

# Oppsummering av Software Architecture

Alexander Nossum  
alexander@nossum.net

17. mai 2007

## **Sammendrag**

Omfatter essensen av programvare-arkitektur og metoder for å utvikle gode arkitekturer. Beskriver kjente patterns og hvordan disse kan påvirke arkitekturens kvaliteter. Dokumentet er en oppsummering i faget TDT4250 Software Architecture avholdt våren 2007 og kommer uten noen garanti overhodet. Figurer brukt i dokumentet er hentet fritt fra internett og mangler referanser.

## **Endringer**

1. versjon

CBAM steg endret, utility-kurve

## Innhold

<b>1</b>	<b>Definisjoner av viktige begrep</b>	<b>1</b>
1.1	Software arkitektur . . . . .	1
1.2	Architectural patterns og modeller . . . . .	1
1.2.1	Architectural patterns . . . . .	1
1.2.2	Reference Model . . . . .	1
1.2.3	Reference Architecture . . . . .	2
1.2.4	Design patterns . . . . .	2
1.3	Enterprise Architecture . . . . .	2
1.4	Kvaliteter og taktikker . . . . .	3
<b>2</b>	<b>Patterns</b>	<b>3</b>
2.1	Architectural patterns . . . . .	3
2.1.1	Client-server . . . . .	4
2.1.2	Layered (multi-tier) architecture . . . . .	4
2.1.3	Model-View-Controller (MVC) . . . . .	4
2.1.4	Control loop . . . . .	5
2.1.5	Service-oriented Architecture (SOA) . . . . .	5
2.2	Design patterns . . . . .	6
2.2.1	Singleton . . . . .	6
2.2.2	Observer . . . . .	6
2.2.3	Abstract factory . . . . .	8
2.2.4	Factory . . . . .	9
2.2.5	Composite . . . . .	10
2.2.6	Facade . . . . .	11
2.2.7	Wrapper/Adapter . . . . .	11
2.2.8	Bridge . . . . .	11
2.3	Tillegg . . . . .	13
2.3.1	Builder . . . . .	13
2.3.2	Mediator . . . . .	13
2.3.3	Proxy . . . . .	13
2.3.4	Template . . . . .	13
2.4	Quality attributes . . . . .	13
2.5	Quality scenarios . . . . .	14
2.6	Taktikker . . . . .	15
2.6.1	Availability taktikker . . . . .	15
2.6.2	Modifiability taktikker . . . . .	16
2.6.3	Testability taktikker . . . . .	16
2.6.4	Performance taktikker . . . . .	17
2.6.5	Usability taktikker . . . . .	17

2.6.6	Security taktikker . . . . .	17
2.7	Funksjonelle krav . . . . .	18
<b>3</b>	<b>Dokumentasjon av arkitekturen</b>	<b>18</b>
3.1	IEEE1471 . . . . .	18
3.2	UML . . . . .	19
3.3	4+1 View model . . . . .	19
<b>4</b>	<b>Analyse av arkitekturen</b>	<b>20</b>
4.1	ATAM . . . . .	20
4.1.1	Quality Attribute Tree . . . . .	21
4.1.2	Analyse av arkitektoniske taktikker . . . . .	21
4.2	CBAM . . . . .	22
4.3	PMA . . . . .	26
<b>5</b>	<b>Business-aspekter</b>	<b>28</b>
5.1	Components Off The Shelf (COTS) . . . . .	28
5.2	Product-line . . . . .	28
<b>6</b>	<b>System integration</b>	<b>30</b>
<b>7</b>	<b>Rekonstruksjon av arkitektur</b>	<b>30</b>

## Figurer

1	Model-View-Controller . . . . .	5
2	Beck Consulting sin visualisering av SOA . . . . .	7
3	IBM developerWorks sin visualisering av SOA . . . . .	7
4	Observer pattern . . . . .	8
5	Konseptuelt klassediagram over abstract factory pattern . . . . .	9
6	Klassediagram over factory pattern . . . . .	10
7	Klassediagram over composite pattern . . . . .	11
8	Klassediagram over Facade pattern . . . . .	12
9	klassediagram over Wrapper/Adapter pattern . . . . .	12
10	Konseptuelt klassediagram over Bridge pattern . . . . .	13
11	Eksempel på et Quality Attribute Tree . . . . .	21
12	Mal for analyse av arkitektoniske taktikker i ATAM . . . . .	23
13	Eksempel på analyse av arkitektoniske taktikker i ATAM . . . . .	24
14	Eksempel på nytteverdikurver (utility curves) brukt i CBAM . . . . .	26
15	CBAM i et nøtteskall . . . . .	27
16	KJ-diagram brukt under PMA . . . . .	29

---

17	Fishbone diagram brukt i Root-Cause-Analyse under PMA . . . . .	29
18	System integrasjon i selskaper (enterprises) . . . . .	31

# 1 Definisjoner av viktige begrep

## 1.1 Software arkitektur

**Software arkitektur** En beskrivelse av strukturen til et system, software elementene som utgjør systemet, de synlige egenskapene til elementene og relasjonen mellom dem.

Denne definisjonen tar ikke for seg, eksplisitt, kvaliteter assosiert til arkitekturen noe jeg personlig mener er viktig for en arkitektur. De funksjonelle kravene er heller ikke tatt med i definisjonen. Dette skyldes at de er med å driver arkitekturen. Uten funksjonelle krav kan en vanskelig designe en arkitektur.

En arkitektur er en fullstendig høynivå beskrivelse av et system og har både gode og dårlige egenskaper knyttet til seg. Det er viktig å gjenkjenne disse egenskapene. Ved bruk av kjente patterns, modeller og taktikker kan gjenkjennelsen være lettere og arkitekturen blir ofte bedre.

## 1.2 Architectural patterns og modeller

### 1.2.1 Architectural patterns

**Architectural Pattern** En beskrivelse av forskjellige typer elementer, relasjonen mellom disse og restriksjoner på disse.

Som beskrevet over er Architectural patterns en konseptuell beskrivelse av en arkitektonisk løsning på et kjent problem. Patternet sier ingenting om funksjonalitet eller hvordan denne er fordelt. Et pattern setter begrensninger på arkitekturen samtidig som den gir framhever ulike Quality Attributes (2.4) (kvaliteter). Patterns er ikke en arkitektur i seg selv men kun essensen av en løsning på et problem. Architectural styles er også brukt som betegnelse på architectural patterns. Forvirrende nok i forhold til fysiske arkitektoniske stiler. Eksempler på Architectural Patterns er *Client-Server*, *Model-View-Controller*, (*SOA*), *Control-loop*. Se (2) for opplisting av viktige patterns med detaljer.

### 1.2.2 Reference Model

**Reference model** En oppdeling av funksjonalitet og dataflyt mellom elementer.

Mer forklart er reference model en standardisert dekomposisjon av et kjent problem slik at elementene sammen løser problemet. For at en reference model kan være god må den ha blitt spunnet ut fra “best practice” løsninger basert på mye erfaring. Reference models finnes som regel bare til veldig “voksne domener” som DBMS, kompilatorer osv. Eksempelvis bør en reference model til et DBMS gi svar på:

- De vanlige delene av DBMS'et
- Hvordan delene, rent konseptuelt, løser problemet.

### 1.2.3 Reference Architecture

**Reference Architecture (RA)** En Reference Model (1.2.2) anvendt på software elementer.

Som antydnet over anvendes en Reference Model (RF) på software elementer. Dette betyr at funksjonaliteten og dataflyten til RFen må stemme med software elementene. RF splitter opp funksjonaliteten men sier ingenting om hvor eller hvordan det faktisk skal utføres. RA beskriver hvilke software elementer som skal brukes, hvilken funksjonalitet de har og dataflyten mellom de.

**Verken** Architectural patterns (1.2.1), Reference Models (1.2.2) eller Reference Architectures (1.2.3) er fullstendige arkitekturer. Dette er kun tidligfase beslutninger som er med på å forme den fullstendige arkitekturen! Med det sagt er det svært nyttig å ha teknikkene tilgjengelig siden de som regel er den absolutt beste løsningen på vanlige problemer.

### 1.2.4 Design patterns

**Design pattern** En konseptuell beskrivelse av en løsning på et kjent problem.

Som Architectural Patterns (1.2.1) er design patterns konseptuelle løsninger på vanlige problemer. Forskjellen ligger i at design patterns er *mer lavnivå* og beskriver mer detaljerte taktikker som regel i form av høy-nivå klassediagrammer. Architectural patterns og design patterns blandes ofte sammen i hverandre, dette er ikke nødvendigvis korrekt. Architectural patterns er, etter min oppfatning, en veiledende stil på arkitekturen mens design patterns svarer på; "hvordan skal vi løse akkurat dette problemet?". Utfordringen ligger derfor i å velge design patterns som sammenfaller med den valgte arkitektoniske stilen som er valgt.

Som en analogi kan man se på bygninger. En barokk stil på et slott betyr ikke nødvendigvis at sanitæranlegget er godt utført ([Versaille](#)).

## 1.3 Enterprise Architecture

Enterprise Architecture (EA) er nødvendig i store bedrifter som har mange forskjellige IT-systemer (portfolio), forretningsprosesser og strategier. Hensikten med

EA er å skaffe seg oversikten over hele bedriften. Hovedfokus for en EA er: *forretningsstrategi, portfolio* og *forretningsprosesser* samt naturligvis informasjon. Kravene til en EA er *forenkle - forstå - formidle*.

I tillegg til portfolio over IT-systemer inkluderes også andre forretningsaspekter som halvautomatiserte prosesser og den overordnede strategien for bedriften. Fokus videre vil være på IT-delen av EA.

Store bedrifter har hundrevis til tusenvis av systemer som skal samarbeide i prosesser, alle med forskjellige interfaces, språk, databaser osv. Dette blir fort så komplekst at det er helt nødvendig å skaffe seg oversikten over bedriften. Strategier for dette er:

- Gruppere i blokker for forenkling
- Analysere bedriften for å finne hvor funksjonalitet og informasjon ligger, samt vite hva en har tilgjengelig.
- Bygge et felles kommunikasjonsspråk til formidling gjennom bedriften.

EA, generelt, støtter seg ofte på Enterprise Frameworks. Rammeverkene er modeller over hvordan man best kommer fram til og vedlikeholder en EA for bedriften. Det finnes hundrevis av slike rammeverk, TOGAF og Zachman er ganske kjente. IBM har utviklet sitt eget rammeverk spesielt med tanke på å sammenkoble IT-infrastruktur og Forretnings-strategi med EA som kobling. Der EA omslutter IT-arkitektur, forretnings-arkitektur, overgangsplanlegging og arkitekturledelse og styring. Personlig synes jeg det er en veldig bra tolkning av hva EA er.

Sett fra et IT-perspektiv holder EA en overordnet oversikt over IT-systemer tilgjengelig i bedriften og forenkler sammenslåingen/samarbeidet mellom dem.

## 1.4 Kvaliteter og taktikker

Kvaliteter til en arkitektur beskriver de ikke-funksjonelle kravene til arkitekturen. Realisasjon av kvalitetene kan gjøres ved å anvende taktikker. Selv om arkitekturen innehar visse kvaliteter gir det ikke noe garanti for at systemet vil inneha de samme kvalitetene - hvis ikke kvaliteter ved arkitekturen er klart definert og blir tatt hensyn til under implementasjon har det ingen hensikt å spesifisere kvalitetene.

## 2 Patterns

### 2.1 Architectural patterns

Som forklart i (1.2.1) er architectural patterns en mal for *hoved-ideén* til arkitekturen. Det finnes mange forskjellige arkitektoniske patterns - alle med sine indi-

viduelle egenskaper og kvaliteter, tilpasset et spesielt problem. I dette dokumentet blir kun de mest vanlige framhevet.

### 2.1.1 Client-server

Client-server pattern er en ren konseptuell beskrivelse av arkitekturen. Hovedfokus ligger i å beskrive at det eksisterer en klient som kommuniserer med en server. Typisk anvendelse er å skille mesteparten av logikk og data fra brukergrensesnitt. Dette forenkler betraktelig deling av data, endring av logikk og gir enkel skalerbarhet på klientsiden.

### 2.1.2 Layered (multi-tier) architecture

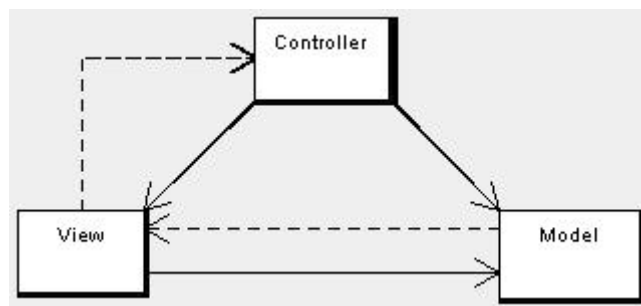
Flerlagsarkitektur er en veldig anerkjent stil. Stilen er lineær i den forstand at kommunikasjon går direkte opp og ned mellom lagene, altså ikke hopper over et lag. Den mest anvendte formen er trelagsarkitektur hvor lagene er delt opp i *presentasjon-logikk-data*. Viktig poeng er at lagene kun kommuniserer med nabo-laget. Eksempelvis vet ikke presentasjonslaget noe om datalaget, mens logikklaget vet om både presentasjonslaget og datalaget.

### 2.1.3 Model-View-Controller (MVC)

MVC er en tilpasning av trelagsarkitekturen i den forstand at den har som mål å skille presentasjon, logikk og data fra hverandre. Kommunikasjonen mellom elementene er dermed **ikke lineær** som vist i figur 1. Patternet er ekstremt populært og anvendes i nesten ethvert “standard” system, spesielt har det blomstret opp veldig hyppig bruk av MVC i websystemer.

Model kan sees på som et datalag, dog som regel med mer logikk og interfaces enn for eksempel en ren database. Model har ansvaret for å gi enkel og intuitiv tilgang til data i systemet. En viktig bemerkning på model er at den ikke skal endre data i noe stor grad, den skal med andre ord ikke inneholde domene-/business-logikk.

View er presentasjonslaget. Hovedoppgaven er å ta imot brukerhandlinger og presentere data fra model. Brukerhandlinger med respons som innebærer logikk sendes til Controller for videre behandling. Vanligvis innebærer dette brukerinntak som vil endre data i model. Konseptuelt har View full synlighet til model. I praksis begrenser man denne tilgangen ved å stadfeste at view sin kobling mot model kun er for *visning av tilstanden til model*. Essensielt vil dette begrense til kun leserettigheter. Model bør forhindre tilgangen view har, men samtidig ikke vite noe verken view eller controller. Oppsummert er view ansvarlig for å motta data som sendes til controller og presentere status til model.



Figur 1: Model-View-Controller konseptuelt diagram. Hele piler indikerer direkte kjennskap, stiplet indikerer indirekte kjennskap/kommunikasjon

Controller sin hovedoppgave er å modifisere data i model. Utløsende faktor er en handling fra view. Controller innehar logikk for å endre model slik at data i model forblir korrekt.

Aktiv bruk av observering kan vise seg å være svært lønnsomt ved bruk av MVC siden view henter data fra model mens controller endrer data i model. Se også (2.2.2).

Oppsummert kan MVC virke komplekst i forhold til trelagsarkitektur. Fordeelen er at controller tilsvarende logikklaget i trelags. ikke kopierer data fra model og sender det direkte videre til presentasjonslaget. Dette minsker unødvendig kommunikasjon mellom lagene og unødig funksjonalitet i controller. MVC krever derimot mer erfaring fra utviklere og mer høynivå funksjonalitet i de forskjellige elementene/lagene. Se også [phpwact:MVC](#) og [Wikipedia](#).

#### 2.1.4 Control loop

#### 2.1.5 Service-oriented Architecture (SOA)

Service-oriented architecture er et relativt nytt begrep innefor software arkitektur. Det finnes ingen entydig definisjon på begrepet og dermed har mange ulike begrep om hva det er og ikke minst hva det ikke er. Personlig liker jeg IBM sin definisjon sammen med Gartner (Ibrahim and Long 2007). IBM tilbyr flere definisjoner basert på synspunktet til brukeren; business, architect, developer. Den mest interessante i denne sammenhengene er definisjonen for arkitekten:

En arkitektonisk stil som krever en tjenestetilbyder, en anmoder og en tjeneste beskrivelse.

Omfatter et sett av teknikker, patterns og krav som adresserer karakteristikk som: modularitet, innkapsling, løs kobling, skille av ansvar, gjenbruk, sammensetningsmuligheter og engangs-implementasjon.

Essensen er å dele opp systemet i komponenter som tilbyr tjenester. Disse tjenestene kan samles i en sammensetning (service bus) slik at de utgjør hele eller deler av et system. Hovedideen er særdeles løs kobling mellom tjenestene og klientene som bruker de.

SOA er en stor og kompleks arkitekturstil som best passer på enterprise-nivå og har i tillegg likheter med Enterprise Architectur (1.3).

Hovedkomponentene i SOA er tjenestene. Tjenestene må ha detaljert beskrevet interfaces og være atomiske. Hensikten er at anmoderen skal kun bekymre seg over hva som tilbys - ikke hvordan det gjøres. I tillegg forenkler dette sammenslåingen av tjenester til delsystemer.

Stor vekst i bruk av web som grensesnitt til systemer har antakeligvis vært med på å skape SOA. SOA lar seg med stor enkelhet anvende på forskjellige systemer. For eksempel kan en tenke seg et system som både skal ha webgrensesnitt og en desktopapplikasjon hvor begge deler mesteparten av funksjonaliteten samt kvaliteter. Med SOA trenger utviklere kun tenke på hva de trenger til sitt system og velge tjenester deretter - uten noen som helst tanke på andre systemer eller hvordan tjenestene fungerer. Dette er en ekstremt stor fordel i enterprise-sammenheng. Forretningsprosesser kan modelleres som sammenslåtte tjenester slik at ledelsen av prosessen garantert innehar eierskap av business-logikken.

SOA er som nevnt på et ganske tidlig stadiet, det finnes ingen enighet om definisjonen eller noe entydige patterns på det. Personlig liker jeg Beck Consulting sin visualisering (2) og IBM sin visualisering (3). Ingen av disse er fasiter, men jeg synes de tar godt fatt i hva SOA egentlig dreier seg om.

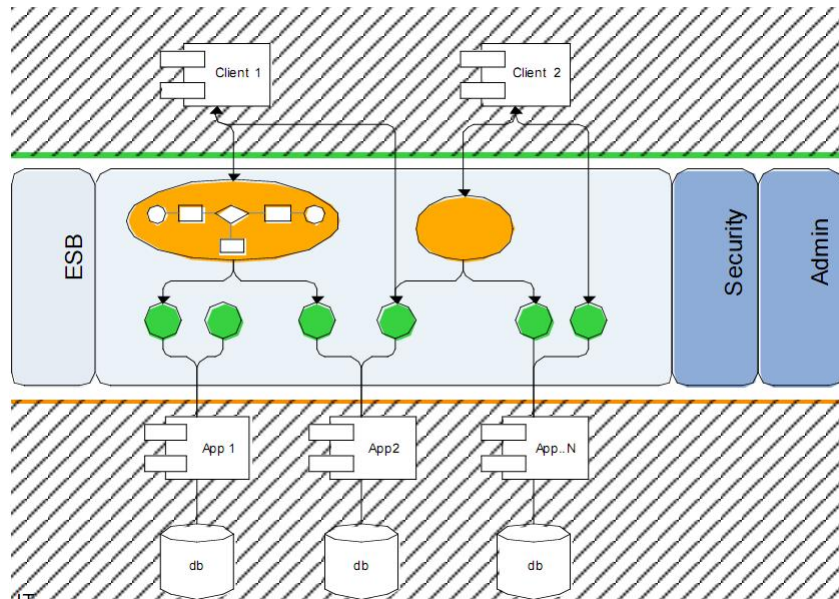
## 2.2 Design patterns

### 2.2.1 Singleton

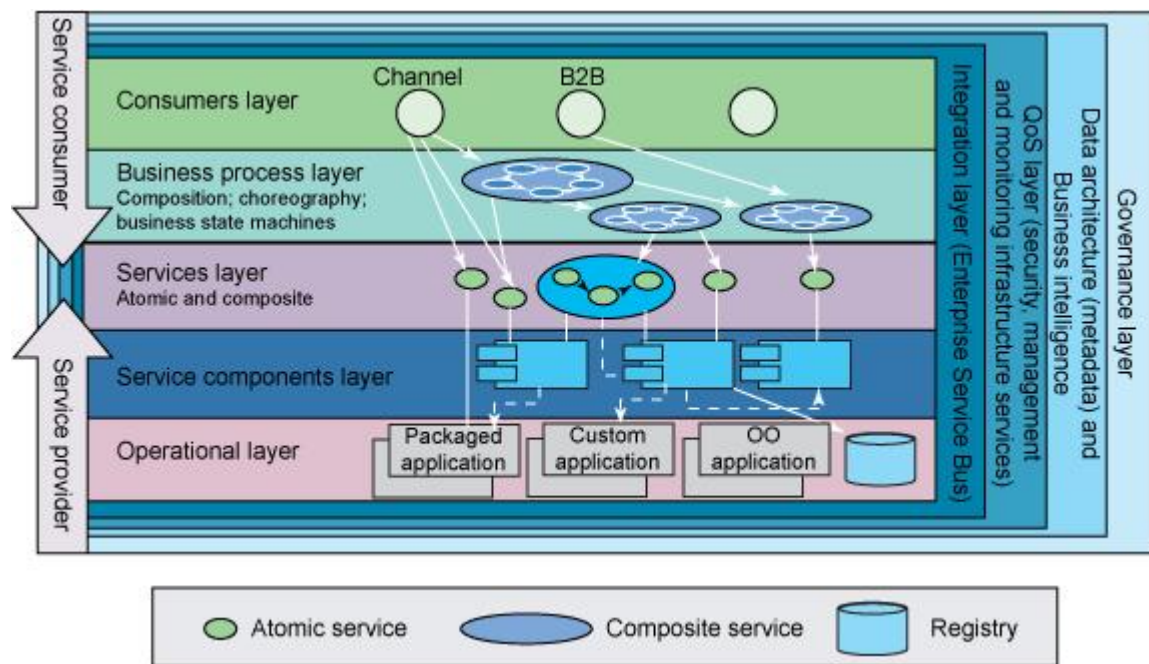
Singleton sitt formål er å sikre at det kun finnes en instans av objektet hvor patternet er anvendt. Dette garanterer at informasjonen i objektet er likt for alle som bruker det. Singleton gjør dette ved å gjøre konstruktøren til klassen privat og lage; *getInstance()* og *Object instance* som sjekker om det allerede finnes en instans av objektet eller om den skal lage en ny og sparer den i *instance*.

### 2.2.2 Observer

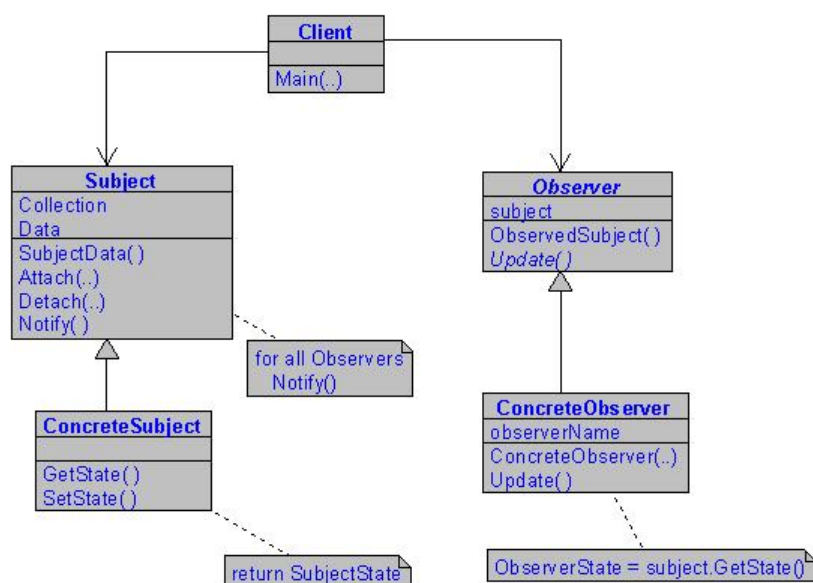
Observer pattern forenkliggjør varsling om endring hos et objekt. Analogien til en forelesning er ganske bra. Foreleseren er objektet som sier noe (endrer informasjon) mens studentene melder seg som abonnenter av foreleseren (observerer). Foreleseren vet ingenting om hvem som observerer henne, alt hun gjør er å si noe (kalle *hasChanged()*). Studentene har meldt seg som abonnenter og mottar *update()* med



Figur 2: Beck Consulting sin visualisering av SOA



Figur 3: IBM developerWorks sin visualisering av SOA



Figur 4: Observer pattern

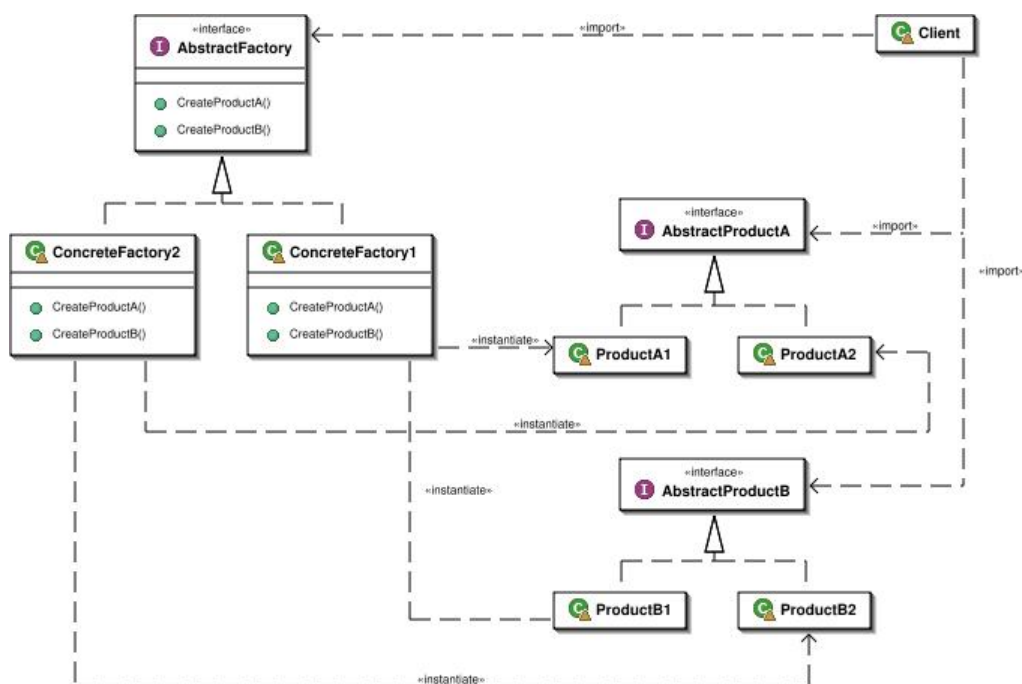
informasjonen etter hver *hasChanged()*. Analogien er kanskje noe søkt, men prinsippene er der. Et annet eksempel er reklameskjermer. Skjermene abonnerer på endringer hos ett objekt med den faktiske reklamen (informasjonen). Når objektet med reklamen endrer til en ny reklame vil automatisk alle skjermene også gjøre det.

Essensen er altså at objekter skal kunne lytte på endringer hos et objekt uten at dette objektet vet om de. Dette fremmer en løs kobling. Se figur 4 for et konseptuelt klassediagram over Observer pattern.

### 2.2.3 Abstract factory

Abstract Factory sin hensikt er å abstrahere vekk detaljer ved instansiering av en gruppe (familie) objekter fra klienten som ønsker funksjonalitet fra gruppen. Dette oppnås ved å bruke abstract egenskapen.

Eksempelvis vil man ikke at klienten skal ta hensyn til hvilken plattform systemet kjøres på. Utvikling av GUI er spesielt godt egnet som eksempel på dette. Klienten sitt mål er å lage en knapp ved kun å instansiere AbstractFactory og kalle på *createButton()*. Uten patternet må klienten først sjekke hvilken plattform det kjøres på og så velge hvordan den vil lage knappen ut i fra det. Abstract factory skjuler denne logikken ved at den selv finner ut hvilken plattform det kjøres på



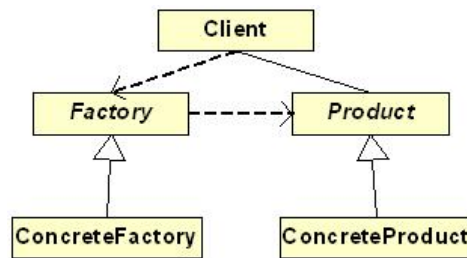
Figur 5: Konseptuelt klassediagram over abstract factory pattern

og instansierer objekter deretter. I tillegg definerer factory'et abstrakte funksjoner som må være oppfylt i arvende klasser (*createButton()*). De faktiske klassene som innehar rett funksjonalitet arver fra *AbstractFactory* og implementerer dermed de abstrakte funksjonene. Dette garanterer at rett klasse blir instansiert og returnert og at den klassen har implementert korrekt funksjonalitet, *createButton()*. Dette kan naturligvis skaleres opp ved å definere konkrete factorys under *AbstractFactory* om i igjen kaller på kaller på et rett sluttprodukt(er).

Et problem med *AbstractFactory* er at de konkrete produktene må inneha ganske lik grunnfunksjonalitet siden metodene blir definert abstrakt i *AbstractFactory* klassen. Det er derfor kun hensiktsmessig å anvende patternet på komponenter som naturlig hører sammen, som familier. Se figur 5 og (WikipediaAbstractFactory 2007) for diagrammer, eksempler og detaljerte forklaringer.

### 2.2.4 Factory

Factory pattern sin hovedoppgave er å lage objekter til en klient. Klienten skal kunne få rett objekt uten å vite nøyaktig fra hvilken klasse det kom fra. Dette gjøres ved at factory klassen innkapsler klasser som den kan lage og innehar logikk som bestemmer hvilken som er rett klasse å returnere til klienten - ofte basert på



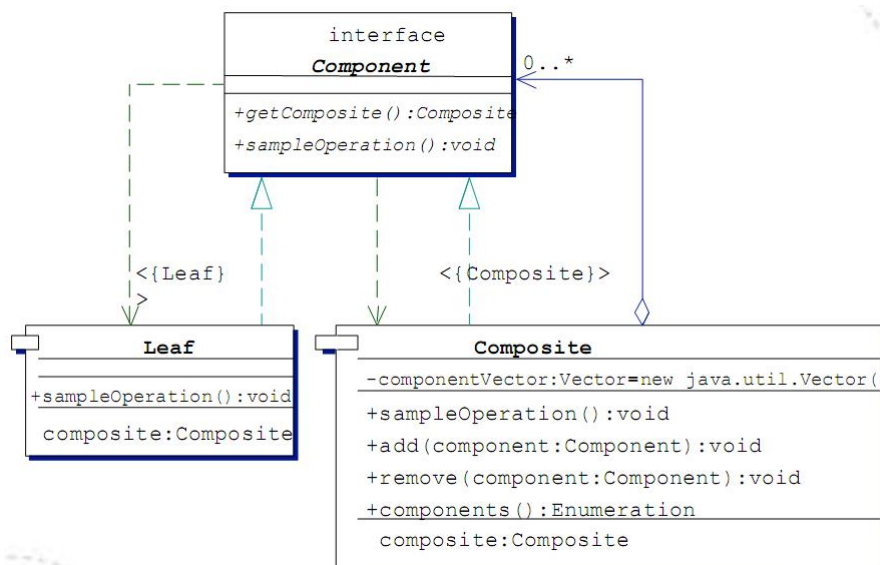
Figur 6: Klassediagram over factory pattern

argumentene i metoden.

Et eksempel på anvendelse av factory pattern er et system for bildeinnlesning. Systemet har en klasse for vært bildeformat som alle implementerer et interface ImageReader. Klienten som skal lese bildet må altså inneha logikk om hvilket bilde den har som skal leses og instansiere riktig klasse etter dette. Anvender vi factory pattern på dette problemet adskiller vi bestemmelseslogikken og klienten trenger kun å tenke på funksjonaliteten heller enn bestemmelsen av klasser. Se figur 6 for klassediagram over factory pattern. Diagrammet er en generell beskrivelse av patternet og fremhever at produktet og factory kan arve fra abstrakte klasser for å definere og framtinge like metoder. I den virkelige verden beskrives klasser med hovedoppgave å lage og returnere objekter for factories.

### 2.2.5 Composite

Composite pattern har som mål å samle noder og løvnoder i en samling for så enkelt å kunne kjøre individuelle metoder på alle nodene i samlingen. Tankgangen er at composite-klassen i tillegg til andre "løvklaser" arver og implementerer funksjonalitet fra en abstrakt klasse, *Component*. Composite-klassen er ansvarlig for å aggregere (samle) instanser som er av typen (arvet) *Component*. Siden composite-klassen i seg selv arver fra denne klassen muliggjør dette for lenking av samlinger (composite). Altså, en composite samler på komponenter og kan dermed samle på instanser av seg selv. Dette kan være nyttig for å danne trestrukturer og videre skoger. Implementasjonen av composite-pattern bestemmer hvordan samlingen faktisk lagres. Patternet i seg selv definerer ikke struktur for rekkefølgen av elementer i samlingen, men dette kan oppnås under implementasjon. Se også figur 7 for en høynivå beskrivelse av Composite-pattern.



Figur 7: Klassesdiagram over composite pattern

### 2.2.6 Facade

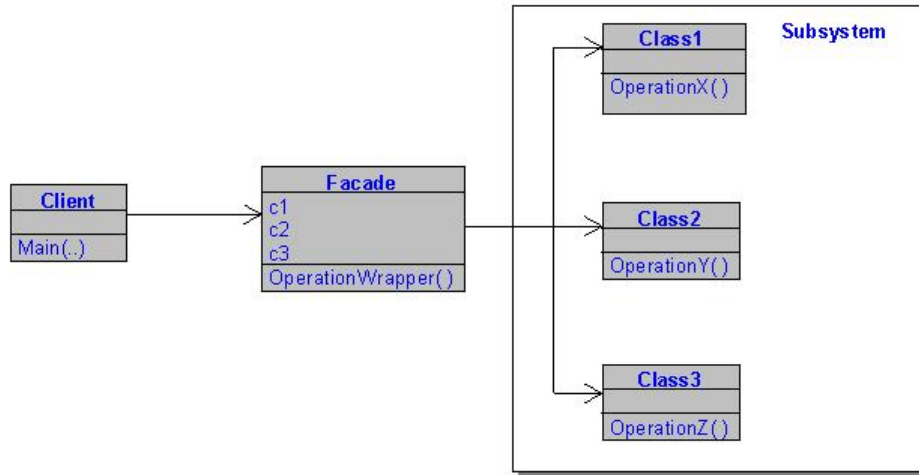
Facade pattern har som hensikt å skjule struktur og logikk i et subsystem fra klienten som ønsker å bruke det. Typisk bruk er å hindre koblinger direkte inn i en pakke ved å legge en fasadeklasse som mellomkobling mellom klienter og interne klasser i pakken. Facadepattern fremmer spesielt modifiability til systemet og muliggjør for enklere parallellutvikling. Se figur 8.

### 2.2.7 Wrapper/Adapter

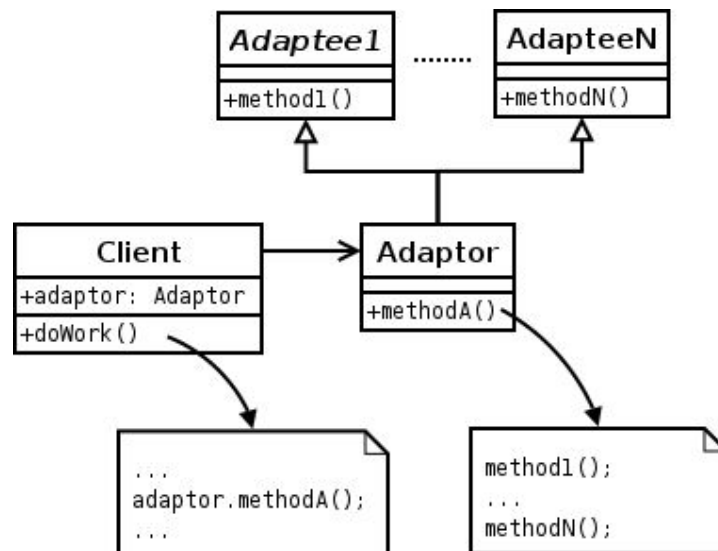
Wrapper er også kjent som adapter-pattern. Formålet med patternet er å få komponenter til å kunne kommunisere når de i utgangspunktet ikke kan. For eksempel hvis en klient skrevet i Java vil bruke et system i Fortran finnes det ikke naturlig støtte for dette i Java. En wrapper må legges rundt Fortransystemet. Wrapperen sørger for et naturlig grensesnitt til klienten og skjuler grensesnittet til Fortransystemet. Wrapper-patternet egner seg spesielt godt når et system har begrensninger med at legacy-systemer skal brukes sammen med det nye systemet. En annen hyppig anvendelse er for hardware-drivere. Se figur 9 for klassesdiagram.

### 2.2.8 Bridge

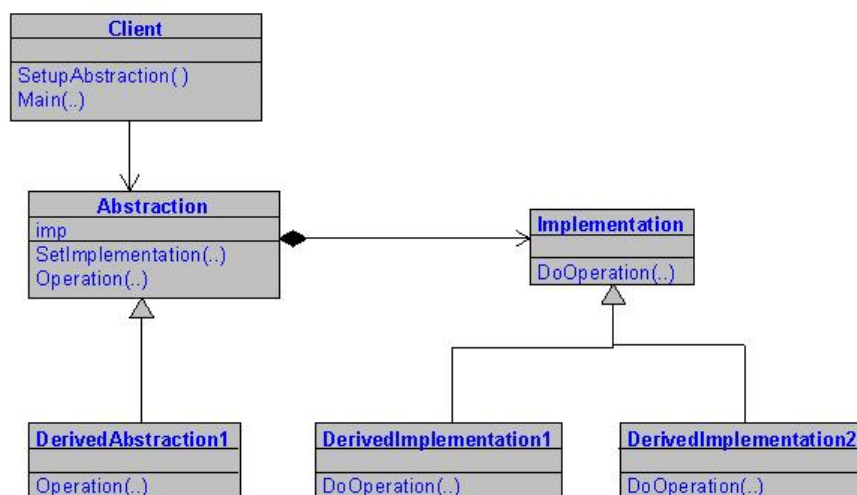
Bridge-pattern har som mål å frakoble en abstraksjon fra implementasjonen så begge fritt kan variere/endres.



Figur 8: Klassediagram over Facade pattern



Figur 9: klassediagram over Wrapper/Adapter pattern



Figur 10: Konseptuelt klassediagram over Bridge pattern

Klienten bruker *DerivedAbstraction* og de forskjellige *DerivedImplementations*. Grensesnittet til disse garanteres ved hjelp av abstraksjonene som de implementerer slik at den faktiske funksjonaliteten i implementasjonen lett kan endres uten at klienten trenger å ta noen hensyn. Se figur 10 for klassediagram og Wikipedia for eksempler og detaljert beskrivelse (WikipediaBridgePattern 2007).

## 2.3 Tillegg

### 2.3.1 Builder

### 2.3.2 Mediator

### 2.3.3 Proxy

### 2.3.4 Template

## 2.4 Quality attributes

Quality attributes er en del av de ikke-funksjonelle kravene til systemet. Definert på forhånd legger de begrensninger på hvordan arkitekturen kan lages. Definert i ettertid vil fungere som en analyse av den valgte arkitekturen.

**Availability** Beskriver hvor tilgjengelig systemet er under kjøring. Med tilgjengelig menes som oftest om systemet eller deler av systemet er brukbart.

**Modifiability** Beskriver graden av endring som kan gjøres på systemet. Graden av endring kan ofte være subjektivt, men som regel menes; hvor lett/fort kan ny funksjonalitet legges til eller gammel taes vekk.

**Testability** Beskriver hvor enkelt systemet eller deler av systemet er å teste. Det må beskrives detaljert hvordan testen skal utføres for at beskrivelsen skal gi mening. Testability henger ofte sammen med modifiability.

**Performance** Beskriver ytelsen på systemet eller deler av systemet. Ytelse måles som regel i tid. Dog det er mulig å beskrive ytelse på andre mer objektive måter.

**Usability** Brukervennlighet sett fra et arkitektonisk synspunkt har ikke sammenheng med det eventuelle grafiske grensesnittet eller noe lignende. Derimot er det graden av brukervennlighet lagt til rette av arkitekturen. Rollback, backup og lignende er eksempler på brukervennlighet.

**Security** Beskriver hvor sikkert systemet er mot uønsket aktivitet.

Kvalitetsegenskapene kan ofte (hvis ikke alltid) være overlappende. Det er et poeng å derfor velge formuleringer nøyaktig og presist. Et valg av en kvalitet er, nesten alltid, garantert et bytte mot en annen kvalitet (tradeoff). Det er viktig å reflektere rundt og være oppmerksom på disse byttene samt dokumentere valgene gjort.

## 2.5 Quality scenarios

Det er meningsløst å si at et system eller en arkitektur skal inneha en eller flere kvalitetsegenskaper uten å spesifisere nøyaktig hva en legger i den kvaliteten. Dette understøttes av den menneskelige oppfatningene av kvaliteter, en person kan fort være av oppfatningen at usability kun går på grafisk design mens en annen kan mene det går kun på databasedesign - ingen av delene er feil. Uten detaljerte kvalitetsegenskaper er det så godt som umulig å objektivt si noe spesifikt om kvaliteter. Quality scenarios er en måte å spesifisere kvalitetsegenskaper med hensyn på de ulike egenskapene svært detaljert, objektivt og dermed utvetydig. Hvordan man velger fremstillingen av et scenario er likegyldig. Det viktige er at hovedpunktene kommer fram. Et forslag til disse er:

**Source:** Hvem utløser scenarioet

**Stimulus:** Hva er den utløsende handlingen

**Artifact:** Hva i systemet blir påvirket

**Environment:** Hvordan er systemtilstanden

**Response:** Hva er responsen til handlingen

**Response measure:** Hvor lang tid tar responsen

Eksempel på et modifiability scenario

**Source:** Developer

**Stimulus:** Wishes to change the user interface

**Artifact:** UserInterface-layer

**Environment:** Maintenance

**Response:** Modification implemented with no side-effects

**Response measure:** 1 day - experienced developer

Et viktig hovedpoeng i beskrivelsen av scenarioet er at det skal **være testbart!**  
Selv scenarioer på testability.

## 2.6 Taktikker

Taktikker er metoder for å oppnå kvaliteter i systemet. Som oftest tar én taktikk sikte på å fremme én kvalitetsattributt (2.4. Arkitektoniske strategier er naturlig nok et sett med taktikker for å oppnå ønskede kvaliteter til et system.

Dette dokumentet vil ta for seg taktikker gruppert under kvalitetsattributter. Taktikkene som er nevnt er absolutt ikke de eneste eller de beste, men er eksempler på taktikker og deres relasjon til kvalitetsattributter.

### 2.6.1 Availability taktikker

Mange tilgjengelighetstaktikker ligger allerede implementert i OS, Språkplattform, DBMS og lignende. Availability måles ofte som et prosentmål på oppetid til systemet. Det er to forskjellige hendelser som kan svekke tilgjengeligheten til systemet; *Failure* og *Fault*.

Failure (sammenbrudd) er den mest alvorlige. Systemet klarer da ikke å levere funksjonaliteten slik det skal gjøre, dette er merkbart for brukeren!

Fault (feil) er mindre alvorlig og er en feil som potensielt kan manifestere seg i en Failure. Systemet kan for eksempel oppdage og dekke over en Fault.

Det er vanskelig å sikre seg direkte på en Failure. Fault er enklere å oppdage og siden Fault er starten på en Failure vil en taktikk som forhindrer Faults også forhindre Failures. Tre klasser på taktikker til availability er:

**Fault detection** Oppdage feil før sammenbrudd. Forskjellige deler av systemet passer på hverandre og oppdager feilen. Bruk av livssignaler eller tegn på upassende oppførsel er vanlig.

**Fault recovery** Gjenoppretting av en feil så fort og/eller bra som mulig. Redundans (parallelitet/backup) sikrer gode gjenopprettingsmuligheter.

**Fault prevention** Unngå alle feil. Eventuelt hindre at feil har noen påvirkning på systemets tilgjengelighet.

Redundans (*redundancy*) vil si at systemet kjører parallellt på to eller flere individuelle plattformer og/eller at det kjøres to (eller flere) systemer på to plattformer implementert ulikt, men med identisk funksjonalitet.

### 2.6.2 Modifiability taktikker

Taktikker for å øke graden av endring til et system innebærer som regel oppsplitting av elementene i systemet og minimering av kommunikasjonen mellom dem. I systemer som har en lang levetid med varierende funksjonalitet er modifiability ekstremt kritisk for systemet. Mange patterns, architectural og design, har til hensikt å forbedre modifiability'en til systemet.

Generelt kan taktikkene for modifiability deles inn i tre klasser

**Samle endringer** Fordel ansvar til bestemte elementer i systemet. Endringer vil da kun påvirke deler av systemet.

**Unngå ringvirkninger** Unngå at en endringer i en del av systemet medfører endring i en annen del av systemet.

**Utsett bindingstid** Utsett sammenbindingen (kodeoversettelsen[clue]??) av systemets deler. Muliggjør for endringer under deployment-fase (spredningsfase). Fokuserer på runtime og loadtime.

### 2.6.3 Testability taktikker

Taktikker for å fremme testbarheten til et system henger ofte sammen med endringsgraden til systemet. Hvis systemet har løst koblede elementer med veldefinerte ansvarsområder vil testbarheten som oftest også være høy.

Generelt kan testbarhet deles inn i to hoveddomener

**Black-box** Testing under kjøring av ferdig system eller ferdig del av systemet.

**White-box** Enhetstesting internt i systemet. Primært under implementasjon.

Testbarheten for black-box legger hovedvekt på testing av input og output fra systemet uten å vite hva som faktisk skjer i systemet. For white-box testing er det mer aktuelt å skape et testmiljø som er styrt av utvikleren.

Nøkkelord for taktikker som går på testbarhet er: ansvarsfordeling, test-states, sentralisert exception-handling, utskrift av systemtilstand og lignende.

#### 2.6.4 Performance taktikker

Taktikker for å fremme ytelsen til systemet vil si at systemet skal effektivt håndtere handlinger og utnytte tilgjengelige ressurser.

Performance er den kvaliteten som er mest utsatt for tradeoffs. De fleste andre kvaliteter går som regel ut over ytelsen på systemet.

Generelle taktikker for å fremme ytelse er

**Forlange ressurser** Redusere ressurser som blir brukt til en handling. Redusere reaksjonene til en handling. Binde/ta eierskap over ressurser.

**Planlegge bruk av ressurser** Utnytte parallellitet (parallellberegning), innføre redundans, kjøre på raskere plattformer osv.

**Styring av ressursbruk** Prioritere reaksjoner og planlegge utførelse deretter.

#### 2.6.5 Usability taktikker

Taktikker for brukervennlighet har ingenting med grafisk design eller lignende, men går på brukervennlig funksjonalitet. Målet er at systemet skal være lettere eller bedre å bruke for sluttbrukeren. Taktikkene deles opp i *runtime*- og *design-time*-taktikker.

**Runtime** Vedlikeholde modeller av: *oppgaven*, *brukeren* og *systemet*.

**Designtime** Typisk skille brukergrensesnitt fra resten av systemet (2.1.3, 2.1.2).

#### 2.6.6 Security taktikker

Sikkerhetstaktikker gjør systemet sikrere mot eventuelle farer systemet er utsatt for. Taktikkene deles i tre klasse

**Unngå angrep** Typisk hindre uvedkommende tilgang til systemet og hindre avlytting.

**Registrere angrep** Systemet registrere at det skjer et angrep og handler deretter.

**Gjenopprettelse etter angrep** Systemet går tilbake til en frisk tilstand og identifiserer angriperen som utløste angrepet.

## 2.7 Funksjonelle krav

Funksjonelle krav beskriver hvilke konkrete funksjoner systemet skal kunne utføre. Disse kravene utgjør en viktig del av drivkraften bak et system men er kun *en del* av driverene.

Dessverre finner man ofte at funksjonelle krav er de eneste kravene til systemet. Dette medfører at verken kunde eller utvikler har noe de skulle ha sagt med tanke på kvaliteter til systemet. Et konseptuelt eksempel kan være et system med sorter en million tall som eneste krav. Utvikler kan sortere tallene med en algoritme som bruker to dager og allikevel er systemets krav oppfylt, selv om kunden ikke mener det. Skalert opp kan dette bli et svært stort problem!

Funksjonelle krav bør beskrives presist og entydig i nært samarbeid med kunden. Kravene bør kun gå på ren funksjonalitet og bør ikke sette noen eksplisitte begrensninger på design av systemet.

## 3 Dokumentasjon av arkitekturen

### 3.1 IEEE1471

Fullstendig dokumentasjon av arkitektur til et system følger som regel ingen spesielle konvensjoner eller standarder. Dermed er ofte dokumentasjonen mangelfull. Dette har IEEE adressert i sin “Recommended Practice” (IEEE1471 2000). I denne artikkelen beskrives en fullstendig dokumentasjon av arkitektur til et system. Essensen av beskrivelsen er *begrunnelse* av valg tatt, såkalt rationale og hensyn til ulike “stakeholders” av systemet. Personlig tror jeg vektleggingen av disse punktene kan begrunnes med at utviklere er flinke med detaljerte diagrammer brukt til implementasjon men glemmer eller utelater dokumentasjon for stakeholders som ikke skal programmere. Dette er vel så viktig å få med. Gode kommunikasjonslinjer mellom kunde og utvikler er essensielt og fører til trygghet hos begge parter.

Personlig mener jeg IEEE1471 er en svært god standardisering for dokumentasjon og beskrivelse av arkitekturer. Jeg innser at visse kapitler kan virke svært unødig under utvikling, men gevinsten tror jeg er formidabel. Det er dermed ikke sagt at denne type beskrivelse passer alle systemer. Standarden er svært omfattende og egner seg best til store arkitekturer hvor oppkjøper og utfører er klart separerte. Men selv på små arkitekturer mener jeg essensen av standarden fortsatt

kan, og bør, brukes - spesielt med tanke på forskjellige viewpoints, rationale, kjent inkonsistens og spesifisering av stakeholders til systemet.

## 3.2 UML

Unified Modelling Language er et svært omfattende diagramspråk primært rettet mot bruk i informasjonsteknologi. Språket er delt opp i forskjellige diagramtyper med forskjellige notasjoner. Diagrammene er best egnet til beskrivelse av objekt-orienterte systemer, men kan også anvendes på prosedyre-orienterte språk. Omfanget på dette dokumentet tar ikke for seg detaljerte beskrivelser av UML og der refereres til litteratur som (Fowler 2004) og (*UML pocket reference* 2006) for videre detaljering.

Et viktig punkt å merke seg når man står ovenfor et diagram som tilsynelatende er beskrevet med UML er at det er veldig få som faktisk strengt følger den korrekte UML-notasjonen. Hvis det ikke er vedlagt en beskrivelse av diagrammet eller en egen tegnforklaring må en prøve som best en kan å forstå konseptet til det som er beskrevet. En sunn holdning ovenfor diagrammer er å aldri ta for gitt at de følger UML, men smile hvis de gjør det. Som skrekkeksempler på dette kan nevnes at både IBM og Microsoft har publisert artikler med diagrammer beskrevet med “slurvete” UML uten noe videre kommentering.

## 3.3 4+1 View model

Artikkelen 4+1 View model (Kruchten 1995) er et forslag til diagrambeskrivelse av en arkitektur. Ideen er at arkitekturen skal vurderes og brukes av forskjellige interessenter (stakeholders) som alle har forskjellige synspunkter og bekymringer knyttet til systemet og arkitekturen. Løsningen foreslått er å ta hensyn til de forskjellige synspunktene ved å innføre “views”. Som antydnet i tittelen er det lagt opp til 4 forskjellige views samt ett tilleggsvue (+1). De fire individuelle views'ene er

**Logical view** Objekt-modellen over systemet. Typisk klassediagram, enten konseptuelt eller detaljert. Et veldig nyttig view som bør inkluderes.

**Process view** Beskriver prosesser eller samhandling i systemet og mellom deler av systemet. Typisk beskrivelse av den mest komplekse prosessen for å fremheve den arkitektoniske grunnideen.

**Development view** Beskriver den statiske fordelingen av hovedelementene i systemet. Typisk pakkestrukturen eller lagstrukturen.

**Physical view** Den fysiske oppdelingen av systemet. Beskriver hvor elementer i systemet faktisk kjøres eller befinner seg.

I tillegg til de fire er det som nevnt et tilleggsview (+1). *Scenario view* har som hovedoppgave å sette alle view'ene i en kontekst for lettere å beskrive konsistensen mellom dem. Dette gjøres ofte med scenarioer hvor det utheves hvilke komponenter som er i aksjon. Dette view'et er egentlig overflødig i den forstand at det overlapper alle de andre. Allikevel er det et veldig nyttig view som blant annet kan gi en kunde lettere forståelse av arkitekturen beskrevet i de 4 andre view'ene.

The 4+1 View model er en relativt gammel artikkel og bruker en særdeles kryptisk notasjon i diagrameksemplene som absolutt ikke er til etterfølgelse. Ideen er derimot svært god og bør anvendes der det er mulig. Det er dermed ikke sagt at alle view'ene trenger å brukes hver gang. Arkitekten må velge hvilke view som er viktige og nyttige, eller kombinere views'ene slik at resultatet blir tilfredstillende. Et viktig poeng å merke seg er dokumenteringen av valg som er tatt og valg av notasjon, også nevnt under 3.2, 3.1 og (IEEE1471 2000).

## 4 Analyse av arkitekturen

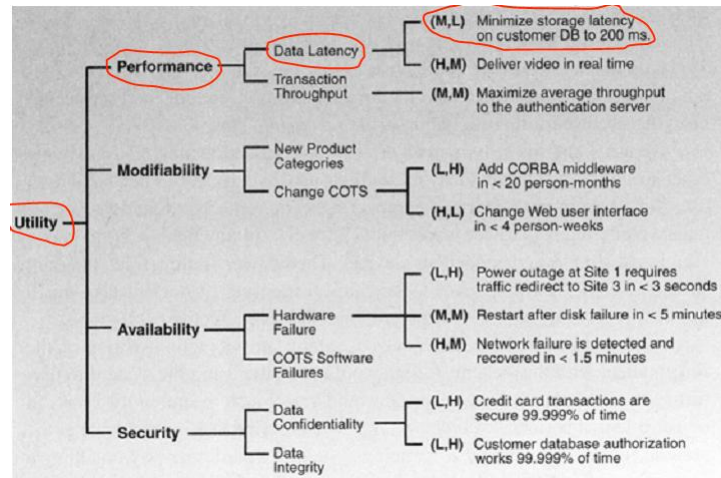
### 4.1 ATAM

**Architecture Tradeoff Analysis Method** er en prosess for å vurdere en arkitektur. ATAM kan anvendes på uferdige og eksisterende arkitekturer. Fordeler med ATAM er

- Stakeholders (interessenter) blir samlet på et møte.
- Kvalitetsmål blir veldefinerte
- Kvalitetsmål blir prioritert i forhold til hverandre
- Beskrivelsen av arkitekturen blir generelt bedre
- Deltakerene får nyttig erfaring ved refleksjon over arbeidet.

Proessen er delt opp i ni forskjellige deler

1. Presentasjon av ATAM som metode
2. Presentasjon av business driver (5)
3. Presentasjon av arkitekturen
4. Klassifisering av arkitektoniske taktikker brukt (approaches)
5. Generering av Quality Attribute Tree (4.1.1)



Figur 11: Eksempel på et Quality Attribute Tree

6. Analyse av de arkitektoniske taktikkene (4.1.2)
7. Brainstorm (vurder) og prioriter kvalitetsscenarioer
8. Analyser arkitektoniske taktikker på nytt.
9. Presenter resultater

Mange av punktene over er selvforklarende eller mindre viktige, derfor vil fokus hvile på punkt 5 og 6.

#### 4.1.1 Quality Attribute Tree

Quality Attribute Tree, også kalt utility-tree, er en metode for å enklere vurdere og prioritere kvalitetsscenarioer til systemet. Hovedtanken er å lage en trestruktur hvor grenene deles opp i kvalitetsattributter og nodene er faktiske scenarioer. Det er scenarioene som blir vurdert. Vurderingsformen er to attributter; *Viktighet* og *Vanskelighetsgrad*. Typisk verdiområde for disse er *Lett*, *Medium* og *Vanskelig*. Dette gjør det relativt lett for erfarne utviklere å klassifisere scenarioene som også implisitt prioriterer de. Figur 11 viser et eksempel på et Quality Attribute Tree, her er L,M,H brukt som verdiområde.

#### 4.1.2 Analyse av arkitektoniske taktikker

Analysen tar for seg de høyest prioriterte scenarioene fra blant annet Quality Attribute tree 4.1.1 og går dypere inn i hvert scenario. Hovedutbytte fra analysen er

identifisering av arkitektoniske beslutninger som understøtter scenarioet, identifisering av *sensitivitet*, *risiko*, *ikke-risiko* og *tradeoff* til beslutningene, argumentasjon for beslutningene samt konsistens til den valgte arkitekturen.

Beslutningene som understøtter scenarioet klassifiseres i kategoriene sensitivitet, risiko, ikke-risiko og tradeoffs. Dette gir en vurdering av beslutningen i tillegg til en sammenligning mot andre beslutninger og kvaliteter i systemet. Kategoriene beslutningen klassifiseres er, som nevnt tidligere

**Sensitivitet** Rene fordeler som scenarioet gir uten tradeoffs. Annen mulig def: Hvorfor eller hvordan scenarioet er kritisk for systemet.

**Risiko** Hvilke kjente risikoer beslutningen har

**Ikke-risiko** Hvorfor det ikke er risiko ved beslutningen.

**Tradeoffs** Byttehandel mot og påvirkning av andre kvaliteter - både positive og negative

Poengene markeres i en egen liste vedlagt analysen med identifiserende nummering, typisk er S#, R#, NR# og T#. Dette fører til lett gjenbruk av klassifiseringen og forenkler arbeidet med å oppdage like beslutninger for identifisering av særdeles god og dårlige beslutninger.

Figur 12 og 13 viser en foreslått mal for analysen og et eksempel på anvendelse.

## 4.2 CBAM

**Cost-Benefit Analysis Method** begynner der ATAM (4.1) slutter. Motivasjonen er at ATAM ikke tar økonomiske hensyn til arkitekturen og prosjektet.

CBAM er en stegvis prosess bestående av 9 forskjellige steg.

**Innhent scenarioer** Innhent scenarioer fra ATAM eller arkitekturbeskrivelsen. Mulighet for nye scenarioer også. Prioriter scenarioene etter hvordan de tilfredstiller de overordnede business-goals for systemet. Velg den tredelen som blir høyest prioritert.

**Gjør scenarioene bedre** Fokuser på stimuli- og respons-målene til de valgte scenarioene. Definer den verste, nåværende, ønskede og best-case responsen til alle scenarioene.

**Prioriter scenarioene** Interessenter får 100 poeng som skal fordeles på scenarioene. Velg ut den øvre halvpart med mest stemmer. Prioriter disse ved å gi den med mest poeng vekt lik 1.0 og vektlegg andre scenarioer i forhold til denne (strengt tatt ikke nødvendig, men kan ha noen fordeler).

Analysis of Architectural Approach				
<b>Scenario #:</b> <i>Number</i>	<b>Scenario:</b> <i>Text of scenario from utility tree</i>			
<b>Attribute(s)</b>	<i>Quality attribute(s) with which this scenario is concerned</i>			
<b>Environment</b>	<i>Relevant assumptions about the environment in which the system resides, and the relevant conditions when the scenario is carried out</i>			
<b>Stimulus</b>	<i>A precise statement of the quality attribute stimulus (e.g., function invoked, failure, threat, modification . . . ) embodied by the scenario</i>			
<b>Response</b>	<i>A precise statement of the quality attribute response (e.g., response time, measure of difficulty of modification)</i>			
<b>Architectural Decisions</b>	<b>Sensitivity</b>	<b>Tradeoff</b>	<b>Risk</b>	<b>Nonrisk</b>
<i>Architectural decisions relevant to this scenario that affect quality attribute response</i>	<i>Sensitivity Point #</i>	<i>Tradeoff Point #</i>	<i>Risk #</i>	<i>Nonrisk #</i>
...	...	...	...	...
...	...	...	...	...
<b>Reasoning</b>	<i>Qualitative and/or quantitative rationale for why the list of architectural decisions contribute to meeting each quality attribute requirement expressed by the scenario</i>			
<b>Architectural Diagram</b>	<i>Diagram or diagrams of architectural views annotated with architectural information to support the above reasoning, accompanied by explanatory text if desired</i>			

Figur 12: Mal for analyse av arkitektoniske taktikker i ATAM

Analysis of Architectural Approach				
Scenario #: A12	Scenario: Detect and recover from HW failure of a primary CPU			
Attribute(s)	Availability			
Environment	Normal operations			
Stimulus	One of the CPUs fails			
Response	0.999999 availability of the switch			
Architectural Decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPUs	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14
Reasoning	<ul style="list-style-type: none"> <li>Ensures no common mode failure by using different hardware and operating system (see Risk R8)</li> <li>Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst</li> <li>Guaranteed to detect failure with 2 seconds based on rates of heartbeat and watchdog</li> <li>Watchdog is simple and proven reliable</li> <li>Availability requirement might be at risk due to lack of backup data channel (see Risk R9)</li> </ul>			
Architectural Diagram	<pre> graph LR     In(( )) --&gt; P[Primary CPU OS1]     In --&gt; B[Backup CPU w/watchdog OS2]     P -- heartbeat 1 sec. --&gt; B     P --&gt; S[Switch CPU OS1]     B --&gt; S     S --&gt; Out(( )) </pre>			

Figur 13: Eksempel på analyse av arkitektoniske taktikker i ATAM

**Tildel nytteverdi** La stakeholders bestemme nytteverdien (utility) for alle responsmålene. Verste, nåværende, ønskede og best-case til hvert scenario fra forrige steg.

**Lag arkitektoniske strategier til scenarioene og bestem forventet respons** Finn eller lag ny arkitektonisk strategi for de gjenværende scenarioene og bestem hvilken respons de mest sannsynlig vil få.

**Bestem nytteverdien (benefit) til den forventede responsen** Bestem hvilken nytte (utility) scenarioene innehar med de forskjellige responsene. Nytteverdikurven til scenarioene lages ved å plote utility mot respons. Til interpolasjon kan det trekkes rette linjer mellom punktene. Se figur 14 som eksempel.

**Kalkuler fordelene/nytten av en strategi** Ut i fra nytteverdikurven til et scenario finner man forventet utility ved å se hvor forventet respons skjærer den interpolerte kurven. Se figur 14. Total nytte (benefit) for *en strategi* beregnes ved:

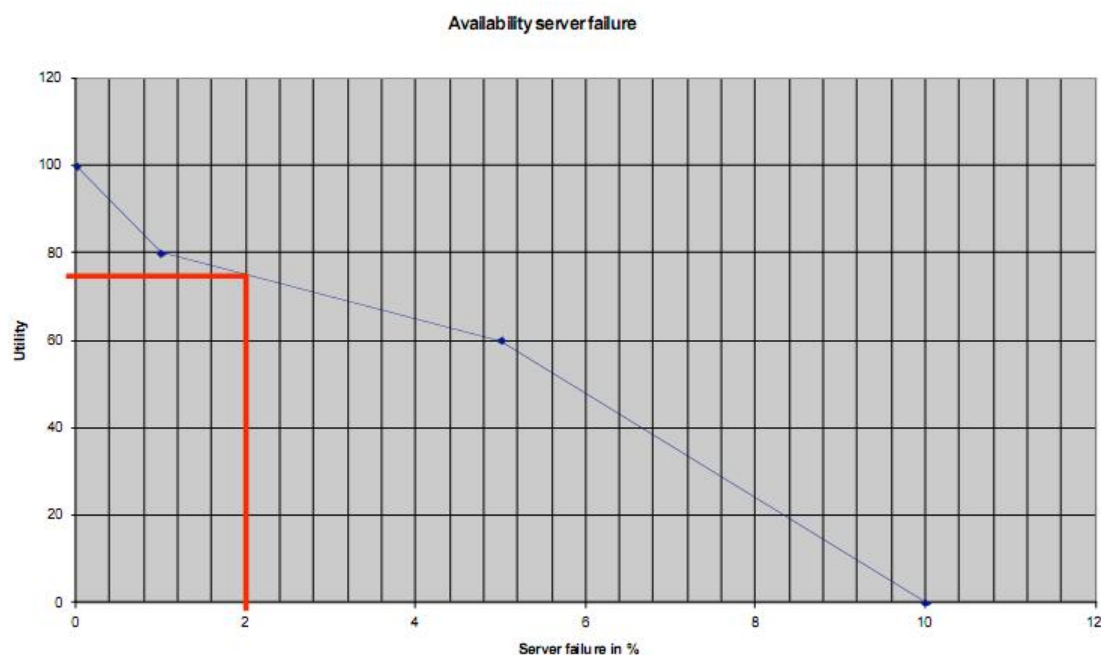
$$\sum^{scenarioN} (expectedRespons - currentRespons) * weight(votes)$$

**Bestem strategi basert på ROI, kostnad og tidsfrist** Bestem kostnaden og implikasjonene til alle strategiene. Regn ut ROI basert på forholdet mellom total nytte (benefit) og kostnaden. Sorter strategier med hensyn på ROI. Velg fra toppen og ned til budsjettet er tomt.

**Gjør en intuitiv refleksjon over resultatet** Kun tall ikke noe fra den virkelige verden - er det realiserbart? Stemmer det med business-goals for systemet? Er interessentene enig?

Se figur 15 for oversikten over CBAM sine steg.

Essensielt er CBAM kun nyttig for de som er interessert i hvor mye dette koster. Metoden kan fremheve viktige kvaliteter og scenarioer hos interessentene, men antakligvis ikke noe mer enn ATAM allerede har gjort (4.1). Tallene som kommer ut i fra CBAM er kun tall satt i forhold til hverandre. Disse bør aldri, aldri stoles blindt på og er kun veiledende. CBAM er kun en metode for å finne de viktigste scenarioene, vekte disse mot hverandre og finne Return On Investment til disse, samt kostnad til utvikling. CBAM eller ikke, dette krever ekstrem erfaring fra analyseteamet og antakligvis vil personer med denne erfaringen kunne gjøre en slik analyse mer effektivt ved bruk av andre metoder. Med det sagt er det nyttig å vite at en slik analysemetode eksisterer. Det kan være en veldig fin trygghetsskaper for en klient og kan antakligvis være med på å sikre kontrakter lettere.



Figur 14: Eksempel på nytteverdikurver (utility curves) brukt i CBAM

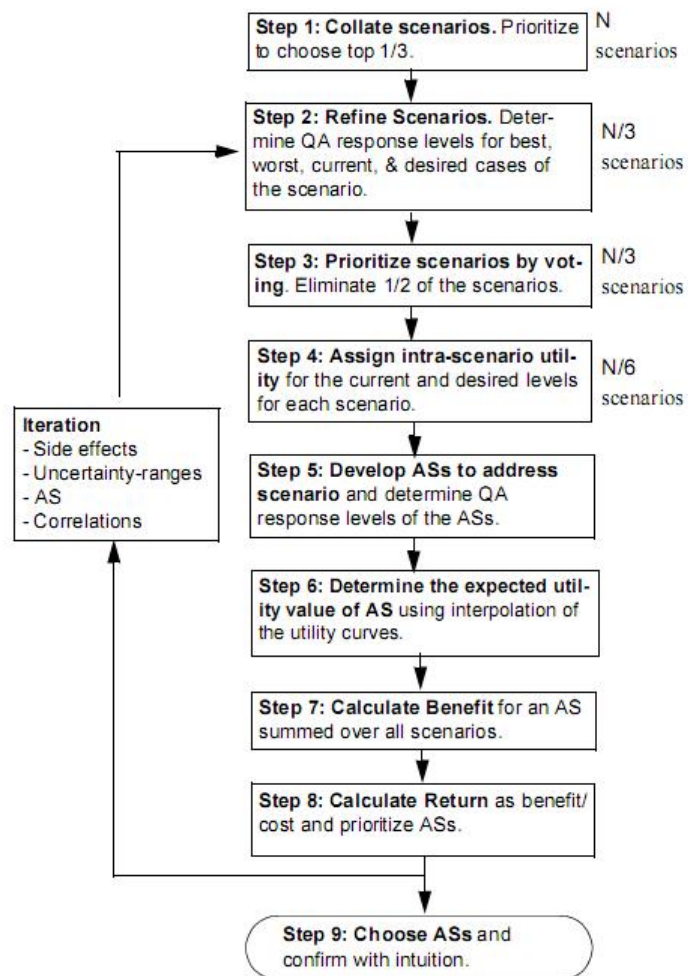
### 4.3 PMA

Post-Mortem Analyse sine primære hensikter er å finne ut hva som skjedde under et prosjekt og hvordan det påvirket prosjektet. Utbytte fra en PMA er erfaring og læring - både positive og negative.

Prosessten deles inn i to faser med forskjellige fokusområder, positivt og negativt. Hver fase består av en brainstorming og en detaljert analyse. Utførelsen og verktøy som brukes kan bestemmes mer eller mindre fritt. Foreslåtte verktøy er KJ-diagram og Root-Cause-Analyse, begge anvender postIt-lapper. Fordelen med dette er at alle deltakerene blir aktive og deltar i analysen - de fleste synspunkt kommer altså fram.

KJ-diagram skapes ved at alle deltakere individuelt skriver ned ideer relatert til fokusområdet på postIt-lapper. Deretter klistres lappene tilfeldig opp på en tavle. Deltakerene leser på hverandres lapper og alle kan flytte de rundt for å gruppere de sammen i temaer. Resultatet er grupper med temaer i fokusområdet. Relasjoner mellom gruppene trekkes for å få fram sammenhenger. Se figur 16 for eksempel.

Root-Cause-Analyse har som hensikt å gå detaljert inn i et av temaene i KJ-diagrammet. Deltakerene må altså velge det (eller de) tema som de vektlegger høyest. Deretter utføres en ny brainstorm i samme stil som for KJ-diagrammet.



Figur 15: CBAM i et nøtteskall

Gruppering av like elementer og relasjoner mellom dem. For en strukturert visning av dette kan fishbone diagrammer brukes, se figur 17.

Fra Root-Cause-Analysen kan det trekkes konklusjoner om hva som var spesielt bra, spesielt dårlig og hvorfor dette skjedde. Ideen er så at prosjektdeltakerene prøver å unngå det som gikk dårlig og omfavne det som gikk bra. PMA forenkliggjør prosessen med å oppdage disse elementene på en lettfattelig og aktiviserende måte.

## 5 Business-aspekter

### 5.1 Components Off The Shelf (COTS)

På større prosjekter legges det ofte begrensninger og styringer for hva som skal brukes under utføringen. For arkitekten er det spesielt interessant å kjenne til hvilke komponenter som finnes fra før eller skal skaffes eksternt som må brukes sammen med eller integrert i systemet. I begrepet komponenter inngår det meste som ikke utvikles under prosjektet. Eksempler på dette er legacy-systems, frameworks, libraries osv.

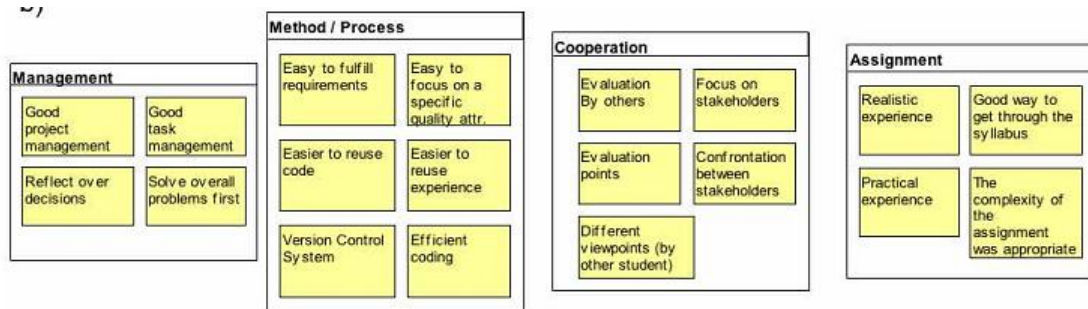
Problemet ved å anvende COTS er at kvalitetene til komponentene som oftest er ukjent og/eller dårlig dokumentert. Siden komponentene blir en del av systemet kan det i verste fall føre til at man ikke har mulighet for å klart definere kvalitetene som systemet totalt sett innehar.

Komponenten har en arkitektur. Denne kan være ukjent av flere grunner som dårlig dokumentasjon, ikke planlagt, mistet osv. Dette kan veldig fort legge tette begrensninger på systemets arkitektur, jf. web-frameworks. Uten god kjennskap til komponentene bør en prøve å minimere bruken av de.

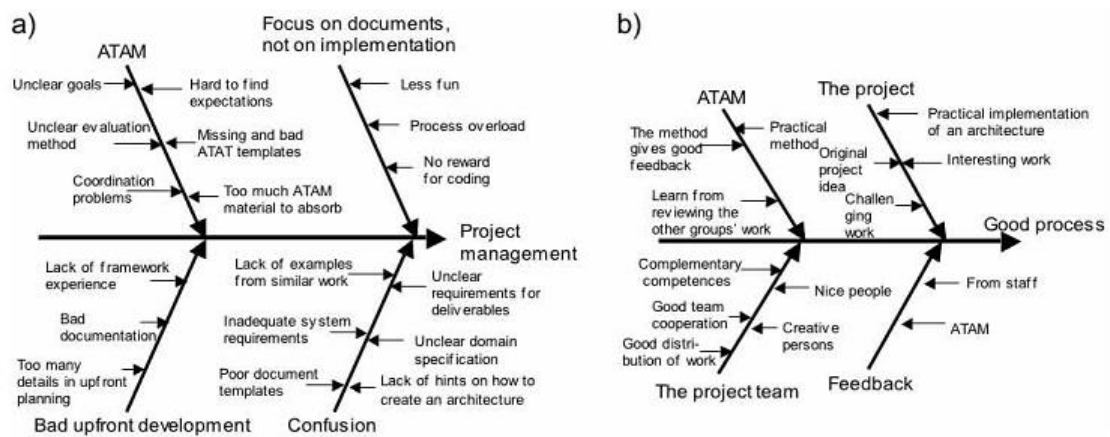
Implementasjonsproblemer kan også veldig fort oppstå ved bruk av COTS. Det hyppigste problemet er antakligvis interface-mismatch. Siden det er et hyppig problem har andre funnet generelle løsninger på det. Det finnes mange design-patterns og standarder for å komme rundt dette problemet, noen av de mest kjente er, wrapper 2.2.7, bridge 2.2.8 og mediator 2.3.2.

### 5.2 Product-line

En product-line er en gruppe med produkter som innehar mange av de samme egenskapene. Arkitektur til en product-line må kunne anvendes på hele (eller størsteparten) av produktene i gruppen. Dette medfører at arkitekturen enten må ha et begrenset omfang og være tilpasningsdyktig innenfor omfanget. Det er derfor essensielt først å definere omfanget deretter utforske mulige variasjoner innenfor omfanget.



Figur 16: KJ-diagram brukt under PMA



Figur 17: Fishbone diagram brukt i Root-Cause-Analyse under PMA

Nøkkelord for COTS er

- Gjenbruk
- Begrenset og definert omfang
- Moden organisasjon
- Klart definerte interfaces
- Generalisering, oppdeling

Som et eksempel på vellykket bruk av product-line kan Nokia mobile nevnes. Mobiltelefonene er delt opp i segmenter hvor telefonene i gruppen har nesten helt lik funksjonalitet og tilsynelatende helt lik arkitektur. Dette gjør at Nokia kan lansere 40-50 *nye* modeller hvert år!

## 6 System integration

System integration består essensielt i å koble sammen mange forskjellige systemer for å danne et nytt system.

Store selskap har som regel en ganske stor portfolio med forskjellige systemer som aktivt brukes i driften. Dette kan fort bli ekstremt uoversiktlig og komplisert for stakeholders av systemene samt stakeholders for portfolioen. Redundans og inkonsistens oppstår dermed også nokså lett og kan i verste fall føre til store økonomiske tap for selskapet. Se figur 18 for en konseptuell skisse over systemintegrasjon.

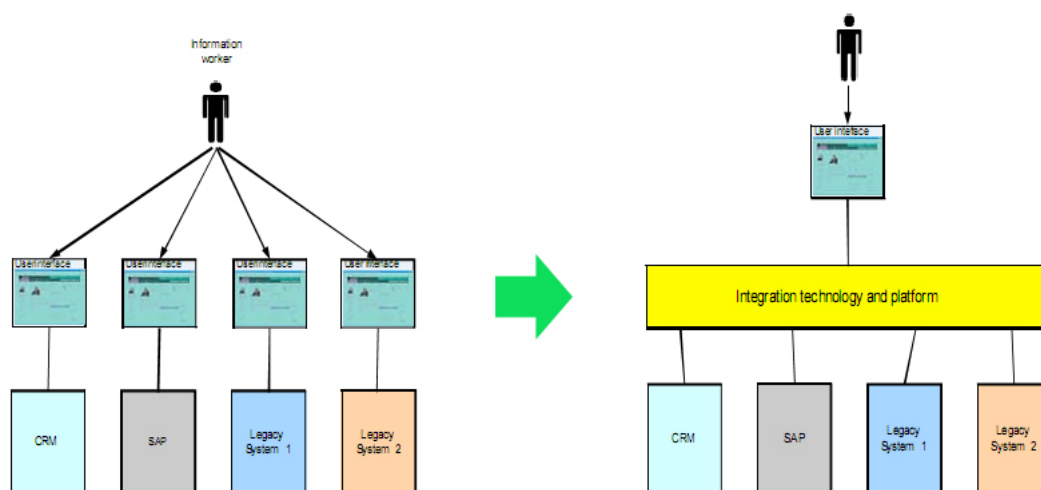
Taktikker for system integrasjon er utallige og blir ikke videre behandlet i denne teksten.

## 7 Rekonstruksjon av arkitektur

Systemer med mangelfull eller ikke-eksisterende dokumentasjon er dessverre ofte å finne. Systemet i seg selv kan fungere brillefint, men straks en vil endre på det bryter det meste sammen. Det er derfor ofte nødvendig med en rekonstruksjon av arkitekturen til systemet for å i det hele tatt ha muligheten til å finne ut om endringer er mulig.

Rekonstruksjon består av tre forskjellige faser.

**Informasjonsinnhenting** Finn ut hvilken informasjon som trengs for å nå et mål (logical view etc.). Innhent all eksisterende dokumentasjon. Analyse av kildekode (hvis tilgjengelig) eventuell analyse av systemet under kjøring.



Figur 18: System integrasjon i selskaper (enterprises)

**Databasekonstruksjon** Lag en database som støtter informasjonen funnet. Bruk kjente DB-modeller og databaser som støtter rekonstruksjon.

**Analyser database** Slå sammen informasjon i databasen ved bruk av spørringer. Gir sammenhengen i informasjonen.

**Rekonstruksjon** Bruk informasjonen og analysene til å rekonstruere arkitekturen.

## Referanser

Fowler, M.: 2004, *UML Distilled Third Edition*, Addison-Wesley.

Ibrahim and Long: 2007, Soa and ea,  
<http://www-128.ibm.com/developerworks/library/ws-soa-enterprise1> .

IEEE1471: 2000, Recommended practice for architectural description of software-intensive systems, *IEEE* .

Kruchten, P.: 1995, Architectural blueprints - the 4+1 view model of software architecture, *IEEE* .

Survey, c. d. p.: 2007,  
[http://www.developer.com/design/article.php/10925\\_1502691](http://www.developer.com/design/article.php/10925_1502691), *developer.com* .

*UML pocket reference*: 2006, Oreilly.

WikipediaAbstractFactory: 2007,  
[http://en.wikipedia.org/wiki/Abstract\\_factory](http://en.wikipedia.org/wiki/Abstract_factory), *Wikipedia* .

WikipediaBridgePattern: 2007,  
[http://en.wikipedia.org/wiki/Bridge\\_pattern](http://en.wikipedia.org/wiki/Bridge_pattern), *Wikipedia* .