beräkningsintensitet mm. Applikationen måste dessutom gå att utveckla inom tidsramen för vår magisteruppsats.

2.6 Val av utvecklingsverktyg

För modelleringsfasen med design av objektmodell, klassdiagram mm så behövde vi ha ett utvecklingsverktyg som stödjer UML-notationen, då vi planerat att använda oss av denna metod för objektorienterad systemutveckling. Vid valet av programmeringspråk var kravet att språket skulle vara lämpat för utveckling av client/server-applikationer med både tunna och feta klienter. Ytterligare punkt som valet styrdes av, var de utvecklingsverktyg, man hade tillgång till och hur väl förtrogen man var med dessa.

2.7 Filosofiskt ställningstagande

Vad sker först i vår studie, datainsamling eller teori? Problemformuleringen är öppen, på så vis ges inte en hypotes, som man från start önskar pröva. Däremot vill man, genom en litteraturstudie, intervjuer och prototypkonstruktion få idéer som ligger till grund för en teori, bestående av en rad uppfattningar om vilket angreppssätt som kan vara tillämpligt i olika situationer. Således utgör datainsamlingen det första steget, vilket följer den fenomenologiska paradigmen. I litteraturen om forskningsmetoder [Easterby-Smith, Lowe & Thorpe, 1991] står omnämnt, att huvudsyftet med grundad teori, som tillhör det fenomenologiska angreppssättet, är att utveckla teori genom metoder som syftar till jämförelse. Samma händelse eller process undersöks i olika miljöer eller situationer, vilket kan översättas till ändamålet med vår uppsats, nämligen att jämföra samma applikation konstruerad i två skilda miljöer. Varför passar denna inriktning vår studie? Ur jämförelsen ville man utvinna förklaringar och nya insikter vilket också följer mallen för grundad teori [Easterby-Smith m. fl., 1991]. Man kan argumentera för att vi från en början redan visste lite om vad som önskades utredas och därav borde en teori existera, visserligen är så fallet men det utredningen syftar till att ge, stämmer inte med utkomsten av en hypotesprövning. Vår studie är ämnad att erbjuda en mer ingående redogörelse än ett svar på en fråga som endast dementeras eller bekräftas.

Hur mycket ville vi påverka organisationen med vårt arbete? Som nämnt vill vi bidra med stöd inför val av teknik så att valet blir mer medvetet. Företaget skall kunna argumentera för valet av teknik. Syftet är att till viss grad påverka sättet att arbeta inom en sektion i företaget [Easterby-Smith m. fl., 1991]. Man kan inte kalla detta ren aktionsforskning, påverkan sker inte under hela utredningen, eftersom man först måste utarbeta resultatet för att sedan överföra detta på organisationen, dvs organisationen har inte inblandats i själva processen, som sig bör i renodlad aktionsforskning.

Hur berörs vår undersökning av begreppen validitet, reabilitet och generaliserbarhet? Innebörden av dessa begrepp varierar beroende på vilket filosofiskt synsätt, som är tillämpligt eller som man väljer att anta [Easterby-Smith m. fl., 1991]. Talar man om validitet i den fenomenologiska inriktningen är innebörden huruvida man som forskare fått full tillgång till information och åsikter hos de informationskällor man använt sig av. I vårt fall, de personer som varit föremål för våra informella intervjuer. Rörande begreppet reabilitet är det samma filosofiska inriktnings mätning av begreppet som kan appliceras på vår studie. Fenomenologin

ställer frågan om en liknande observation skulle göras av andra forskare, vid andra tillfällen, vilket kan knytas till den jämförelsen, som är ett resultat av vår fallstudie. Vad gäller begreppet generaliserbarhet är det även här, den fenomenologiska inriktningens definition; om idéer och teorier som framkommit i en situation är gångbara i andra sammanhang, som bäst överensstämmer med vår ansats. I vårt fall är frågan; om de bedömningar vi kommit fram till är användbara vid byggande av andra applikationer av den typ vi valt att undersöka. Frågan om generaliserbarhet är ytterst viktig för att se om syftet med utredningen har uppnåtts, nämligen att kunna använda resultatet i alla situationer där man skall utveckla applikationer av det slag vi studerat.

2.8 Kvalitativa och kvantitativa metoder

Vilka metoder har vi använt för att samla in den data vi behöver för att besvara aktuell frågeställning? Då vi från början inte visste exakt vilka resultat som var tänkbara, ansåg vi det, till övervägande del vara lämpligt med kvalitativa metoder. Kvantitativa metoder kan användas när man vet vilka företeelser man önskar mäta, med kvalitativa metoder däremot tittar man på hur något uppfattas, men får också fram olika företeelser som kan ställas mot varandra. Sistnämnda faktorer överensstämmer med vår studie i den bemärkelsen att vi inte haft någon företeelse som kunnat mätas i siffror, önskan har varit att få fram punkter där utvecklingen av arkitekturerna skiljer sig åt samt aspekter på funktionalitet i de båda miljöerna, dvs rent kvalitativa företeelser. Olika delar av vår studie har krävt skilda angreppssätt, för att få fram önskad information. Den kvalitativa metod som är tillämplig är framförallt fallstudien. Fallstudien till vilken prototyputvecklingen är knuten, har varit tillämplig i den del av utredningen, där man ville ta reda på vilka effekter arkitekturvalet får för utvecklingen av de två applikationerna. Informella semistrukturerade intervjuer [Easterby-Smith m. fl., 1991] ansågs lämpa sig bäst för att identifiera grupper som berörs av arkitekturvalet, samt fånga upp aspekter som grupperna upplever som viktiga, berörande arkitekturerna. Varför det ansågs lämpligt var för att man har möjlighet att erhålla en förklaring till vad som påverkar åsikterna om och uppfattningen av en viss situation [Easterby-Smith m. fl., 1991], i detta fall situationen att man använder en arkitektur istället för en annan. Som enda inslag av kvantitativ metod i vår undersökning har vi vår litteraturstudie. Litteraturstudier, räknas till kvantitativa metoder [Easterby-Smith m. fl., 1991], i synnerhet nyhetsartiklar, vilka utgör en stor del av det material som är relevant för vår utredning. I litteraturen om forskningsmetodik [Easterby-Smith m. fl., 1991] står omnämnt nyttan av att kombinera kvantitativa och kvalitativa metoder, dvs metodologisk triangulering. Användandet av flera metoder gör att man får ett vidare angreppssätt och därmed en bredare informationskanal, vilket kan bidra till att undersökningen blir mer heltäckande. I tillägg innebär användandet av kvantitativa metoder en förstärkt reabilitet, då man har konkreta siffror och fakta på resultat. Då vår studie till stor del saknar ett kvantitativt inslag, missas ovanstående aspekter. Inriktningen på vår studie ger, som tidigare nämnt, dock inte utrymme för kvantitativa mätningar, det är en omöjlighet att pressa fram siffror på vilka konsekvenser arkitekturvalet får för utvecklingen, mer än möjligen i form av tidsrymd och kostnader. Vad gäller reabiliteten är det helt klar ett bekymmer att utesluta kvantitativa metoder, men resulterar studien i punkter som stöds av tidigare erfarenheter eller uppfattningar torde resultatet inte uppfattas som helt ogrundat.

2.9 Form av forskning

För att kunna svara på frågan om vilken form av forskning som är ändamålsenlig, bör man titta på vad som skall utvinnas ur vår studie. Resultatet skall dels ge en jämförelse mellan de två teknikerna, varur vi vill utvinna punkter som skall bidra med stöd vid beslut om arkitekturens uppbyggnad. Resultatet kan ses som en lösning på ett specifikt problem nämligen hur man skall komma fram till ett beslut i en viss fråga. Enligt litteraturen om forskningsmetoder [Easterby-Smith m. fl., 1991], skulle vår studie beröra tillämpad forskning. Omnämnt i litteraturen står också att det vid tillämpning av denna slags forskning är viktigt att förklara vad som händer, det är viktigt att vara kritisk till idéer och metoder som används samt att ta hänsyn till kvalitén på de bevis, som ges, som stöd för en idé. Den sistnämnda faktorn, hänsyn till kvalitén berör frågan om validitet diskuterat ovan. Slutligen nämns att denna form av forskning är vanlig på magisternivå, vilket är ytterligare ett argument till att använda densamma.

3 Teori

Teoriavsnittet har som syfte att tjäna som ett ramverk för de områden som vi berör i vår studie. Client/server-avsnittet syftar till att introducera läsaren till de grundläggande tankarna med denna arkitektur och berör också för- och nackdelar med arkitekturen. Förståelsen för client/server-arkitekturen är en nödvändig grund för att kunna tillgodogöra sig resonemanget om skilda typer av arkitekturen. Nästföljande avsnitt tar upp vad client/server-arkitekturen får för inverkan på ett system med fokus på konstruktionen, vilket är högst relevant med tanke på att detta är kärnan i vår uppsats. Skiktad arkitektur och olika designalternativ, har en stark anknytning till konceptet med tunna och feta klienter, varför det är relevant att belysa dessa aspekter. Hur man väljer att dela upp sin client/server-arkitektur påverkar hur klienter och servrar kommer att byggas. Viktigt är också att se sambanden mellan de olika begrepp som cirkulerar, att det inte är lösryckta avgränsade delar, utan att koncepten överlappar varandra. Vidare går vi djupare in på varje del i client/server-arkitekturen. Under rubriken klientsidan finns en beskrivning av tunna och tjocka klienter. Avsnittet serversidan går snabbt över några av de funktioner servrar kan ha samt vad som kan påverka valet av serverimplementation. Rubriken "Mellanvaran" behandlar den del av arkitekturen som möjliggör kommunikationen mellan de föregående begreppen, tyngdpunkten i detta avsnitt har vi lagt på ORBar och en egen del har avsatts åt standardarkitekturen CORBA, då dessa teknologier är subjektet för utvecklingen av våra tunna klientapplikationer.

3.1 Client/server-arkitektur

Client/server-arkitekturen är en modell för distribuerade system. Modellen visar hur data och processer fördelas mellan ett antal processorer [Sommerville, 1997] (se figur 1). Modellen fungerar som ett ramverk för uppbyggnaden av applikationer eller hela system, dvs hur komponenter skall fördelas mellan de olika processorerna.

Arkitekturen har tre huvuddelar; nämligen klienterna, nätverket och servrarna. Viktigt är att uppmärksamma att detta är en logisk modell. Klienterna och servrarna kan representeras av en eller flera fysiska enheter. Innebörden av det sistnämnda kan vara att klient och server körs på samma fysiska enhet, den logiska uppdelningen bevaras dock. Detta eliminerar behovet av nätverket för kommunikationen mellan klient och server [Sommerville, 1997].

Klienten eller klienterna är den del av arkitekturen som initierar kommunikationen med servern. Att hålla reda på vilken server det är som information skall begäras från tillhör också klientens uppgifter [Renaud, 1996].

Servern kan normalt sett inte påkalla kommunikation med klienten, däremot kan den själv fungera som en klient gentemot andra servrar (se nedan). Det finns dock fall där man använder sig av så kallad push-teknik, där servern initierar kommunikationen med klienten [Schettino & O'Hara, 1998]. Serverdelen tillhandahåller den information som en klient ställer förfrågan om. Funktionerna som servrar erbjuder är många, det kan röra sig om databashantering, utskriftshantering, applikationshantering mm. En och samma server är däremot funktionsspecifik, dvs de transaktioner som utförs är funktionellt relaterade, vilket dock inte hindrar en server från att begära service från en annan server för att slutföra sin uppgift. I en situation av detta slag behöver klienten endast kommunicera med en server, då kommunikationen med övriga inblandade servrar, för klienten görs transparent. Ytterligare

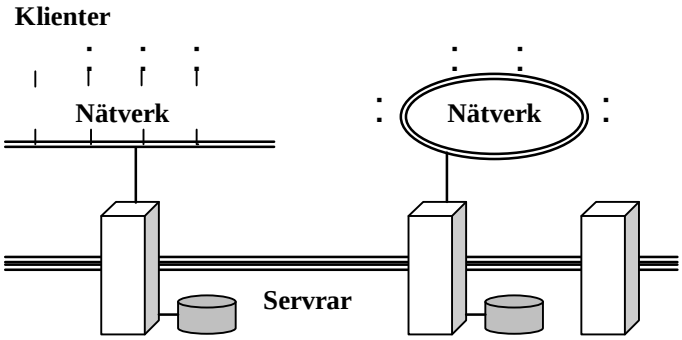
uppgifter som en server måste ta hand om är all dataadministration, så som backuper och återställande efter en krasch [Renaud, 1996].

Nätverket är länken mellan klienten och servern, möjliggöraren av kommunikationen mellan de båda delarna. I nätverket överförs data för behandling i respektive klient och server. Om dataöverföringen är intensiv kan detta vara en bidragande faktor till sämre prestanda i nätverket. Med snabbare nätverk är stora dataöverföringar dock inget problem. Utöver det fysiska nätverket behövs det en s.k. "mellanvara", som administrerar kommunikationen mellan klient och server över nätverket [Loosley & Douglas, 1998]. Mellanvara diskuteras mer ingående i avsnitt 3.5.

Som ovan nämnt hanterar klient och server olika processer, en fråga att ta ställning till vid modellering av arkitekturen är uppdelningen av dessa processer mellan klient och server. Valet står mellan att lägga huvuddelen av processerna på klienten, att balansera dem jämt eller att ge servern den största lasten. I "Introduction to client/server", Renaud. (1996) står uttryckt att äkta client/server kan man tala om, när processerna är jämt fördelade mellan klient och server.

Största fördelen med client/server-arkitektur är att utbyggnaden av ett system blir smidig. Det är lätt att lägga till nya servrar och integrera dessa med existerande system samt att uppgradera servrar utan att det får förödande effekt för övriga delar av systemet [Sommerville, 1997].

Svårigheterna med client/server-arkitekturen uppkommer motsägelsefullt nog, när man vill företa förändringar på existerande klienter och servrar, t ex vid en integration. Problemet är relaterat till det faktum att det inte finns någon gemensam datamodell; undersystem organiserar ofta sin data på olika sätt. Frånvaron av en gemensam referensdatamodell gör det svårt att förutspå de problem som kan uppstå när man skall integrera data, dvs man kan inte räkna ut exakt hur andra delar av ett system kommer att påverkas av förändringen [Sommerville, 1997]. Applikationer i client/server-miljö kräver dessutom en hög grad av administration. Ponera att man har begränsad åtkomst till servern, denna auktorisering av användare kan komma att behöva uppdateras. Vid en uppdatering av detta slag behöver man ändra auktoriseringslogiken för varje berörd klient. För att underlätta uppdateringen kan man tillämpa en datalös arbetsstation, likt de som används i UNIX-miljö. Alla applikationer ligger här på en filserver, dessa laddas ner varje gång applikationen startas, exekveringen sker sedan på klientdatorn. Utnyttjar användarna inte ett stort antal applikationer, anses detta som ett fördelaktigt alternativ. Största belastningen i en sådan lösning sker under uppstarten av applikationen, då nedladdningen över nätverket sker. Vad man vinner på detta tillvägagångssätt är att alla uppdateringar kan ske på ett enda ställe, dvs på filservern [Renaud, 1996].



Figur 1. Client/server-arkitektur i två skikt. Från *"High-Performance Client/Server"* av C. Loosley & F. Douglas, 1996, N.Y.: John Wiley & Sons, Inc.

3.2 Vad innebär användandet av client/server-arkitektur för design, konstruktion och systemets prestation?

Samtliga av dessa aktiviteter sägs bli mer krävande i client/server-miljö jämfört med traditionella centraliserade system [Loosley & Douglas, 1998]. Designen blir mer komplicerad p g a att distribuerade applikationer ofta har mer komplexa funktioner och byggs för flera slags komponenter. Mer data rör sig mellan skilda platser, vilket innebär att större vikt bör läggas vid planering för kapacitetsbehov samt vid frågor berörande tidsenlighet mellan datan. Ytterligare beslut som tillkommer är vilken hårdvara och mjukvara som krävs. I tillägg finns inte några fastslagna designmetoder som hjälp vid bedömningen, vilket innebär att designen kräver uppfinningsrikedom. Anledningen till tumreglernas frånvaro beror på att de flesta inte byggt tillräckligt många distribuerade system för att skapa sådana regler [Loosley & Douglas, 1998].

Att bedöma hur väl systemet kommer att nå upp till acceptabel prestationsnivå blir också svårare i en client/server-arkitektur. Detta påstående grundar sig på att systemet består av komponenter som interagerar på ett sätt som är svårt att förutse och på så sätt introduceras problem som man ej planerat för. Även denna svårighet är en följd av frånvaron av en utarbetad modelleringsmetod och riktmärken. När en applikation är distribuerad förändras arbetsbördan, även om man vet hur applikationen uppför sig när den kör i en centraliserad miljö, skiljer uppförandet sig en hel del i en distribuerad miljö, detta bidrar till att det blir än svårare att förutspå prestanda [Loosley & Douglas, 1998].

Som ett resultat av att designen kan innehålla komponenter som inte passar ihop, blir också konstruktionsfasen svårare. Även denna aktivitet påverkas av att de metoder som används är unga och i stor utsträckning obeprövade, med detta tillvägagångssätt ligger det nära tillhands att man oavsiktligt konstruerar ett system som inte är optimalt ur prestandasynpunkt [Loosley & Douglas, 1998].

Finns det då något sätt att motverka dessa svårigheter, i [Loosley & Douglas, 1998] ges några förslag. Vad man skall tänka på vid den logiska designen är att gruppera funktioner. Vid efterföljande fysiska design bör vikt läggas vid att placera ut funktionerna på lämpligt ställe i distributionen, dvs på olika processorer. Varje lager måste sedan designas för optimering, detsamma gäller alla länkar mellan lagren. Att designa de olika lagren rätt, är ytterst kritiskt, detta kommer sig av att fel kan ge upphov till extensiv kommunikation mellan lagren. M a o

är den grundläggande logiska designen den mest kritiska, fel som uppstår här blir svåra att rätta till ens med större beräknings- eller nätverkskraft.

3.3 Skiktad arkitektur

Betraktelsen av en applikation kan ske både på logisk och fysisk nivå. Med detta menas att man kan tala om olika skikt men det innebär inte nödvändigtvis att dessa existerar på olika fysiska enheter.

3.3.1 Logiska lager

Vad gäller logiken som är subjektet för uppdelningen i skikt, bör de existerande typerna också redogöras för. Dels omnämns presentationslogik, vilken håller skärmhanteringsfunktioner, applikationskontroll, datavalidering och felhantering på användarnivå, dessutom ombesörjer denna logik att SQL-satser presenteras i gränssnittskomponenter. Ytterligare en typ av logik är datalogiken som hanterar SQL-satser och transaktionslogik. Transaktionslogiken kan i sin tur förklaras med kontrollen över när databasen skall tillåtas uppdateras och därtill relaterade uppgifter. Väljer man att implementera en treskiktsarkitektur tillkommer ytterligare ett logiskt skikt nämligen applikationslogiken. Applikationslogikens uppgift är att fungera som kommunikatör mellan de båda ovanstående lagren samt att utföra beräkningar och datamanipulering. Oberoende av hur många logiska lager man väljer att tillämpa ligger samtliga lager mellan två hanterare; presentations- och databashanterare. Exempel på respektive är Windows 95 som presentationshanterare och Oracle som databashanterare. M a o är dessa lager redan färdiga för produktion, medan utvecklaren själv måste ta ansvar för att skapa de logiska lagren [Loosley & Douglas, 1998]. De logiska lagern kan betraktas som uppdelare av en applikations funktioner i tre områden, omnämnda hanterare kan också väljas att ses som logiska lager. Presentationshanteraren hanterar all grafisk layout medan databashanteraren handhar lagring, hämtning, administration och återskapande av all persistensdata.

3.3.2 Varierande uppdelningar

Vanligt i mindre applikationer är tvåskiktsarkitektur, här är all data i client/server-arkitekturen förlagd till databasservern. Centraliserad data är lämpligt om stora delar av datan uppdateras eller manipuleras ofta [Renaud, 1996]. Vid konstruktion av tvåskiktsarkitektur måste utvecklaren ta hänsyn till gränssnittet mellan presentations- och datalogik.

Treskiktsarkitektur passar bra i sammanhang med distribuerad data. I större client/server-system lämpar det sig med denna arkitektur. Här har man en applikationsserver som hanterar transaktionslogiken och slussar meddelanden rätt till aktuell databasservern [Renaud, 1996]. Treskiktsarkitekturens struktur är mer komplicerad än tvåskiktsarkitekturen. Som ovan påpekat är det lämpligt att lägga till ett tredje lager när det finns flera användargränssnitt som kommunicerar med flera databaser. Det extra lagret har till syfte att sköta kommunikationen mellan presentations- och datalogiken. Handhavandet av kommunikationen sker på så sätt att lagret tar emot transaktioner från presentationslogiken och bryter ner dem i lokala komponenter som skickas vidare till lämpligt datalogiskt lager. En aspekt som gör denna struktur än mer komplex är att databasen oftast är distribuerad, vilket gör att hänsyn måste tas till hur man skall nå konsistens i datavärden bland duplicerad eller relaterad data [Loosley & Douglas, 1998]. Ett alternativt namn på client/server-arkitektur i treskikt är företagssystem, detta kommer sig av att system som omfattar ett företag i helhet ofta består av flera mångfacetterade delar som har behov av att kommunicera sinsemellan.

3.3.3 Designalternativ

Talar man om tvåskiktsarkitektur finns det tre fysiska designalternativ att välja mellan [Loosley & Douglas, 1998]:

1. **”Avlägsen presentation”** – Både presentations- och datalogik finns på servern. Presentationshanteraren finns kvar på klienten, dess uppgift är att handha skärm kontroll funktioner.

2. **”Avlägsen datatillgång”** – Kan sägas vara motsatsen till ovanstående design. Här ligger presentationslogik och datalogik på klienten.

3. **Distribuerad logik** – Presentationslogiken är lokaliserad på klienten och datalogiken på servern.

Val av det första designalternativet, dvs placering av all logik på servern, är positivt ur både konstruktions- och administrationshänseende. Sistnämnda påstående kommer sig av att all kod är placerad på en central enhet. Nackdelen är dock att prestandan blir sämre, vilket är ett resultat av att nätverket är tungt belastat av kommunikationen mellan presentationshanteraren och presentationslogiken. Även servern tar en tung last, som resultat av att den kör både presentations- och datalogik. Väljer man däremot att lägga all logik på klienten innebär också den här designen att prestandan sjunker, detta kommer sig av att nätverket belastas med SQL - anrop mellan datalogiken på klienten och datahanteraren på servern. Försämrad integritet är ytterligare ett problem som tillkommer i denna design. Transaktioner kodas på klienten, som är mindre säker än servern, vilket ger ökad osäkerhet vid uppdateringar i databasen. Designen ifråga lämpar sig bäst för ”läs” applikationer, dvs applikationer som endast hämtar data, med tanke på ovannämnda integritetsproblem. Distribuerad logik proklameras vara det bästa alternativet. Nätverkstrafiken är relativt låg, eftersom varje meddelande mellan klient och server är en hel transaktion. Serverns belastning blir också lägre, då presentationslogiken har förflyttats till klienten, följaktligen blir vinner man också integritet genom att hantera datan på servern som är säkrare. Med distribuerad logik slipper man dock inte undan alla problem; utveckling och administration av logiken delas upp, hanteringen av dessa aktiviteter blir m a o mer kostsam [Loosley & Douglas, 1998].

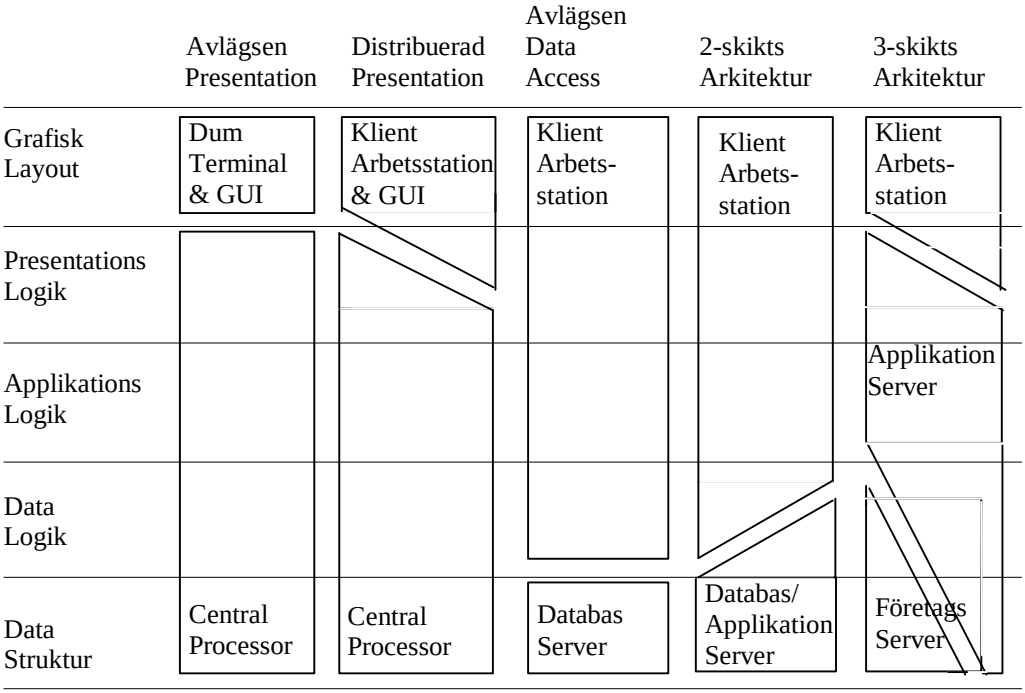
I treskiktsarkitektur anses de logiska lagren vara samtliga fem ovannämnda (se logiska lager). Klienten förutsätts bära presentationshanteraren och logiken, applikationslogiken finns på en separat server medan ytterligare en eller flera servrar tar hand om datalogik och databashanterare [Loosley & Douglas, 1998]. Andra uppdelningar av de logiska lagren är också möjliga.

För att röra till begreppen ytterligare finns det fler definitioner på hur man kan välja att dela upp logiken i client/server-miljön. [Mathiassen, Munk-Madsen, Nielsen & Stage, 1998] har givit ett förslag som till vissa delar följer ovanstående uppdelning av logik men som tar uppdelningen ett steg längre. I denna modell har man inte bara valt att dela upp olika slags logiker utan även en och samma logik mellan klient och server. Arkitekturerna skiljer sig liksom ovan i bemärkelsen av hur distributionen av logik företas. Systemet betraktas av Mathiassen som bestående av tre komponenter; användargränssnitt, funktion och modell, vilka motsvarar de logiska lagren; presentation, applikation och data. Genom att på olika sätt kombinera dessa utvinns skilda arkitekturer. Vad som skiljer sig mot modellen i Loosley & Douglas (se ovan) är distribuerad presentation där presentationslogiken delas upp, distribuerad data men även distribuerad funktion.

Klient	*Server*	*Arkitektur*
Användargränssnitt	Användargränssnitt + Funktion + Modell	Distribuerad presentation
Användargränssnitt	Funktion + Modell	Lokal presentation
Användargränssnitt + Funktion	Modell	Centraliserad data
Användargränssnitt + Funktion	Funktion + Modell	Distribuerad funktion
Användargränssnitt + Funktion + Modell	Modell	Distribuerad data

Figur 2. Modell för distribuerade applikationer. Från *"Objektorienterad analys och design"* av L. Mathiassen A. Munk-Madsen, P.A. Nielsen & J. Stage, 1998, Lund: Studentlitteratur.

Nedan ges en sammanställning över de distribuerade modeller som vi gått igenom i aktuellt avsnitt. Det kan diskuteras huruvida avlägsen presentation skall rymmas inom definitionen för client/server-arkitektur men vi väljer att hänvisa till [Loosley & Douglas, 1998] där konceptet ryms inom definitionen.



Figur 3. Olika distribueringsmodeller. Från *"High-Performance Client/Server"* av C. Loosley & F. Douglas, 1996, N.Y.: John Wiley & Sons, Inc.

3.4 Klientsidan

Beroende på var man väljer att placera logiklagren kommer klientens benämning att skifta. Som ovan påvisat finns en mängd kombinationer.

3.4.1 Tjocka klienter

Väljer man en design ovan där både presentationslogik och applikationslogik placeras på klienten har man siktat in sig på att implementera en tjock klient. Modellen följer m a o tanken om decentraliserad ”beräkning”. Varje användare har sin egen fristående dator med full funktionalitet, och servern fyller bara en funktion som behållare för data. Vad som finns på servern är m a o ett DBMS, dvs databashanteringsystemet. Huvuddelen av applikationerna placeras alltså på klienten medan servern är tunn (se nedan under Serversidan) och följaktligen har minimalt hård- och mjukvarukrav. Klienten kräver däremot å sin sida mycket processorkraft. Har man på detta sätt, applikationer på vardera klient krävs att man har regler

för hur man organiserar systemet, bl.a. rekommenderas standardkonfiguration för att bevara uppdateringar konsistenta i samtliga klienter. Alla klienter bör konstrueras på samma sätt, de enda variationer som bör förekomma är de som är en följd av klientens specifika roll, t ex funktion eller typ av operativsystem [Renaud, 1996]. En tjock klient är inte totalt beroende av en central dator för att kunna köra, vilket gör att den inte blir lika känslig för ett avbrott i centraldatorn. Exempel på tillämpningar där tjocka klienter används är mobila enheter där man kanske inte vill ha en konstant uppkoppling mot servern utan vill kunna ha funktionalitet ändå, vidare sägs tunga applikationer såsom spel och cad/cam vara lämpade att placeras på klienten [Danielsson, 1998].

3.4.2 Tunna klienter

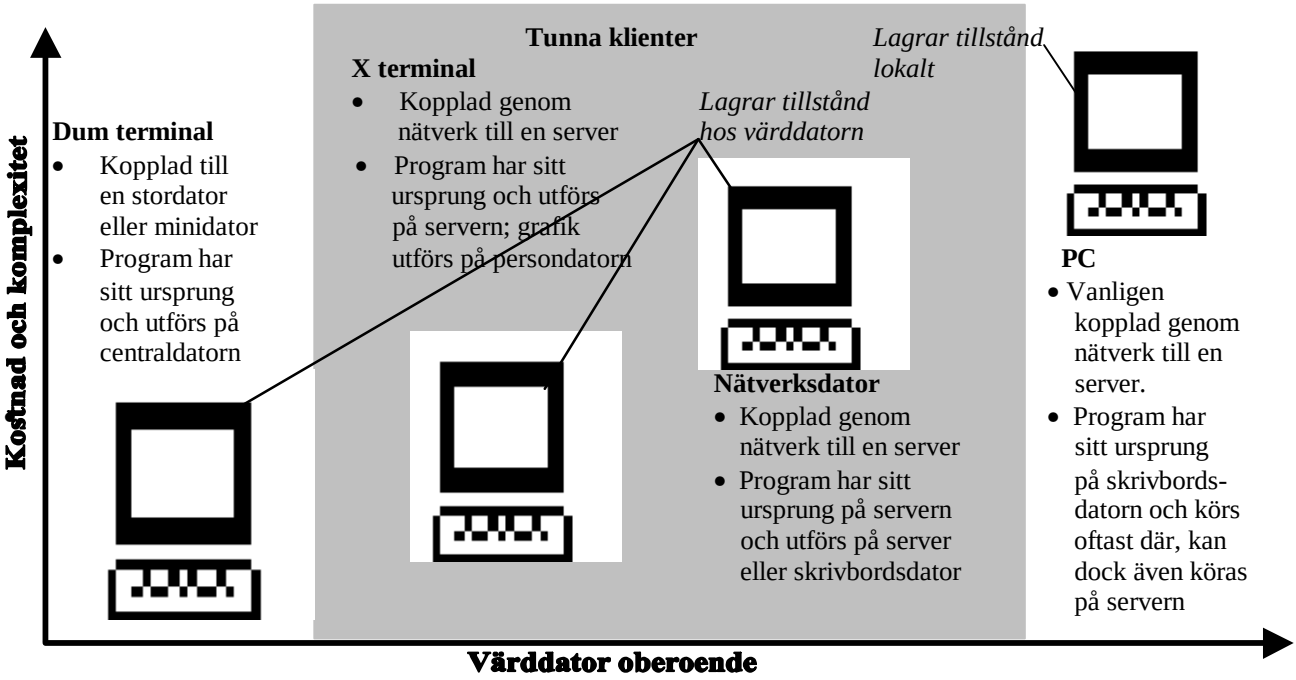
Det finns ingen enhetlig definition på begreppet tunna klienter, utan åsikterna om vad som skall definieras som en tunn klient går isär. Tunna klienter kan istället ses spänna över ett brett spektrum. Vad som är gemensamt för samtliga tunna klienter är att de är designade för att administreras centralt, vilket innebär att man, om man har en dator som endast implementerar tunna klientapplikationer kan avlägsna cd-rom- och diskettenheter på denna. Idén är att begränsa förmågan hos klienten genom att endast inkludera de viktigaste funktionerna, det är också därav namnet tunn klient kommer, med avseende på klientapplikationens ringa omfattning. I en arkitektur som endast tillämpar tunna klientapplikationer krävs endast en dator med begränsad lagringskapacitet och med ett minimalt operativsystem [http://www.whatis.com 1999-04-08]. En direkt effekt av användandet av tunna klienter, är lägre kostnader, eftersom mycket utrustning har utelämnats på klientdatorn, dessutom underlättas administrationen då den utförs på ett och samma ställe, dvs servern.

Ett exempel på när det är lämpligt att välja en tunn klientarkitektur är när antalet samtidiga databasuppkopplingar är stort. Prestandan kan förbättras med hjälp av en applikationsserver. Applikationsservern kan också bidra till förenklad kommunikation, genom att den kan isolera protokollen, som används som koppling till databasservern [Loosley & Douglas, 1998]. Argument som talar mot en applikationsserver är att den skapar ytterligare ett ställe där det kan gå fel, då den utgör en extra instans som transaktionerna måste passera. Utvecklingen blir dessutom mer komplicerad, då man har ytterligare en plattform som skall testas och implementeras [Renaud, 1996].

Begreppet supertunna klienter förekommer också, vilket tydligare visar på att det finns flera varianter av tunna klienter. En klient som endast handhar presentationshanteraren, kan kallas supertunn. Användargränssnitt och applikationslogik kan laddas ner under exekveringen av applikationen, eller istället väljas att köras på servern.

Exempel på tillämpningar som kan använda tunna klienter är e-post, datainmatning samt informationssökning på internet, t ex sökmotorer som Altavista [Danielsson, 1998].

TUNNA OCH TJOCKA KLIENTER



Figur 4. Olika slags klienter. Från *www.byte.com/art/9704/img/047csh2.htm 1999-04-09*

3.5 Mellanvara/Kommunikationsteknologier

När man delar upp en applikation och placerar olika delar på skilda maskiner måste dessa kunna kommunicera. Termen mellanvara betecknar alla mjukvarukomponenter som möjliggör denna kommunikation. En alternativ benämning på mellanvara är applikationsnivåprotokoll. Ovanstående fysiska modeller kräver olika slags mellanvaror, dvs man får välja mellanvara utifrån vilken skiktad arkitektur man beslutar att implementera och följaktligen vilken slags kommunikation som företas i aktuell arkitektur (För vidare läsning se Loosley & Douglas, 1998). En typ av mellanvara är gateways, som kontrollerar trafiken mellan datorer inom ett nätverk (en server som fyller funktionen som gateway fungerar ofta också som en proxyserver, vilken ”uppfyller” säkerheten i nätverket, exempel på detta är t ex en brandvägg). Vidare finns; object request brokers, dvs ORBar (se nedan under applikationsmellanvaror) och köhanterare, där meddelanden i form av processer eller objekt placeras i en kö, målprocessen hämtar upp meddelandet när den kan enligt någon av modellerna FIFO och LIFO (First In First Out, Last In First Out). Ytterligare mellanvaror är transaktionsövervakare, där klienterna besparas jobbet att formulera förfrågningar till okända databaser och transaktionsmonitorer, som bevakar klienterna efter förfrågningar och vidarbefordrar dessa till databasservern. Mellanvaran kan sägas vara kärnan i client/server-applikationen, då den bistår med språket som både klient och server kan förstå. Några av de existerande mellanvarorna har också som syfte att gömma alla detaljer som lågnivånätverksprotokoll innebär för programmeraren [Loosley & Douglas, 1998], exempel på sådan mellanvara är CORBA och RMI.

Sättet på vilket komponenterna kommunicerar kan ha stort inflytande på applikationens prestanda. Beslutet om vilka kommunikationssätt som bör användas styrs bl.a. av aspekter som hur stor kapacitet servern har och hur stor klientpopulationen är [Loosley & Douglas,

1998]. Mellanvaran kan bidra med varierade typer av service; direktionsservice, där klientförfrågan visas till rätt server, säkerhetsservice, som hindrar klienter som inte har tillåtelse från att koppla upp sig till en viss server, meddelandebyggnad; klientens förfrågan packas till ett meddelande för vidaresändning till servern men packas också upp när svaret kommer tillbaka. Ytterligare en service är översättning av tecken för att stödja samarbete mellan klient- och serverplattformar som har olika teckenuppsättningar [Loosley & Douglas, 1998].

3.5.1 Typer av mellanvara

Tre typer av mellanvara finns för att stödja de logiska uppdelningarna i ovanstående avsnitt. Mellanvaran distribuerar m a o applikationerna på olika nivåer. Varje typ har en motsvarighet i ett logiskt lager, benämningen blir därför; presentationsmellanvara, applikationsmellanvara och databasmellanvara. Exempel på en mellanvara av slaget presentation är en web-baserad applikation; webbrowsern på klienten, http protokollet och webservern fungerar tillsammans som en presentations mellanvara. Så gott som alla tvåskiktsarkitekturer är implementerade med databasmellanvara. Denna typ av applikation står också i relation till feta klienter (se avsnitt 3.4.1 ovan). Den feta klientapplikationen initierar SQL-förfrågningar till ett DBMS, databasmellanvarans uppgift är här att hantera dessa förfrågningar genom att förflytta dessa genom nätverket till DBMSet och returnera svaret till applikationen. Applikationsmellanvaran förekommer i sin tur i skilda varianter; punkt-till-punkt eller direktmeddelande, RPC som står för remote procedure call, meddelande kö, objektförfrågningsmäklare (ORB) samt distribuerad transaktionsmonitor. Applikationsmellanvaran skiljer sig från de båda andra mellanvarorna vad gäller flexibilitet. De andra mellanvarorna är skrivna för att kommunikation skall kunna realiseras mellan en användardefinierad applikation och en standardapplikation, utrymmet för egen design är minimal medan applikationsmellanvaran är gjord för kommunikation mellan två användardefinierade komponenter och har egenskaper som ett generellt programspråk. Vad som behövs tas ställning till är kommunikationssättet [Loosley & Douglas, 1998].

3.5.2 Databasmellanvara

Exempel på databasmellanvara är ODBC, Open Database Connectivity som är framtaget av Microsoft, JDBC som står för Java Database Connectivity och är Javas svar på ODBC.

3.5.2.1 JDBC

JDBC är en industristandard för databasoberoende koppling mellan Java och olika databashanterare (klasserna för att hantera JDBC ligger i java.sql). JDBC låter en Java-applikation programmeras mot dess generella gränssnitt, och JDBC-drivrutiner översätter sedan anropen mot JDBC till anrop mot den installerade databasproduktens API. Det krävs att databasen som man använder sig av stödjer ANSI SQL-2 standarden [Eriksson, 1997]. Fördelen med att använda sig av JDBC är att applikationen blir oberoende av vilken databasleverantör som valts. Uppgifter som JDBC sköter är upp- och nedkoppling av förbindelser, exekvering och hantering av resultat från SQL-satser samt metadata om databasen. Det finns JDBC drivrutiner som är skrivna helt i Java och därför inte kräver någon installation på klientdatorn utan kan laddas ner över Internet för användning i t ex en applet [Eriksson, 1997].

3.5.3 Applikationsmellanvara

I treskiktsarkitekturer är det applikationsmellanvaran som är aktuell att använda. Utbudet är brett, varför vi valt att gå lite mer in på denna typ av mellanvara än de övriga två. De mest

använda typerna är punkt-till-punkt meddelanden, RPC och objekt meddelanden [Loosley & Douglas, 1998].

3.5.3.1 Punkt-till-punkt meddelande

Denna typ av mellanvara förmedlar meddelanden mellan två parter som är förbundna. Den kan användas till att bygga flera slags distribuerade applikationer, där komponenter kan ha rollen av klient, server eller samarbetande punkter. Att bygga applikationer m h a denna teknologi är komplext [Loosley & Douglas, 1998].

3.5.3.2 Remote procedure call

RPC mekanismen låter ett program kalla på en procedur som är placerad på en avlägset belägen dator på samma sätt som programmet skulle ha kallat på en lokal procedur. RPC är tidkrävande då denna teknologi sätter igång ett stort antal processer på servern [Loosley & Douglas, 1998].

3.5.3.3 Objektmeddelanden

När man hanterar meddelanden mellan objekt krävs det en annan typ av mellanvara än vid förmedling mellan distribuerade program, ett exempel på en sådan vara är en s.k. ORB (Object Request Broker). Det finns flera typer av ORB teknologier; CORBA, en standard arkitektur för ORBar, vilken behandlas i mer detalj i nedanstående avsnitt, COM/DCOM (Component Object Model) för PC miljö samt RMI (Remote Method Invocation) som är Javaspecifik [www.sei.cmu.edu/str/descriptions/orb_body.html, 1999-04-08]. Används en ORB som följer standarden kan objekt kalla på varandras service oberoende av objektspråk, verktyg, plattform och placering. ORBens uppgift är att tolka ett objekts förfrågan, hitta ett objekt som kan svara på frågan samt returnera svaret till objektet som ställt frågan. Det faktum att ORBar hanterar meddelanden mellan objekt, resulterar i ökad trafik i nätverket. I kombination med att ORBar ofta byggs på RPC innebär detta dessutom extra tryck på nätverket, vilket ger än sämre prestanda. Jämför med de båda tidigare nämnda teknologierna är ORBar mer komplexa. Fördelen som gör att teknologin används trots prestandaförsämring är det stöd som ges för möjliggörande av kommunikation mellan olika språk och plattformar [Loosley & Douglas, 1998].

3.5.4 CORBA

CORBA är som ovan nämnt en specifikation för en standardarkitektur för objekt förfrågnings mäklare. CORBA specifikationen är en industristandard skapad av OMG (Object Management Group), specifikationen har varit i utveckling sedan 1992. Direkt stöd för objektorienterad design erbjuds genom CORBA. De flesta CORBA ORBar bygger på C++ som huvudsakliga klient och server miljön, men även JAVA baserade ORBar börjar blir vanligare [www.sei.cmu.edu/str/descriptions/corba_body.html, 1999-04-08]. Specifikationen kombineras med s.k. OMA (Object Management Architecture), som också är en specifikation. OMA definierar olika service för design av distribuerade applikationer, vilka delas in i tre kategorier:

- CORBA Service - Är grundstenen för att kunna bygga mer komplexa distribuerade applikationer. Innehåller asynkron händelsehantering, som innebär att flera förfrågningar kan ske utan att en klient behöver invänta svar, transaktionshantering, persistens, parallellitet och namngivning.

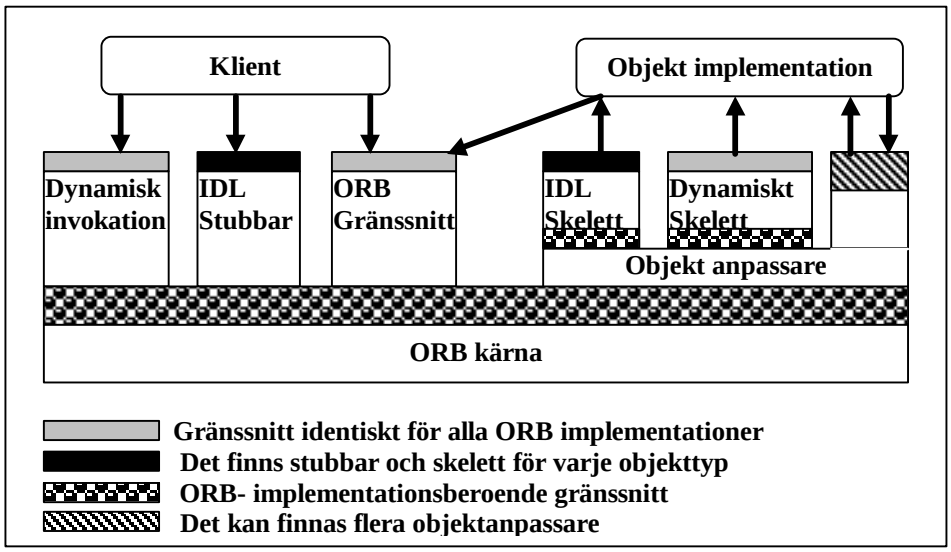
- CORBA Faciliteter - Kan användas för vissa distribuerade applikationer, vilka kan vara användargränssnitt, informationshantering, systemhantering och uppdatering.
- Applikationsobjekt - Ger service som är specifik för en applikation eller klass av applikationer.

För att förstå CORBA är det viktigt att uppmärksamma IDL (Interface Definition Language) processorn. Objekt definieras i CORBA m h a IDLen, som är en objektorienterad gränssnitts formalism. IDL har syntaktiska likheter med programspråket C++ och har endast funktionen som definitions gränssnitt, m a o kan man inte specificera händelser m h a IDLen.

Viktigt att uppmärksamma är att CORBA specificerar att klient och objekt kan implementeras i olika programspråk och exekveras på olika maskinhårdvara och olika operativsystem. Klient respektive objekt kan inte upptäcka några av ovanstående detaljer hos varandra. IDL gränssnittet bidrar helt med det gränssnitt som behövs mellan klient och objekt.

CORBA har ett antal komponenter [www.sei.cmu.edu/str/descriptions/corba_body.html, 1999-04-08], huvuddelen av dessa är:

- ORB kärnan - CORBA körnings infrastruktur, definieras av återförsäljaren.
- ORB gränssnitt - Standard gränssnitt till alla ORBar, definieras av IDL
- IDL Stubbar - Generas av IDL processorn för varje IDL gränssnitt. Gömmer nätverksdetaljer på låg nivå för klienten.
- Dynamisk invokation - Ett alternativt sätt för klienter att nå objekt. Tillhandahåller en generisk mekanism för att skapa förfrågningar under körtid.
- Objekt anpassare - Kan utvecklas för att ge tillgång till objekt som lagras i en avlägsen objektdatabas. Alla ORBar som är anpassade till CORBA skall stödja en specifik objekt anpassare som kallas BOA (Basic Object Adaptor).
- IDL Skelett - Serversidans motsvarighet till stubbar. IDL skelett tar emot förfrågan från objekt anpassaren och kallar på lämpliga operationer i objektimplementationen.
- Dynamiskt Skelett - Serversidans motsvarighet till dynamisk invokation. Medan IDL skelettet initierar specifika operationer i objektimplementationen överlåter det dynamiska skelettet denna process åt objektimplementationen. Skelettet är användbart för att utveckla mekanismer för operationer mellan olika ORBar.



Figur 5. CORBAs gränssnitts struktur. Från *www.sei.cmu.edu/str/descriptions/corba_body.html* 1999-04-12.

CORBA anses komplext [www.sei.cmu.edu/str/descriptions/corba_body.html, 1999-04-08]. Komplexiteten kommer sig av att flera återförsäljare utvecklar ORBar med olika egenskaper och möjligheter, vilket innebär att användaren måste lära sig dessa specifika funktioner för att kunna dra full nytta av teknologin. Dessutom kräver specifikationen expertiskunskap i relaterade teknologier såsom, distribuerad systemdesign, distribuerad och multitrådad programmering, objektorienterad design, analys och programmering. Svårigheterna med att bygga robusta distribuerade system kvarstår trots att CORBA bidrar till att underlätta utvecklingen av distribuerade applikationer. I tillägg har det faktum att CORBA-specifikationen ändrats kontinuerligt resulterat i att utvecklade ORBar blivit instabila.

3.6 Serversidan

Hur man väljer att designa klienterna kommer att påverka hur servern ser ut och vice versa. Väljer man en tunn klient måste servern exempelvis bära en tyngre börda. Serverns kapacitet måste då höjas för att klara den ökade belastningen, alternativt sätter man upp fler servrar för att tillgodose klienternas behov av service. Väljer man däremot att bygga feta klienter kommer servern att behöva bistå med mindre kapacitet och har därför mindre krav på hårdvara och operativsystem.

3.6.1 Databasserver

En databasserver använder en klientdator för användargränssnitt och applikationslogikdelarna av en applikation medan servern bistår med datahanteringsdelen av applikationen. Definitionen enligt [Loosley & Douglas, 1998] är; "kombinationen av DBMS-mjukvara och den hårdvaruplattform som mjukvaran finns på"[4].

3.6.2 Applikationsserver

En definition på applikationsservern är att den använder PCn bara för användargränssnittet, medan servern används för applikationslogik och datahantering [Tseng, 1999]. När servern håller samtliga affärslogikfunktioner samt databashanteringen benämns den, tjock server. En sådan konstruktion är aktuell vid tvåskiktslösningar. Vid val av en serverarkitektur av denna

[4] Fri översättning Loosley, Douglas s713

typ kan tillgången till servern bli trög och svarstiderna försämrade [www.vt.edu:10021/M/mpriddy/term.html, 1999-04-09]. Ytterligare definition innebär att applikationsservern bara håller applikationslogik och dirigerar förfrågningar vidare till ytterligare server för datahantering, dvs servern agerar som en klient gentemot andra servrar [www.vt.edu:10021/M/mpriddy/term.html, 1999-04-09]. Används applikationsservern enligt sistnämnda definition innebär det samtidigt att en treskiktslösning tillämpas (se ovan under "skiktad arkitektur").

3.6.3 Tunna servrar

Senaste trenden inom serverområdet är s.k. tunna servrar. Serverns uppgift är att ge vilken utrustning som helst nätverkskapacitet. Teknologin kan appliceras på en mängd applikationer; skrivare, scanners , CD-ROM och hårddiskar kan kopplas samman och delas där de behövs, vilket minskar nätverkstrafiken. Anledningen till att nätverkstrafiken minskar är att utrustningen som innehåller servern pratar direkt med klienten. Fler områden är konsumentprodukter, som uppvärmning och säkerhetssystem, vilka m h a teknologin kan kontrolleras från valfri plats samt nätverks hårdvara, t ex hubbar, som kan styras genom ett webgränssnitt [www.axis.com, 1999-04-09]. En tunn server kan liknas vid en tunn klient i den bemärkelsen att båda teknologierna kan ses som datorer med en specifik uppgift, med begränsad lagringskapacitet och ett minimalt operativsystem [www.whatis.com/thinserv.html, 1999-04-09]. Ett bra exempel på en teknik som möjliggör en konstellation av tunna servrar är Jini arkitekturen[5].

[5] www.whatis.com 1999-04-26

4 Resultat

Här följer i kronologisk ordning resultatet av de metoder vi tillämpade. Genom litteraturstudien har det framkommit ett antal faktorer där åsikter och rent konkreta aspekter för tunna respektive tjocka klienter skiljer sig åt. Under intervjuavsnittet finns en redogörelse för de intressen som kan vara aktuella att ta hänsyn till vid konstruktion av en applikation. Avsnitten om val av applikation, utvecklingsverktyg samt uppdelning av arkitektur syftar till att visa relevansen av dessa val i förhållande till frågeställningen samt varför man har fattat ett visst beslut. Genom redogörelse för prototypen vill vi förmedla en bild av det objekt som varit fokus för vår jämförelse. Med hjälp av resultatdelen; erfarenheter från utvecklingsarbetet visas slutligen de skillnader i utvecklingen som de båda klienttyperna innebar.

4.1 Litteraturstudie; jämförelse mellan tunna och tjocka klienter

Problemformuleringen kräver svar på frågan hur de olika klientarkitekturerna skiljer sig åt vad gäller inverkan på en client/server-applikation. Genom att betrakta skrivelser på området kan man få hjälp att upptäcka dessa skillnader, samt få tillgång till en referensram för jämförelse mot egna erfarenheter. Åsikterna om man skall använda tunna eller tjocka klienter går vitt isär. Visst är det så att det finns argument till varför man bör använda den ena tekniken framför den andra men långt ifrån övervägande delen av dessa argument är baserade på objektiva eller kvantitativa fakta. Man får en föraning om att det kanske inte finns något rakt svar på vad som är bättre eller sämre utan att det är situationen som styr lämpligheten. Vi har gjort en ansats till att dela upp argumenten i skilda områden för att skapa en struktur i jämförelsen.

4.1.1 Uppbyggnad

Som nämnt i teoriavsnittet går t o m åsikterna om vad som är en tunn och tjock klient isär. Linjen för en tunn klient kan sägas gå vid en klient innehållande på sin höjd en liten del av applikationslogiken medan en tjock klient innehåller all logik utom databashanteringen. Tunna klienter varierar sedan i tjocklek [Loosley & Douglas, 1998].

4.1.2 Konstruktion

Vid konstruktion av en tjock klient behövs oftast endast hänsyn tas till utvecklingen av klienten och servern då den vanligaste konstruktionen är tvåskiktsarkitektur. Visserligen behövs det mellanvara men denna är väl utvecklad speciellt när det gäller kommunikation mellan applikationslagret och databaslagret som är aktuell när man talar om en tjock klient. Kommunikationen realiseras med hjälp av standardiserade teknologier såsom JDBC, vilket inte skiljer sig i stor utsträckning från övriga teknologier som används vid utvecklingen av klient och server. M a o behöver utvecklaren inte besitta några andra kunskaper än det programspråk som ändå måste till för att kunna bygga klient respektive server. Vad gäller konstruktion av en tunn klient, krävs däremot extra tekniska kunskaper, p g a den kommunikationsteknologi som behöver implementeras för kommunikation mellan klient och applikationslager, den s k applikationsmellanvaran. Aktuella teknologier skiljer sig från de som krävs för implementation av klient och server. I tillägg skapar detta fler frågor vid designen, då man måste ta ställning vilken sorts kommunikationsteknologi som är lämplig och det finns ett omfattande utbud (se ovan under "Mellanvara/kommunikationsteknologier") [Loosley & Douglas, 1998].

4.1.3 Flexibilitet

Med en tjock klient finns inte stora möjligheter till att påverka hur kommunikationen skall ske eftersom standarderna redan är satta av återförsäljare av de standardprogramvaror som skall till för kommunikation mellan klient och server. Valfriheten med en tunn klient är desto större, p g a de många möjligheterna som mellanvarorna inbjuder till. Teknologierna är till skillnad från de som används vid konstruktion av tjocka klienter, inte fördefinierade utan måste definieras av programmeraren själv, som därmed får fritt spelrum. Väljer man en tunn klient med ett applikationsskikt blir det lättare att bygga ut systemet med fler klienter än det är om man har en arkitektur i två skikt, men med tanke på att även tunna klienter kan byggas i två skikt är detta inte ett direkt resultat av valet att implementera en tunn klient [Loosley & Douglas, 1998]. Ytterligare bidrag som man får med en tunn klient är förmågan att ändra applikationslogiken utan att behöva ändra på användargränssnittet [www.vt.edu:10021/M/mpriddy/term.html, 1999-04-09].

4.1.4 Reabilitet

Hur känslig är den tunna respektive tjocka klienten för ett avbrott i servern. Skulle servern gå ner när arkitekturen är uppbyggd av tunna klienter kommer samtliga klienter att påverkas vilket gör att hela strukturen får avbrott, vilket i sin tur kan åsamka stora kostnader. Med en tjock klient blir beroendet av servern inte lika påtagligt för systemet som helhet, skulle servern krångla har klienten ändå tillräcklig förmåga för att fortsätta med flera uppgifter, dvs de som inte berör upphämtning av data [www.yougeek.com/rants/fatrant.html, 1999-04-09]. Byggs en tunn klient dessutom med en treskiktsarkitektur, vilket är en vanlig företeelse skapas ett extra lager i vilket fel kan uppstå [Renaud, 1996]. Av detta följer även att extra kontroll måste skapas vid uppdateringar i databas på flera nivåer i arkitekturen, vilket också bidrar till extra komplexitet (se ovanstående avsnitt om konstruktion). En annan fråga som berör reabiliteten där en tjock klient är en nackdel är vid uppdatering av applikations och datalogik. Risken finns, när antalet klienter är många att inkonsistens uppstår då man kan missa att uppgradera någon enhet. Sker administrationen däremot centralt på ett och samma ställe, vilket tunn klientarkitektur understödjer, undviks olyckor av ovanstående slag och reabilitet och konsistens i datan behålls [www.sei.cmu.edu/str/descriptions/clientserver_body.html, 1999-04-08].

4.1.5 Säkerhet

En fet klient anses mer sårbar än en tunn klient, då det inte krävs åtkomst till servern för att få del av större delen av logiken [www.yougeek.com/rants/rfatrant.html, 1999-04-09]. I en tunn klient sker extra kontroll för åtkomst till server och därmed också huvuddelen av logiken. Det finns också motsatta åsikter som menar att en tjock klient kan hålla mycket i minnet som användaren inte nödvändigtvis behöver få tillgång till, kontrollen av detta antas ske genom extra inloggningsrutiner [www.yougeek.com/rants/fatrant.html, 1999-04-09].

4.1.6 Prestanda

I [Renaud, 1996] ges rekommendationen att lägga större delen av processerna på klienten, dvs skapa en tjock klient eftersom servern riskerar att bli för trög om den innehåller mycket logik och användarantalet är stort. Detta anses i synnerhet tillskrivas applikationer med många interaktiva processer, dvs där det sker mycket kommunikation mellan de logiska lagren. Lägger man då största delen av logiken på klienten minskar nätverkstrafiken, eftersom större delen av kommunikationen sker internt hos klienten. En siffra på över 50 samtidiga databassessioner ges som tumregel för att införa ett applikationsskikt, vilket m a o skulle innebära en förflyttning av applikationslogiken från klient till server och därmed bidra till en

tunnare variant av klient [Renaud, 1996]. Anhängare av tjocka klienter menar att dessa kommer att dominera arkitekturen trots att de kräver mer kraft, det är bara en fråga om att tillgången till processorkraft skall bli större [www.yougeek.com/rants/fatrant.html 1999-04-09]. Tunna klienter anses öka nätverkstrafiken då den pratar mycket med servern, detta bidrar till längre svarstider och en trögare process. Utöver ökad nätverkstrafik bygger många tunna klienter på kommunikationsteknologier som också saktar ner kommunikationen genom att kräva synkron överföring, dvs att klient måste invänta svar innan den skickar en ny förfrågan. Exempel på dessa teknologier är RPC och ORB som byggs ovanpå RPC och därför bidrar till ytterligare prestanda förlust [Loosley & Douglas, 1998]. Även tjocka klienter anses ha prestanda problem [www.vt.edu:10021/M/mpriddy/term.html, 1999-04-09] p g a den mängd kod som hanteras, vilket också skapar längre svarstider och problem när det gäller tillgången till data om datamängden är stor.

4.1.7 Administration av applikationen

Använder man sig av feta klienter måste administrationen ske på varje klient, p g a att den logik som berörs av uppdateringar ligger just på varje klient. Med en tunn klient däremot, där man har större delen av logiken förlagd till en server, samordnas administrationen och sker på ett enda ställe och på samma gång.

4.1.8 Utrustnings krav

Feta klienter kräver mer hårdvara då processerna som skall köras är många, m a o finns större behov av exempelvis minne. Vidare krävs ett omfattande operativsystem, diskettenhet och cd-romenhet för att kunna administrera varje klient. Tunna klienter gör inte stora anspråk på minne då största delen av processerna sker på servern. Även logiken finns lagrad på servern vilket också bidrar till minskat hårdvarukrav, diskettenheter och övriga hjälpmedel för uppdateringar kan också avlägsnas, tack vare att administrationen kan ske på ett ställe och operativsystemet kan begränsas, vilket också är ett resultat av att processerna förflyttas bort från klientdatorerna [hcgl1.eng.ohio-state.edu/~tsengh/gs637lt1.html, 1999-04-09].

4.1.9 Kostnader

Vanligast förekommande argument för att använda tunna klienter är att kostnaden sjunker drastiskt. Påståendet kommer sig delvis av ovanstående faktorer; att varje klient kräver mindre hårdvara och operativsystem p g a att inga tunga processer ligger här. Klienten behöver m a o inte vara fullt utrustade utan hålls så enkla som möjligt och utgör därigenom en betydligt mindre kostnad [www.whatis.com/thinserver.html, 1999-04-08] [hcg11.eng.ohio-state.edu/~tsengh/gs637lt.html, 1999-04-09]. Administration av applikationer sker centralt, vilket gör att man inte behöver springa runt till varenda dator och uppdatera eventuella förändringar i applikationslogiken. På administrationen sparas mycket tid och därmed också pengar som annars hade lagts på att hålla datan konsistent på flera platser [www.yougeek.com/rants/rfatrant.html, 1999-04-09].

4.1.10 Helheten är viktigast

Samtliga faktorer måste räknas in när man skall bestämma form av klient. Slutsatsen av de flesta resonemang utmynnar i att hänsyn måste tas till vilken applikation som skall utvecklas och vad företaget har för behov. M a o finns inget rakt svar på vad som är bättre eller sämre utan det är situationen som avgör. Ett övervägande måste företas, om vilka variabler som är mest nödvändiga att prioritera. Däremot ges inte mångtaliga exempel på vilka applikationer som kan tänkas lämpa sig för respektive teknik, en anledning till detta kan vara att de praktiska erfarenheterna från området är få.

4.2 Intervjuer

Intervjuerna företogs i huvudsak för att uppmärksamma vad som var viktigt att framhålla vid en jämförelse av de båda arkitekturerna, nämligen de intressen som styr valet. Bakom valet ligger kundens önskemål om ett visst system, systemet är i sin tur till för några specifika användare, vilka jobbar på ett visst sätt och de personer som skall underhålla systemet vill också ha ett finger med i spelet. Samtliga av de krav som kommer från olika håll måste utvecklaren ta hänsyn till vid konstruktionen, samtidigt som även utvecklaren har önskemål som skall uppfyllas. Mycket litteratur finns på området men vi tyckte att det var viktigt att få bekräftat och eventuellt tillagt aspekter som man inte tänkt på, från personer med erfarenhet från respektive område. Intervjuerna syftar ingalunda till att ge någon kvantitativt mått utan är inrättade för att bidra med kvalitativ information rörande de båda arkitekturerna. Vid samtalen kom det också fram en del erfarenheter av tunna respektive feta klienter, vilka är högst relevanta för vår utredning då den just vill ta fram eventuella skillnader i de båda arkitekturerna. Detta gör att vi inkluderat dessa synpunkter i redogörelsen för intervjuerna.

4.2.1 Intervjupersonernas bakgrund

Intervjuerna har företagits med tre olika individer. En person som har arbetat mycket med användare och lyft fram deras intresse i projekt. Genom denna intervju har vi kunnat komma fram till vilka generella krav en användare kan tänkas ha. För att få fram faktorer på underhållssidan valde vi att intervjua en tekniskt inriktad person som kunde bidra med ledtrådar om administrativa intressen men även utvecklarintressen. Av denna intervju kunde man också utvinna krav som finns från en beställningstagares sida. Ytterligare en intervju som gav nya infallsvinklar om intressenter företogs med en person som jobbar med mobil användning av informationsteknologi. Denna erfarenhet är också viktig, med tanke på att client/server-arkitekturen inte sällan används i applikationer avsedda för mobilt bruk.

4.2.2 Intressenter

Nedan följer en kort redogörelse för respektive intressent samt slutligen en mer översiktlig tabell över de olika krav som kan ställas på ett system och som på så sätt indirekt påverkar valet av arkitektur.

4.2.2.1 Slutanvändarens intressen

Naturligtvis är användarens krav beroende av vad det är applikationen skall användas till samt vilka erfarenheter den person, som skall använda applikationen har. Detta gör det extremt svårt att generalisera vad en användares krav är, vad man dock kan göra är att ta fram ett antal krav som finns för att sedan använda dessa när man skall välja arkitektur, men då bara titta på de som är relevanta, de andra kan man välja att bortse från.

4.2.2.2 Administratörens intressen

Administratörens största bekymmer är uppdateringar och nätverkstrafik. Som systemadministratör är man intresserad av den fysiska uppdelningen av applikationen för att få underhållningen av alla system så effektiv som möjligt. Till detta hör också intresset av att logga transaktioner för att kunna spåra händelser bakåt i tiden. Ytterligare en viktig faktor är nätverkstrafiken, för att administrationen skall bli så enkel som möjligt vill man inte ha överbelastade nätverk utan vill få en jämn fördelning på trafiken.

4.2.2.3 Utvecklarens intressen

Till utvecklingsperspektivet hör att det är viktigt med möjlighet till snabb prototypkonstruktion, för att kunna ge förslag till kunden. Ytterligare en aspekt är naturligtvis utvecklingstiden, det skall även gå snabbt att ta fram en färdig applikation. Som utvecklare är man intresserad av uppdelningen i logiska skikt samt att göra skikten så fristående som möjligt för att underlätta förändringar i och återanvändning av programkod. Vilken kunskapsbas tekniken kräver är också att uppmärksamma, samt för att kunna uppfylla kundens krav vilka funktioner som går att realisera.

4.2.2.4 Beställarens intresse

Beställarens önskemål är i stort sammanvävda med slutanvändarens, detta till följd av att systemet är till för användare, fungerar inte detsamma för slutanvändaren riskerar beställaren att förlora stora summor. Den största skiljepunkten ligger i att beställaren också har intresse av en så snabb utveckling och låg kostnad som möjligt. Andra viktiga aspekter är säkerhet i systemet i fråga om intrång men även reabilitet och konsistens i datan som systemet håller. Kraven överlappar dessutom administratörens i det att man önskar den lösning som ur underhållningssynpunkt är mest kostnadseffektiv.

4.2.2.5 Sammanställning

Slutanvändare	**Systemadministration**	**Utvecklare**	**Beställare**
• Starttid • Svarstid • Möjlighet att ångra på olika nivåer. • Integritet • Låsning/ icke låsning • Naturligt/ intuitivt • Typ av interaktion/ presentation • Olika nivåer av tekniskt kunnande • Personlig anpassning av gränssnitt/ Konfigurationsmöjligheter • Tillförlitlighet/Stabilitet • Mobilitet • Tillgänglighet	• Loggning av transaktioner • Omfattning på uppdatering • Nättrafik/ Belastning • Hårdvarukrav • Komplexitet; olika tekniker applikationen bygger på. Distribuerade servrar, databaser m.m. • Antal 3:e partsprodukter • Plattformsoberoende	• Utvecklingstid • Möjlighet till snabb prototyping • Möjligheter att realisera viss funktionalitet • Kunskaps bredd; teknik	• Utvecklingstid • Utvecklingskostnad • Säkerhet / Integritet • Hårdvarukrav • Modularitet; integrationsmöjlighet med andra system. • Plattformsoberoende

Tabell 1. Olika intressenters möjliga krav på en applikation.

Denna sammanställning skall uppmärksamma vad som skall tas hänsyn till vid valet av en viss arkitektur. Meningen är att uppställningen skall fungera som stöd när man kontrollerar vilka krav som är mest centrala vid utvecklingen av en viss applikation.

4.2.3 Synpunkter på feta klienter

Representanten för den tekniska sidan ansåg att feta klienter ur användarens synvinkel är positivt, detta p g a att man som användare besitter större inflytande över sin miljö. Ur underhålls synpunkt är det inte lika effektivt, då administratören, t ex vid uppdateringar måste se till att vardera maskin uppdateras, detta är mycket tidsödande. Speciellt användbart skall det vara att ha en fet arkitektur om användaren har en mobil enhet, där man inte hela tiden har

möjlighet att vara uppkopplad till en central server. Feta klientapplikationer som laddas ner på hårddisken får inte vara för stora, då de vid en sådan situation blir labila.

4.2.4 Synpunkter på tunna klienter

Från tekniksidan framkom även problem i samband med arkitektur med tunna klienter; vid synkronisering behöver man lösa frågan om hur man gör för att få data med tillförlitlighet som är konsistent. Tunna klienter anses ge upphov till mer nätverkstrafik, vad som krävs i dataöverföringen är överföring av block med data så att kommunikationen blir mer effektiv. Det finns även tekniska problem förknippade med mobilt användande och tunna klienter överföringshastigheten fyller idag inte alltid de krav som man önskar, varför det kan vara problematiskt att få tillgång till tidsenlig information.

4.3 Val av exempelapplikation

Att välja applikation är ett steg på vägen för att nå fram till det slutliga resultatet, det är viktigt att se till så att valet inte ger en snedvridning av resultatet. När man skall utveckla en client/server-applikation kan tänkas att det finns vissa applikationer som lämpar sig bättre för en viss typ av arkitektur, exempelvis att interaktionsintensiva applikationer skulle lämpa sig bättre för en fet klientarkitektur då kommunikationen sker i färre skikt och att en applikation där kommunikationen mellan klient och server är mer sparsam lämpar sig väl för en tunn klientarkitektur. Vi ville använda en exempelapplikation som kunde tänkas lämpa sig för implementation i båda de aktuella arkitekturtyperna. Att hitta en applikation som ligger helt neutralt visade sig svårare än vad vi tänkt. Valet föll istället på en tidrapporteringsapplikation. Anledningen var att vi genom detta val hade möjligheten att ändå göra en rättvis bedömning, då man genom applikationen kan representera situationer där båda arkitekturer kan tänkas lämpa sig. Applikationen innehåller två olika klienter, en för inrapportering av arbetstimmar och en för att få fram statistik. Statistikdelen av applikationen representerar en transaktionsintensivare applikation, medan inrapporteringsdelen visar på en applikation med mindre kommunikation mellan klient och server. På detta sätt har vi erhållit en applikationsdel där transaktionsfrekvensen är låg, p g a att kommunikationen endast sker en väg; från gränssnitt till databas och en applikationsdel som är mer intensiv till följd av de beräkningar som skall till för att få fram önskad statistik.

4.4 Val av utvecklingsverktyg

Liksom val av applikation kan ses som ett delresultat för att nå fram till det slutliga resultatet kan man även betrakta val av verktyg på detta sätt. När man väljer verktyg är det viktigt att fokusera på vad applikationen skall användas till, men det finns också andra faktorer som kan ha en inverkan på valet.

Programmeringspråk

För att konstruera prototyperna har vi valt att använda oss av programspråket Java då det lämpar sig väl för client/server-applikationer och innehåller den funktionalitet som krävs för att skriva nätverksapplikationer [Harold, 1997]. Genom Java ges också möjligheten att utveckla plattformsoberoende applikationer [Skansholm, 1998]. Java är objektorienterat och har inbyggt stöd för multitrådning, dvs att man parallellt kan exekvera flera olika processer [Eriksson, 1997]. Möjligheten till multitrådning är en viktig egenskap vid utveckling av client/server-applikationer då man vill kunna köra separata trådar/processer för varje klient som arbetar mot serverdelen i applikationen.

Kommunikationsteknologi

Som mellanvara, vilken måste till i den tunna klienten, för att de logiska lagren skall kunna kommunicera har vi valt CORBA. Bakom detta val ligger såväl politiska som rent ändamålsenliga orsaker. På Astrakan var man intresserad av tekniken, då den ännu inte till fullo prövats i något projekt. Ytterligare en orsak till att välja CORBA, utgjordes av att denna teknologi är mer lättillgänglig än t ex RPC eller andra mellanvaror på lägre nivå, med CORBA behöver man inte bry sig om lågnivådetaljer, vilket underlättar för utvecklaren.

Utvecklingsmiljö

Vidare har vi valt att använda IBMs integrerade utvecklingsmiljö VisualAge för Java då den erbjuder ett bra verktyg för grafisk design av användargränssnitt med SWING-komponenter och har stöd för flera samtidiga utvecklare med gemensam kod. Den version av vi använt oss av är VisualAge 2.0 Enterprise Edition, vilken stödjer jdk 1.1.6 och SWING 1.0.2. VisualAge har fått en mängd utmärkelser[6], bl.a. ”Best Java IDE” 1998.

Modelleringsverktyg

Till modelleringsfasen i prototypframtagningen så använde vi oss av Rational Rose 4 för att beskriva klasser, deras attribut och metoder. Skälen till att vi bestämde oss för att använda Rational Rose var att den kan generera klasser i Java direkt från modellen samt att den stödjer UML-notationen. Samtidigt används detta verktyg i Astrakans verksamhet, varför det även av detta skäl ligger nära tillhands att använda omnämnt verktyg.

4.5 Val av arkitekturuppdelning

Slutligen har vi ett återstående delresultat som vi var tvungna att uppnå, nämligen beslutet om uppdelningen av de logiska lagren, vilket till stor del är knutet till om man skall konstruera en tjock eller tunn klient. Tanken har varit att bygga upp de båda arkitekturerna på så lika sätt som möjligt, trots denna ansats landade vi vid beslutet att dela upp de båda arkitekturerna i olika antal skikt. Det föll sig naturligt att dela upp den tjocka klienten i två skikt, då man i princip har två logiska lager, nämligen det där gränssnitts- och applikationslogik är placerat, samt det lager där datalogiken är placerad. Vad gäller den tunna klienten kunde man tänka sig att lägga uppdelningen i två lager med endast gränssnittslogiken på klienten och övrig logik på servern. Använder man sig däremot av en treskiktsarkitektur, blir uppdelningen mer tydlig i form av att man har gränssnittslogik på ett ställe (klient), applikationslogik på ett (applikationsserver) och datalogik på ett tredje (databasserver). Mot bakgrund av ovanstående faktorer beslutade vi att bygga den tunna arkitekturen i tre skikt. När man i realiteten väljer uppdelning av arkitektur finns det ett flertal faktorer att ta hänsyn till, men som tidigare påpekat i avgränsningen, ämnar vi inte behandla de i vår undersökning.

4.6 Prototyperna

Stycket har direkt anknytning till problemformuleringen, då det bidrar med större delen av empirin i vår studie. För en mer konkret bild av konstruktionen i exempelapplikationerna se Appendix B.

4.6.1 Krav på prototyperna

Eftersom huvudtanken med denna utredning är att ta reda på skillnader i utvecklingen av tunna och tjocka klienter, har inte alltför stor möda lagts ner på användarvänlighet eller design av användargränssnitt. En annan anledning till att inte spendera tid på en annars så viktig

[6] http://www.software.ibm.com/ad/vajava/awards.htm

aspekt är den tidsram inom vilken vi varit tvungna att hålla oss. Vid implementationen har fokus legat på att få fram prototyper som skall fylla funktionen av exempelapplikationer. Applikationer vars syfte är att representera olika typer av gränssnitt (se ovan under val av applikation) för att få en så rättvis bedömning som möjligt.

4.6.2 Användargränssnittet

Den feta klientapplikationerna är på ytan lik den tunna, det är dock det som är placerat under ytan vi är intresserade av. Inmatningsgränssnittet består av tre komboboxar, i vilka användaren anger aktuell användare, det projekt, för vilket tid önskas inrapporteras samt tillhörande aktiviteter. För att presentera tider för aktuell användare finns en tabell i vilken användare också rapporterar in ytterligare tid genom att en knapptryckning på ”ny inmatning”. När användaren sedan trycker på "ok" knappen sparas informationen ner i databasen och fönstret stängs, väljs avbryt knappen stängs fönstret också, men tillagd information sparas inte. Statistikgränssnittet är uppbyggt med hjälp av två tabeller, vilka vardera visar två typer av information. Användaren har här möjlighet att välja att titta projektvis och aktivitetsvis, dvs väljer man en viss typ av projekt kommer alla aktiviteter för detta projekt att visas, väljer man en aktivitet kommer istället alla personer som är inblandade i denna aktivitet att visas. Förutom att välja projekt eller aktivitet skall användaren ange om denna vill se timmar eller kostnader för respektive objekt. Beroende på vad som väljs kommer man få se totala kostnader alternativt totala timmar summerade i ett textfält under respektive tabell.

4.6.3 Designen med tjock klient

Här är klienter uppbyggda av ovanstående gränssnitt, som innehåller både gränssnittslogik och applikationslogik. Servern tillhandahåller endast tillgång till databasen. Antal skikt har som ovan nämnt, här begränsats till två.

4.6.4 Designen med tunn klient

Klienterna i denna arkitektur är uppbyggda i tre skikt; användargränssnitt, applikationsserver och databasserver. Användargränssnittet innehåller endast gränssnittslogiken, applikationsservern bevarar verksamhetslogiken och databasservern har samma funktion som i den tjocka arkitekturen. Skillnaden i denna arkitektur är att datan inte passerar direkt till användargränssnittet utan först måste passera och bearbetas i applikationsservern, för att sedan skickas till gränssnittet, färdig för presentation samma gäller transaktioner åt motsatta hållet.

4.7 Erfarenheter från utvecklingsarbetet

Av utvecklingen erhålls mycket viktig information om just skillnader i de olika klient arkitekturerna. Genom erfarenheten kan man besvara problemformuleringen om vilken skillnaden är att utveckla i den ena eller andra tekniken men den leder också fram till vad man bör tänka på när det gäller viss funktionalitet i en applikation, som konstrueras i en specifik client/server-miljö.

4.7.1 Utveckling av den tjocka klienten

Vi valde att börja med den tjocka klienten. Tanken bakom detta förfaringssätt var att delar av koden till denna klient sedan skulle kunna återanvändas av den tunna klienten. Vad som krävdes för att få upprättat denna klient var att sätta sig in i den logik som redan existerade (se nedan under programlogik). Då vi dessutom använde delar av ett existerande ramverk, blev det ännu mer att sätta sig in i än bara själva programlogiken. En hel del kod var m a o redan

klar, för att kunna använda denna var man dock tvungen att förstå vad den generar för händelser, vilket var ganska problematiskt. Vad vi vann på detta förfarande var t ex att vi, tack vare all färdig kod inte behövde gå in och skriva några databasanrop. Vad som i princip behövdes göras, var att skapa användargränssnittet samt modeller för komponenterna i gränssnittet samt koppla händelser till komponenterna. Detta låter kanske inte så avancerat men det tog tid att förstå sig på kopplingarna mellan komponenter, modeller och händelser som ovan nämnt. Tabellerna som vi använde för att visa information men även för att lägga till information i, var de komponenter som ställde till mest problem för oss. Tabeller är väldigt komplexa, vilket bidrar till att det finns oändliga möjligheter om man använder dem. När vi väl kommit över alla hinder och avslutat inmatningsgränssnittet kunde vi ta med oss de erfarenheter vi skaffat från detta arbete och slutgöra statistikgränssnittet snabbare trots att detta är mer komplext än det förstnämnda.

4.7.1.1 Modelleringen

Modelleringen är ett naturligt steg i systemutvecklingen. Eftersom vårt största intresse var att nå fram till en jämförelse beslutade vi oss för att hålla modellen så förenklad som möjligt. Totalt består modellen av fyra klasser; en för användaren, tidsrapporten, projektet och aktiviteten i projektet. Modelleringen fungerade i detta projekt dessutom som grund för den kodgenerering som företogs. Som tidigare nämnt skedde själva objektmodelleringen i Rational Rose. Under modelleringen togs också ställning till hur arkitekturerna skulle konstrueras. Vad gäller den tjocka klienten var vi överens om att klienten skulle bära både gränssnitt och affärslogik och att den skulle bestå av två skikt, resterande logik placerades m a o på servern. Appendix B innehåller en mer ingående beskrivning av modelleringsfasen.

4.7.1.2 Kodgenereringen

Som påpekat ovan användes vår objektmodell för generering av skelettet till våra klasser. Vad som utvanns av genereringen var ett databasskikt och ett verksamhetsskikt; metoder för att skapa objekt, samt accessmetoder för vardera klass och relation. Dessutom genererades ett databasskript för upprättande av tabeller. Genereringen gjorde en stor del av rutinarbetet i programmeringen, varför även detta inslag var tidsbesparande.

4.7.1.3 Programmeringsmetoden

Astrakan har en utarbetad metod vid programmering. Man använder objekt hela vägen från databas ut i användargränssnitt, dvs man hämtar t ex inte upp datafragment från databasen utan hela objekt. Vi bestämde oss för att tillämpa deras metod dels för att det känns naturligt att arbeta med ren objektorientering när man både har använt sig av en objektorienterad modellering och programspråk, samt av den anledning att koden blir mycket väl strukturerad. Tillämpningen har dock inte varit helt oproblematisk. Det har varit tidskrävande att sätta sig in i en logik som redan är utarbetad. Vi har stundtals känt det svårt att överblicka vad det är som hänt, just p g a det faktum att vi inte har haft full förståelse för kodens uppbyggnad. För övrigt hade det underlättat för oss om kommenteringen av koden varit mer utförlig.

4.7.1.4 Programlogiken

Vi anser det relevant att beskriva mer ingående hur logiken är uppbyggd, då den är en vital del av applikationens uppbyggnad och funktion. Detta är också viktigt att beskriva med tanke på den jämförelse vi vill åstadkomma vad gäller skillnader i utvecklingen av de båda arkitekturerna. Verksamhetsskiktet består av tre delar; affärsobjekt, kontrollobjekt samt databasobjekt. Varje klass har två representationer i vardera affärs- och databasobjekt. I affärsobjektet finns för varje klass en implementation av accessmetoderna, som tidigare nämnt, dessutom finns det ytterligare en implementation för de unika regler som tillhör en

specifik klass. Affärslogiken rymmer också två generella klasser som kallas affärskontroller respektive databaskontroller vilka ombesörjer kommunikationen med databasobjektet. I databasobjektet har vardera klass i sin tur en representation i persistens objekt, dvs en klass med alla för klassen tillhörande attribut och relationer, samt en databasrepresentation som hämtar upp information om det aktuella objektet från databasen. Precis som affärsobjektet har databasobjektet två generella klasser som håller kopplingen till affärslogiken, nämligen ett persistensobjekt och ett databaskontrollobjekt. Till sammanhanget hör att samtliga av ovanstående generella objekt har "Object" som sin superklass medan övriga specifika klasser ärver dessa generella klasser. Kontroll objekten innehåller egendefinierade hjälpklasser för t ex uppbyggnad av SQL-satser. Utöver verksamhetsskiktet finns en mängd omdefinierade klasser, ett exempel är klassen "Vector" som har anpassats för att kunna hantera de specifika objekt som existerar i verksamhetsskiktet. Denna uppbyggnad, med anpassning för hantering av de egna objekten genomsyrar hela logiken. Vid utveckling av den tjocka klienten behövdes ingen vidareutveckling av logiken utan den kunde användas i befintligt skick.

4.7.1.5 Krav på utvecklingsmiljön

För att kunna realisera den tjocka klienten behövdes som nämnts endast databasen och utvecklingsverktyget VisualAge för Java (se nedan), men det skulle gått lika bra att använda ren jdk även om det kan vara mer tidsödande. Att sätta upp miljön bidrog inte till några problem det var inte heller problematiskt att sätta upp sökvägar för att kunna köra applikationerna.

4.7.1.6 Utveckling i en integrerad utvecklingsmiljö

Vad som var helt nytt för oss var VisualAge. Verktyget tog bra mycket längre tid att sätta sig in i än vad vi räknat med.

Miljön

När man arbetar i verktyget kan man välja grafisk komposition eller att arbeta direkt i koden. Koden finns samlad i vad som kallas en arbetsbänk, här finns samtliga projekt och paket samlade på ett överskådligt sätt i en trädstruktur. Under varje paket finns tillhörande klasser och under varje klass tillhörande metoder. Designen gör det väldigt lätt att få en överskådlig blick över ingående klasser. För att ytterligare underlätta överblicken finns en hierarkisk översikt för att se vilka klasser som är relevanta för en annan klass. Vad gäller felhantering finns också en översikt som rapporterar alla aktuella fel.

Grafisk komposition

Värt att uppmärksamma med aktuellt verktyg är metoden för design av användargränssnittet. Komponenter kan likt många andra programmeringsverktyg snabbt och smidigt placeras ut till önskad design. För att koppla relationer och knyta händelser till komponenter, dras linjer mellan komponent och händelse. I varje händelse måste man i tillägg ange vilken funktion det är som skall anropas och vilken slags händelse det är som skall utlösa denna funktion. Konceptet är inte trivialt att sätta sig in i, det kräver en hel del ansträngningar innan man förstår vad som händer. En hjälp var att titta på den kod som genererades. Metoden kräver ett annat tankesätt än vad vi varit vana vid, men när man väl kommit underfund med tekniken visar den sig vara väldigt smidig att arbeta med.

Kodning

Utöver möjligheten till grafisk design av användargränssnitt kan man naturligtvis koda direkt i verktyget. Hjälpmedel finns också för att skapa nya metoder och klasser i koden. Kompileringen sker efterhand som man lägger till kod. Man är faktiskt tvungen att kompilera koden så snart man lämnar en metod för att arbeta med en annan. Detta är tidsödande när man

snabbt vill hoppa mellan olika delar av koden. Ytterligare en faktor att uppmärksamma, är att vissa delar av koden som genereras av verktyget skrivs över varje gång en omgenerering sker. Detta gör att det är väldigt noga att när man skall lägga egendefinierad kod i ett sådant avsnitt lägger denna inom ett område som är skyddat, för varje automatgenererad metod finns åtminstone ett sådant område.

Debugger

Det går smidigt att sätta brytpunkter i koden, debuggern öppnar sig själv när den kommer fram till aktuell brytpunkt. Tillgången till information om variabler är bra och det är lätt att följa förändringen i en variabel. Genom att markera en variabel kan man få tillgång till den information som variabeln innehåller eller titta på en trädstruktur som visar samtliga av de för tillfället berörda variablerna.

Problem

Speciellt den grafiska designen var mycket tidsödande, vid en förändring uppdateras gränssnittet väldigt sakta. Dokumentationen till VisualAge kan inte berömmas i större utsträckning. Trots den extensiva dokumentation som fanns att tillgå var det svårt att hitta svaren på de specifika problem vi ville ha svar på. Själva VisualAge är lite instabilt, när man stänger ner verktyget sker en slutlig nedsparing av de projekt man arbetat med. Nedsparingen har flera gånger misslyckats, med förlorad data som följd. Detta har kostat ett antal förlorade arbetstimmar.

4.7.1.7 Databas och databasmellanvara

Databasen som använts är från Oracle, då det är en databasleverantör som kan tillhandahålla drivrutiner för kommunikation med Java, samt att det fanns riklig dokumentation för produkten. Oracle har en stor marknadsandel, det finns ett stort antal installationer och produkten är anpassad till ett flertal olika miljöer. Vi har använt oss av Oracle 7.3.3 och installationen har fungerat klanderfritt under hela utvecklingstiden. För att gå in i databasen och kontrollera att uppgifter lagts in rätt samt för att administrera databasen använde vi oss av ett verktyget Toad. Hjälpverktyget var mycket intuitivt, så det var lätt att utföra de uppgifter som krävdes.

Den tjocka klienten pratar JDBC direkt mot databasen. Som vi påpekat ovan besparade återanvändningen och genereringen av kod oss att behöva implementera denna del. Vad som var nödvändigt att ändra i koden, var att ställa om IP-adressen till den dator som databasinstallationen var placerad på.

4.7.1.8 Krav på kunskaper, tid och kostnader

De krav som ställdes på kunskap för att utveckla den tjocka klienten var ”enhetliga”, dvs det krävdes inte kännedom om många olika tekniska områden, utan det räckte i princip med kunskap om Java och SQL. Tidsåtgången för modelleringen var inte omfattande, då det var klart hur den skulle företas. Kodningen av den tjocka klienten tog drygt 4 veckor, trots att applikationen inte var omfattande. Kostnader för utveckling av klienten, skulle bestå i de båda utvecklingsverktygen samt ovannämnda tid för utveckling.

4.7.2 Utveckling av den tunna klienten

Vad som behövs i tillägg till den tjocka varianten är ytterligare ett gränssnitt, nämligen CORBA skiktet, vilket placeras mellan användargränssnitt och verksamhetsskikt. Applikationslogiken skall förflyttas ett steg och hos klienten skall endast det grafiska gränssnittet tillåtas att ta plats. Avsnitten om kodgenerering, programmeringsmetoden och databasmellanvara skiljer sig inte mellan utvecklingen av de två prototyperna, varför vi valt

att utelämna dessa avsnitt. Något som tillkommer är avsnittet om den applikationsmellanvara som vi valt att använda oss av.

4.7.2.1 Modellering

Vid modellering av den tunna klienten fanns det mer att ta hänsyn till än vid den tjocka; skulle man göra en supertunn klient med hjälp av dynamisk html eller nöja sig med en tunn klient. Hur många skikt var lämpligt att använda sig av. För att få en helt rättvis bedömning borde också denna klient vara uppdelad i två skikt, med enda skillnaden i att man placerar affärslogiken på servern istället. Vi ansåg dock att om man väljer detta tillvägagångssätt har man frångått en viktig tanke med tunna klienter, att skapa ett mellanskikt som ökar möjligheten till skalbarhet. Vad gäller frågan om dynamisk html eller Java, ville vi inte frångå strävan efter likhet mellan klienterna, dvs båda skulle skrivas i samma språk för att avlägsna eventuella bias i fråga om skillnad i programspråkens effektivitet eller uppbyggnad. Resultatet av diskussionerna blev konstruktion av en tunn klient skriven i Java med tre skikt, dvs användargränssnitt som endast begär information och visar upp den i gränssnittet, en applikationsserver som håller verksamhetsskiktet samt databasservern.

4.7.2.2 Programlogiken

Programlogiken som vi hade att utgå ifrån var den samma som nämnt ovan under utveckling av den tjocka klienten, men i fallet med tunn klient är man också tvungen att ta hänsyn till hur logiken skall delas upp på lämpligt sätt mellan de olika skikten, det är inte så självklart att dra en linje mellan applikations- och presentationslogik. Frågor uppkommer huruvida olika delar av programmet tillhör den ena eller andra logiken, gränsen mellan de båda kan vara flytande. Inga riktlinjer finns på hur man skall företa en sådan uppdelning utan denna måste helt anpassas efter hur logiken hänger ihop och hur nära olika delar av logiken behöver vara, varför uppdelningen blir godtycklig. Man skulle behöva mer tid för att pröva olika uppdelningar och därigenom upptäcka vad som torde vara lämpligast.

4.7.2.3 Utvecklingsverktyget

I VisualAge finns det möjlighet att installera ett utvecklingsmiljö för IDL-gränssnitt. Man kan sedan konfigurera VisualAge för användning av IBMs egen ORB alternativt en ORB från någon annan leverantör. Vi valde att använda oss av möjligheten att utnyttja en ORB från en annan leverantör. Vi valde att använda VisualAge även för utvecklingen av de klasser som tillkom för att kunna implementera den tunna klienten.

4.7.2.4 Applikationsmellanvara

Val och installation av ORB

Det var svårt att komma igång med CORBA, det fanns ingen CORBA-kunnig person att rådfråga på Astrakan eller i den egna bekantskapskretsen. Det tog två veckors iterativt arbete för att få vårt första CORBA-exempel att fungera tillfredsställande. De främsta orsakerna till att utvecklingen gick trögt var att det var svårt att få utvecklingsverktygen för CORBA att fungera tillfredsställande med den integrerade utvecklingsmiljön för Java, i vårt fall VisualAge. Vi försökte först att använda IBMs egen ORB, Component Broker, vilket innebar att vi var tvungna att lägga på ett antal uppgraderingar på VisualAge för att utnyttja deras utvecklingsmiljö för CORBA. IBMs ORB lämnades därhän efter ett antal tappra försök när det stod klart att det saknades vissa nödvändiga klasser i uppgraderingen. Efter att ha undersökt ett antal leverantörer av ORBar så föll valet på Visibroker från Inprise, då den var väl dokumenterad och vi hade tillgång till litteratur som behandlade den ingående.

Utveckling med CORBA

Vid kompileringen av klasserna var det en del problem som avhjälptes genom att ställa in miljövariabler som PATH och CLASSPATH korrekt. Dessa miljövariabler är sökvägar som används för att hitta kompilatorn och de externa klasser som man refererar till i de egna klasserna. Ett annat problem uppstod vid användningen av Visibrokers IDL-till-Java-kompilator, kompilatorn försökte skapa en logfil i ett bibliotek som inte existerade, vilket resulterade i att kompileringen avbröts utan att genera något felmeddelande som beskrev problemet. Detta gav upphov till en hel del frustration innan felet kunde ringas in och avhjälpas. För att kunna använda oss av CORBAs funktionalitet blev vi tvungna att lägga till kod i både klient- och serverapplikation. Först måste man beskriva alla berörda objekt med IDL-gränssnitt och skapa implementationer av objekten i målspråket, i vårt fall Java. När Java filerna kompileras skapas klassfiler och vid kompileringen av IDL-filerna skapas en mängd filer. De viktigaste filerna som genereras är stubbarna och skeletten, de används av klient-respektive serverapplikation för att realisera metodanrop från klientapplikationen.

Kommunikation mellan delar

När all kompilering var avklarad så återstod det att få delarna i applikationen att exekveras och kommunicera med varandra. För att få kommunikationen att fungera var man tvungen att först starta en ”Naming service”, som är ett slags namnregister där servern först namnger och registrerar sina objekt. När serverapplikationen startats går det bra att starta godtyckligt antal klientapplikationer. Klientapplikationen ställer sedan en förfrågan till namnregistret där den skickar med namnet på objektet man vill kommunicerar med och får sedan en objektreferens till det aktuella objektet. När klientapplikationen har objektreferensen så kan den direkt anropa objektets metoder utan att gå via namnregistret.

4.7.2.5 Krav på utvecklingsmiljön

Vad gäller utvecklingsmiljön var det markant mer problematiskt att sätta upp den. Förutom vad som krävdes för den tjocka klienten behövdes applikationsmellanvaran implementeras. Upprättandet av denna miljö var inte så smidig som de två ovanstående, databasmellanvaran och utvecklingsverktyget. Att ställa in sökvägar var också betydligt krångligare, bl.a. p g a att vi fick göra det direkt i dosmiljö. Inblandning av ytterligare miljö, applikationsmellanvaran, försvårar utvecklingsprocessen i stor omfattning.

4.7.2.6 Krav på kunskaper, tid och kostnader

Utöver vad som nämnts under samma rubrik om tjocka klienter, tillkommer ytterligare krav vid utveckling av den tunna klienten. De tekniska kraven ökar i och med berörd mellanvara. Trots att vi valde att implementera en mellanvara som gömmer krångliga tekniska detaljer för programmeraren, tog det mycket extra ansträngning, för att uppnå tillräcklig kunskap för att kunna applicera denna teknik. Utvecklingen av den tunna klienten tog uppskattningsvis 3 veckor extra jämför med den tjocka, vilket till stor del var ett resultat av den tid det tog att läsa in sig på området och den tid det tog för själva implementeringen. Även modelleringsfasen bidrog till längre utvecklingstid genom de ytterligare aspekter som fanns att ta hänsyn till, så som anpassad uppdelning av logik och bestämning av hur skiktad arkitekturen bör vara. Som ett resultat av de båda tidigare ökade kraven, växer också kostnaderna. Kostnaderna blir dessutom större p g a den extra utrustning som måste till i form av t ex licenser för programvaror som används vid implementering av mellanskiktet.

5 Diskussion

Huvudtanken med uppsatsen har varit att undersöka de effekter som valet av klient får på utvecklingen. Vi vill även bidra med vidare idéer om vad man bör tänka på när man står inför att utveckla en client/server-applikation, t ex hur olika faktorer kan påverka valet av klient. Under rubriken generell diskussion vill vi därför förmedla våra tankar om aspekter som är väsentliga att uppmärksamma när man står inför valet av arkitektur. En kritisk granskning av resultat är också på sin plats för att se hur man skulle kunna förstärka detta och i anknytning till denna granskning, idéer om vidare frågor att behandla under aktuellt problemområde.

5.1 Generell diskussion

Som utvecklare kan man inte bara titta på den egna situationen, det finns många andra faktorer att ta hänsyn till. Utan användare finns ingen att utveckla applikationen för, som resultatet visar finns många intressen att ta vara på, många gånger intressen som är svåra att förena. Vad som är essentiellt för utvecklaren, är att ta fasta på och uppmärksamma de viktigaste av dessa intressen samt att ta vara på så många som möjligt. Detta styr sedan hur applikationen utformas, vilket i sin tur kommer att påverka vilken arkitektur som är lämplig.

Frågorna som kan ställas för att nå fram till ett slutgiltigt beslut om arkitektur är många. Man bör bl.a. ställa frågan vilken teknisk utrustning som finns att tillgå, t ex vilken överföringskapacitet nätverket har och hur kraftfulla klientdatorer som önskas användas. Har man tillgång till ett snabbt nätverk behöver man inte bekymra sig för transaktionsfrekvensen och en tunn klientimplementation torde inte utgöra något hinder prestanda mässigt. Är nätverkets förmåga däremot begränsad bör man titta på hur transaktionsintensiv applikationen är; går det mycket information fram och tillbaka eller går kommunikationen bara en väg och är applikationen beräkningsintensiv. I kombination med ett nätverk med låg kapacitet för överföring av data, torde det inte vara ändamålsenligt att implementera en tunn klient i en applikation som har hög transaktionsintensitet om användaren t ex kräver snabba svarstider. Skall man ha "lågbudget”-klientdatorer bör man vidare tänka på att man inte kan placera för mycket logik på klientdatorn, men bör samtidigt vara medveten om att denna lösning kräver en kraftfull server för att klara den belastning samt hålla den mängd logik som förflyttas från klienterna.

Ytterligare en företeelse att beakta är hur mycket kontroll användaren behöver över applikationen. Skall applikationen anpassas individuellt, då är det lämpligare att placera logiken nära användaren. Är applikationen däremot standardiserad bör logiken lämpligen placeras centralt bl.a. för att underlätta administrationen. En annan viktig aspekt att uppmärksamma är antalet användare och huruvida man planerar att utöka antalet användare. Är det sistnämnda en viktig aspekt bör man överväga att ha tunna klienter för att öka skalbarheten och därmed underlätta en eventuell utvidgning av antalet klienter. I anknytning till vad som togs upp ovan berörande placering av logiken kan man försöka göra en uppskattning av hur stort behovet av administration och uppdatering kommer att vara samt var den största delen uppdatering kommer att ske, dvs i applikations-, databas- eller presentationslogik. Kommer applikationen beröras av frekvent uppdatering torde det vara lämpligast att placera så stor del som möjligt på serversidan. Är det endast delar av logiken som behövs uppdateras bör man se till att placera dessa samt underliggande lager centralt.

Vidare bör hänsyn tas till om applikationen skall användas med fast uppkoppling till en server eller om den skall vara mobil med en uppkoppling som sker sporadiskt för att hämta in information när den behövs. I fallet mobil användning kan det vara bättre med en tjock klientapplikation, då klienten frigörs från totalt beroende av servern. Är uppkopplingen konstant kan tänkas att en tunn lösning går lika bra, då ett beroende i detta fall inte behöver utgöra något hinder.

Med effekten på utvecklingen i åtanke tillsammans med nyss nämnda frågor, kan utvecklaren göra ett medvetet val och styra in detta på vad som kan anses lämpligt, med tanke på de monetära resurser och tidsresurser som finns att tillgå.

5.2 Slutsats

Tunna klienter ger effekten av mer resurskrävande utveckling än tjocka klienter. Resurserna uppträder i form av tid, utrustning och kostnader. Ökade krav på tidsresurser kommer sig av ökad komplexitet vid modellering och design, så även vid kodningsfasen, testkörning och inhämtande av kunskap. Samtliga faktorer bidrar m a o till en längre utvecklingsprocess vid utveckling av en tunn klientapplikation jämfört med en tjock. Vidare krävs det resurser i form av ytterligare verktyg och licenser, vilket visserligen beror på vilken teknik man väljer för att implementera den tunna klienten, men det krävs ändå alltid extra utrustning jämfört med utveckling av en tjock klient. Tänker man dessutom testa miljön krävs det en mer komplex installation för att upprätta en genuin miljö. Tittar man däremot på de hårdvaruresurser som krävs, så kan dessa sjunka förutsatt att klientdatorn endast implementerar tunna klientapplikationer, men denna aspekt berör utvecklingen i mindre mån. Ovannämnda resurskrav bidrar också till att kostnaderna för utvecklingen ökar tillsammans med det faktum att utvecklaren måste ha en viss teknisk specialkompetens, som troligen är dyr att köpa in om man inte redan har tillgång till den. Utvecklingen av en tunn klient är dessutom mer osäker då det inte finns några gemensamma utprövade modeller för hur konstruktion skall ske dvs det är lättare att bygga in fel då man inte har klart för sig hur uppdelningen skall ske eller vilka effekter den kommer att få. Men inget ont som inte för något gott med sig, modelleringen och designen av en tunn klient innebär också stor frihet för utvecklaren att bestämma över uppdelningen av komponenter och därmed möjlighet att anpassa applikationen till existerande miljöer. I tabellen nedan har vi sammanfattat ett antal faktorer där tunna och tjocka klienter skiljer sig åt. Med tanke på den extra möda som krävs för modellering, programmering och de extra kostnader som kan tillkomma för kompetensförhöjning, extra utrustning och licenser samt den osäkerhet som tillkommer är det ett mycket viktigt övervägande huruvida man skall konstruera en tjock eller tunn klient. Viktigt är att utvecklaren tar sitt ansvar och erkänner situationer som kräver utveckling av en tunn klient, även om det krävs betydligt mer resurser och större insatser från utvecklaren. Man kan kort och gott säga att fördelarna måste överväga de kostnader som tunna klienter innebär i utvecklingsstadiet, alternativt att applikationen kräver en tunn klient för att kunna realiseras.

	Tunn Klient	**Tjock Klient**
Krav på Serverprestanda	Höga	Låga
Krav på Klientprestanda	Låga	Höga
Komplexitet	Hög	Låg
Flexibilitet	Hög	Låg
Utvecklingskostnad	Hög	Medel
Krav på utvecklarens kunskaper	Höga	Medel
Underhållskostnader	Låga	Höga

Tabell 2. Skillnader mellan tunna och tjocka klienter.

6 Utvärdering

6.1 Kritisk granskning

För att återknyta till frågan om reabilitet, validitet och generaliserbarhet går vi här igenom resultatet för att se vilka synpunkter som kan framkomma vid en kritisk granskning. Vad gäller reabilitet och litteratur har vi utgått ifrån att källorna varit tillförlitliga, då litteraturen är skriven av ansedda författare och utgiven av kända förlag. Däremot skall uppmärksammas att det förekommer mycket subjektivitet berörande åsikter om rekommenderat teknikval. En liten mängd material har hämtats från produktsidor vilka därmed kan betraktas som mycket subjektiva och inriktade, vilket på så sätt påverkat validiteten negativt. Vid intervjuerna gavs utrymme för subjektiva åsikter, vilket kan tyckas påverka reabiliteten; hade man intervjuat fler hade kanske motstridiga åsikter och ytterligare aspekter uppkommit. Men man måste då ta i beaktning att intervjupersonerna har god kunskap inom respektive område, vilket kan bidra med en grad av legitimitet. Vidare är det svårt att avgöra huruvida man fått full tillgång till de intervjuades kunskap i och med att man har företagit fria intervjuer, detta måste dock vägas mot möjligheten att få fram extra information genom att tillåta denna frihet. I tillägg finns inget intresse i form av prestige eller önskan att hålla kunskapen för sig själv för att dölja information, inte heller har det rört sig om känslig information, varför intervjuerna ur denna synvinkel inte borde ha utsatts för validitetsförlust. Hur troligt är det att de idéer som framkommit lämpar sig i en annan miljö? Visserligen kommer utvecklingen skilja sig en aning beroende på vilken applikation som utvecklas och vilken miljö den utvecklas i, men det kommer ändå oberoende av detta finnas behov av liknande hänsynstaganden och beslut att fatta vad gäller respektive arkitektur. Vardera klientstruktur har sina egenskaper som kräver specifik behandling vilken inte skiljer sig mellan olika miljöer, grundstegen för att bygga en tunn respektive tjock klient är desamma vilken miljö utvecklaren än befinner sig i, däremot kommer miljön troligen påverka valet av arkitektur, men det är en helt annan fråga. Hur är det med frågan om samma observationer kommer att göras av andra utredare vid andra tillfällen? I bedömningen av utvecklingstid för respektive klienttyp bör man beakta att tidsangivelserna för utvecklingen i denna studie skall ställas i relation till att utvecklarna inte hade någon tidigare erfarenhet av den aktuella applikationsmellanvaran. Kanske är det så att någon som redan besitter en gedigen kunskap berörande de extra teknologier som krävs för att utveckla en tunn klient inte skulle se detta som ett hinder eller tycka att det hade så stor effekt på utvecklingen. Mot detta faktum kan ställas att tekniken idag är ung och det inte är så många som är bevandrade på området, varför det ändå torde höra till vanligheterna att det finns en extra barriär att forcera innan man kommit genom utvecklingen av en tunn klient.

6.2 Förslag på vidare frågor

Ytterligare aspekter som kan undersökas på området tunna och tjocka klienter hör till det mer subjektiva slaget, men det finns också rent faktiska aspekter att undersöka. Framtagna prototyper skulle kunna användas för prestandatester av respektive applikation, vilken syftar till att ge en jämförelse som kan dementera eller bekräfta de många uppgifter som finns berörande prestanda. Tester av detta slag skulle också kunna bidra till att upprätta en kriterielista, där egenskaper hos respektive typ av klient spaltas upp. Vid beslut om konstruktion kan kriterielistan matchas mot de krav som finns på aktuell applikation, för att slutligen få fram lämplig typ av klient. För att få respons på mottagandet av en applikation som är tillverkad med vardera typ av klient, kan man genomföra användartester på prototyperna. Användartesterna skulle kunna genomföras exempelvis genom observation och intervjuer berörande hur användaren uppfattade applikationen.

7 Referenser

7.1 Böcker

Easterby-Smith, M. , Lowe, A. , & Thorpe, R. (1991). ***”Management Research An Introduction”***. London: Sage.

Eriksson H. E. (1997). ***”Programutveckling med Java”***. Lund: Studentlitteratur

Harold E. R. (1997). ***”Java Network Programming”***. Sebastopol: O’Reilly & Associates, Inc.

Loosley, C. , & Douglas, F. (1998). ***”High-Performance Client/Server”***. N.Y.: John Wiley & Sons, Inc.

Mathiassen, L. , Munk-Madsen, A. , Nielsen, P. A. , & Stage, J. (1998). ***”Objektorienterad analys och design”***. Lund: Studentlitteratur.

Renaud, P. (1996). ***”Introduction to Client/Server Systems”***. N.Y.: John Wiley & Sons, Inc.

Schettino, J., & O’Hara, L. (1998). ***”CORBA for dummies”***. Foster City, CA.: IDG Books Worldwide

Skansholm, J. (1998). ***”Java Direkt”***. Lund: Studentlitteratur.

Sommerville, I. (1997). ***"Software Engineering"***. Harlow: Addison-Wesley.

Wiedersheim-Paul, F. , & Eriksson, L. T. (1991). ***”Att utreda, forska och rapportera”***. Malmö: Liber-Hermods

7.2 Artiklar

Dahlbom, B. (1995) Göteborg Informatics***, Scandinavian Journal of Information Systems,vol 7, no 2***.

Dahlbom, B. (1997) The New Informatics***, Scandinavian Journal of Information Systems,vol 8, no 2***.

Danielsson, L. (1998). Vi måste kämpa mot de stora tillämpningarna. ***Computer Sweden, 53***.

Lewandowski, S. M. (1998). Frameworks for Component-Based Client/Server Computing. ***ACM Computing Surveys, Vol. 30, 1***.

Pettersson, L. (1996). Tunna klienter i Suns vision. ***Datateknik, 13***.

Sandred, J. (1998). Dumskallarna är här. ***Datateknik, 05***.

Why Network-Centric Computing? (1998, 28 maj). ***Neoware Systems, Inc***.

Tseng, H. Z. (1999, april 9). ***hcgl1.eng.ohio-state.edu/~tsengh/gs637lt1.html***.

www.whatis.com. (1999, 8 april).

www.whatis.com/thinserv.html. (1999, 9 april).

www.byte.com/art/9704/img/047csh2.htm. (1999, 9 april).

www.sei.cmu.edu/str/descriptions/orb_body.html. (1999, april 8).

www.sei.cmu.edu/str/descriptions/corba_body.html. (1999, april 8).

www.vt.edu:10021/M/mpriddy/term.html. (1999, april 9).

www.axis.com. (1999, april 9).

www.yougeek.com/rants/fatrant.htm. (1999, april 9).

www.sei.cmu.edu/str/descriptions/clientserver_body.html. (1999, april 8).

8 Appendix A

8.1 Definition av begrepp

För att klargöra vad vi menar med olika benämningar som tas upp i beskrivningen av problemområdet, följer nedan definitioner, relevanta för vår beskrivning.

Client/Server - Arkitektur för distribuerade system, dvs system där moduler kan köras på olika plattformar, som består av en serverdel som tillhandahåller tjänster till klientdelen när denna gör förfrågningar till serverdelen. Placeringen av användarinteraktion, valideringar, utskrifter, databaskommunikation fördelas olika mellan klient- och serverdelen beroende på val av arkitektur. Tidigare var client/server synonymt med tvåskiktsarkitektur (se nedan). Nu är client/server snarare ett samlingsbegrepp för allt från tvåskiktslösningar till flerskiktslösningar.

Tvåskiktsarkitektur - Innebär att nätverks arkitekturen är indelad i två delar, server och klient. Tvåskiktslösningar kan realiseras med både tunna och tjocka klienter (se nedan).

Treskiktsarkitektur - Består av server, mellanskikt och klient. Klienten handhar endast det grafiska användargränssnittet, mellanskiktet består av en applikationsserver som innehåller all affärslogik (se nedan) och det tredje skiktet utgörs av en databasserver.

Feta/Tjocka klienter - Vid användningen av tjocka klienter placeras merparten av affärslogiken och eventuellt databasinteraktionen på klientsidan.

Tunna klienter - Klientsidan innehåller vanligen, endast presentationslogik, placeringen av affärslogik och databasinteraktion sker på serversidan.

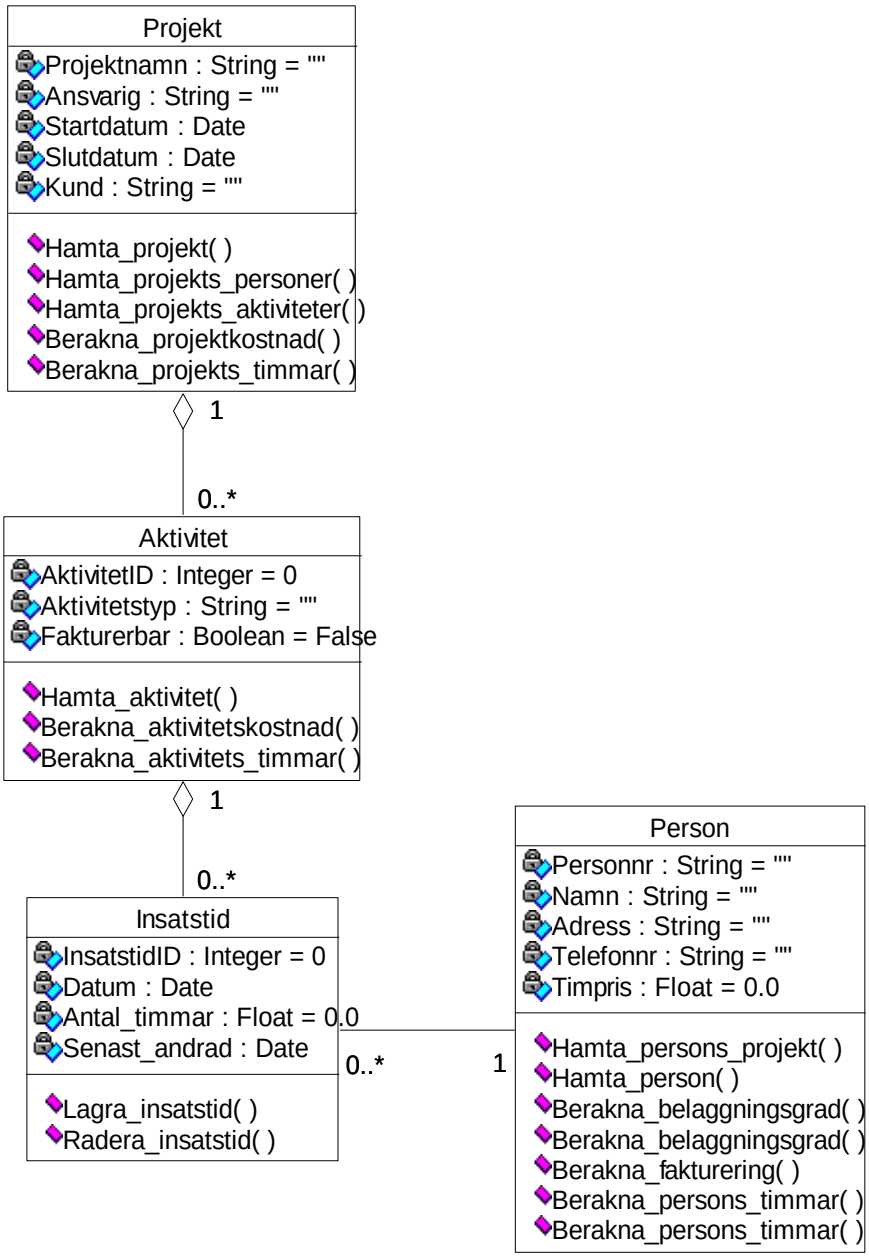
Affärslogik - Likställt med applikationslogik [Loosley & Douglas, 1998, s28], vilken förmedlar data och funktionsanrop mellan presentationslogik och dataskikt. Tar emot transaktioner från presentationslogiken, som bryts ner och skickas till lämpligt dataskikt. Utför olika beräkningar samt datamanipulation, t ex statistisk analys, på data som kommer från dataskiktet och skickar detta vidare till presentationsskiktet.

9 Appendix B

Nedan presenterar vi några utvalda delar av modelleringsfasen. Vi har valt att inkludera de delar av UML-metoden som är relevanta för vår studie. Objektmodellen visar grunden till de klasser som i ett senare skede genererades av kodgeneratorn. Funktionslistan ger en översikt över de funktioner vi valt att implementera i de båda gränssnitten. Komponentstrukturen ger en indikation på hur vi valt att dela upp komponenterna i respektive arkitektur. Under rubriken gränssnitt visar vi användargränssnittet så som det implementerats.

9.1 Objektmodellen

Objektmodellen gjordes i Rational Rose, verktyg för objektmodellering som stödjer UML metoden. Från denna modell kan man i Rose också generera kodskelett. Vi har dock valt att inte utnyttja denna möjlighet med Rose utan istället använt oss av Astrakans egentillverkade kodgenerator.



9.2 Funktionslista

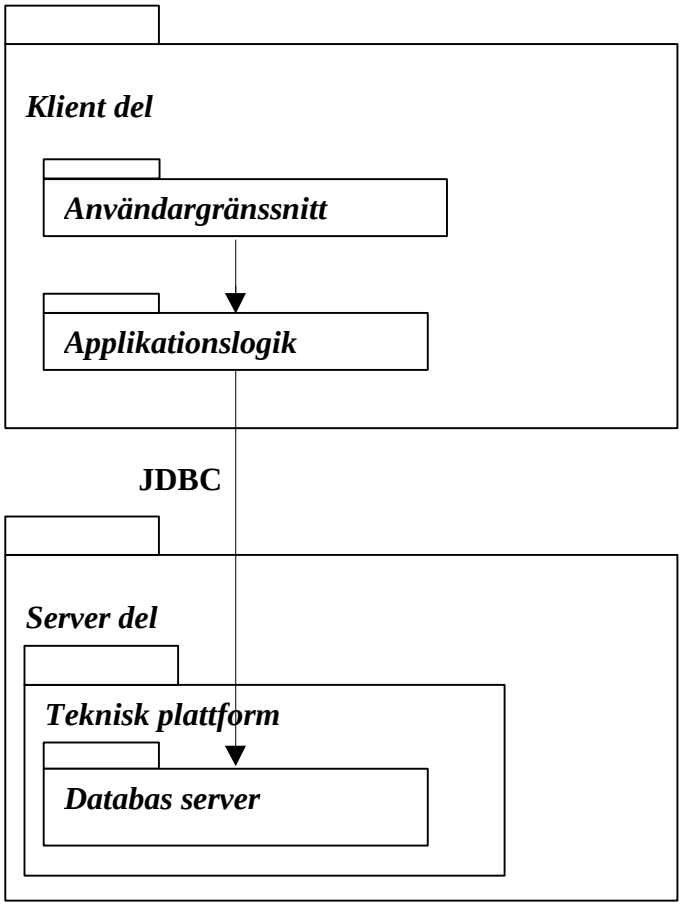
Här anges svårighetsgrad på funktionen, vilken typ av funktion det rör sig om samt vilken klass i klassdiagrammet som funktionen tillhör. Samtliga beräkningsfunktionerna hör till presentationsgränssnittet, medan funktionerna för att hämta information utnyttjas av både inmatnings- och presentationsgränssnitten. Lagringsfunktionen har tillskrivits inmatningsgränssnittet. Vissa funktioner är definierade som komplexa, dessa beskrivs mer ingående genom t ex pseudokod, vi har valt att utesluta denna beskrivning, då vi anser att denna mer är till som ett stöd vid vår programmering än som åskådligörare av våra designbeslut.

Funktion	***Komplexitet***	***Funktionstyp***	***Klasstillhörighet***
Hämta projekt	Enkel	Uppdatering	Projekt
Hämta aktivitet för ett projekt	Enkel	Uppdatering	Projekt
Hämta personer för ett projekt	Enkel	Uppdatering	Projekt
Beräkna projektkostnad	Komplex	Beräkning	Projekt
Beräkna timmar för ett projekt	Komplex	Beräkning	Projekt
Hämta aktivitet	Enkel	Uppdatering	Aktivitet
Beräkna aktivitetskostnad	Medel	Beräkning	Aktivitet
Beräkna antal timmar för en aktivitet	Medel	Beräkning	Aktivitet
Lagra insatstid	Enkel	Uppdatering	Insatstid
Radera insatstid	Enkel	Uppdatering	Insatstid
Hämta person	Enkel	Uppdatering	Person
Hämta alla projekt för en person	Enkel	Uppdatering	Person
Beräkna beläggningsgrad för en person i genomsnitt	Komplex	Beräkning	Person
Beräkna beläggningsgrad för en person under angiven period	Komplex	Beräkning	Person
Beräkna fakturering för en person	Komplex	Beräkning	Person
Beräkna antal timmar för en person i ett projekt	Medel	Beräkning	Person
Beräkna antal timmar för en person under en viss period	Medel	Beräkning	Person

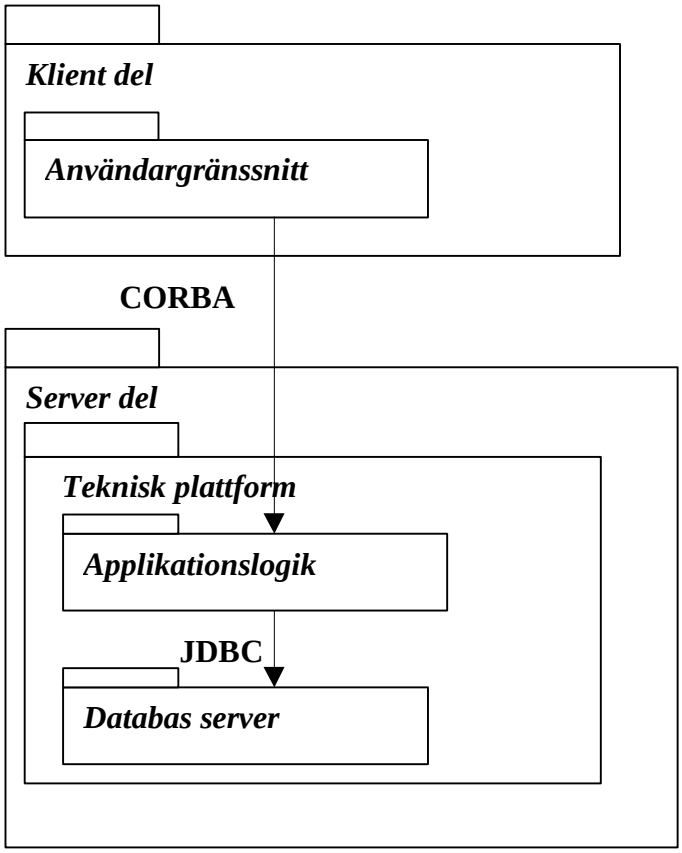
9.3 Komponentstruktur

I den feta komponentstrukturen har presentationslogiken eller användargränssnittet om man vill kalla det så, samt applikationslogiken båda placerats på klienten. Presentationslagret kommunicerar med applikationslogiken m h a objektreferenser, vilket är tillämpligt vid användning av objektorienterat programmeringsspråk. Datalogiken har placerats på servern. Applikationslagret och datalagret kommunicerar genom JDBC (Java DataBase Connectivity), ett gränssnitt som definierar klasser för att representera databaskopplingar, SQL-satser mm. Klienten i den tunna komponentstrukturen bär endast presentationslogiken. Servern består egentligen av två servrar; en applikationsserver och en databasserver. Applikationslogiken ligger som namnet avslöjar på applikationsservern och datalogiken på databasservern som i den tjocka klientens komponentstruktur. Skillnaden är här att det krävs ett extra gränssnitt för att kommunikationen mellan den tillkommande servern och presentationslogiken skall kunna realiseras. Vi har valt att använda CORBA (common object request broker architecture) för att implementera kommunikationen, detta är en metod för att kommunicera objekt mellan olika lokaliseringar. Mellan applikationslogik och datalogik är kommunikationstekniken densamma som i den feta klientstrukturen.

Den feta klienten



Den tunna klienten

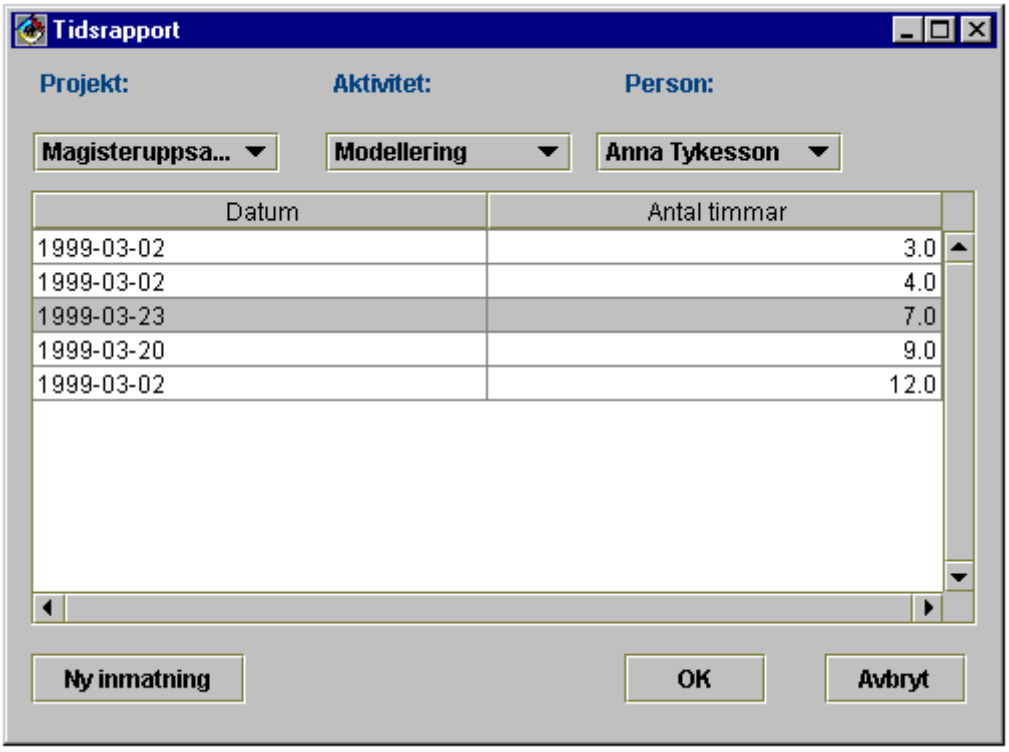


9.4 Användargränssnitt

Gränssnitten har modellerats så enkelt som möjligt. Vi har begränsat informationen till den del som är absolut nödvändig för att få ett logiskt sammanhang.

9.4.1 Inmatningsgränssnittet

I inmatningsgränssnittet under projekt finns alla tillgängliga projekt i företaget. Beroende på vilket projekt som väljs kommer tillhörande aktiviteter att visas under rubriken aktivitet. Vi har för att förenkla ytterligare utelämnat inloggningsfunktion och istället valt att användaren får ange vem denna är genom val i komboboxen under rubriken person. När användare gjort sina val kommer han/hon att få upp tidigare inrapporterade tider och han/hon kan genom en knapptryckning på ny inmatning få upp en tom rad i vilken han/hon kan skriva in en ny tid. För att spara informationen trycker användaren ok och för att ångra avbryt.



9.4.2 Presentationsgränssnittet

Presentationsgränssnittets syfte är att ge information om projekt respektive aktiviteter. Även i detta gränssnitt så styr valet av projekt vilka aktiviteter som kommer upp. I tabellen för projekt kommer användaren få information om alla aktiviteter som tillhör projektet. I tabellen för aktiviteter ges information om alla personer som har rapporterat tid för aktiviteten. Användaren måste också ange vilken information som önskas, timmar eller kostnader och kommer under vardera tabell få en sammanställning av totala kostnader respektive timmar. Designen är gjord utan möjlighet till att simultant granska timmar och kostnader, informationen som tas fram i de båda tabellerna måste m a o vara av samma typ.