NTNU
Det skapende universitet

DiaModlGen

Generering av javakode basert på DiaMODL interaksjonsmodeller

Nils Jørgen Kvam Mittet

Master i informatikk
Oppgaven levert: Mai 2006
Hovedveileder: Hallvard Trætteberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Forord

Arbeidet med oppgaven startet som et samarbeid mellom Anders Braaten og meg. Sammen med veileder, Hallvard Trætteberg, ble det bestemt å gjøre et arbeid som baserte seg på hans arbeid med utviklingen modelleringsspråket DiaMODL. Selv om oppgavene endte opp med å bli meget forskjellige bygger de på samme teoretiske grunnlag. Braatens oppgave er et arbeid med å videreutvikle DiaMODL slik at prototyping støttes bedre. Dette er gjort ved å utvikle ProtoMODL. Mitt ønske var å gjøre et konkret arbeid rettet mot systemutvikling, noe som har resultert i DiaModlGen. Vi hadde begge erfaringer fra arbeidslivet og ville bruke denne erfaringen i valg av problemstilling.

Målgruppe

Målgruppen for denne oppgaven er først og fremst personer som har interesse eller erfaring fra feltene systemutvikling, MMI eller prototyping. Utviklere i arbeidslivet og studenter innen IKT vil også kunne ha nytte av denne oppgaven

Takksigelser

Mange har vært en inspirasjon og støtte under arbeidet mitt med denne oppgaven. Først og fremst vil jeg takke vår veileder Hallvard Trætteberg, for tro på oss i perioder det gikk saktere en ellers. Uten hans arbeid hadde vi heller ikke hatt noe grunnlag for denne oppgaven. Kjæresten min Synnøve, som syntes det var helt greit at jeg satt og jobbet med oppgaven på kveldstid. Arbeidsgiveren min Bemanningshuset/ Ementor, som lot meg få styre arbeidstiden min selv. ”Doggi” for å stille opp med et par kalde flasker ”motivasjon” etter en lang og hard arbeidsdag. Ivar Friheim for nøye gjennomlesing og strukturert kritikk. IDI, Institutt for Datateknikk og Informasjonsvitenskap og til slutt, men ikke minst Anders Braaten for et langt og ukomplisert samarbeid.

Sammendrag

DiaMODL er et hybrid modelleringsspråk, delvis basert på UML Statecharts, og kan dermed brukes som en ren abstrakt notasjon for å modellere interaksjon samtidig som det er mulig å modellere konkrete interaksjonsobjekter. En editor er utviklet for DiaMODL, men den mangler funksjonalitet for å presentere det modellerte brukergrensesnittet på en konkret måte. Denne oppgaven vil presentere en utvidelse til editoren som tilfører konkretisering i form av kodegenerering. Utvidelsen gjør det mulig å sørge for at oppførselen til modellerte interaksjonsobjekter blir realisert i form av kjørbar javakode. På den måten tilføres merverdi til både editoren og mellom roller i en systemutviklingsprosess. Basert på funn i genereringskoden vil det bli utledet regler som ligger til grunn for en generell metode for DiaMODL-kodegenerering. Genereringskoden er implementert i et rammeverk, og har fått navnet DiaModlGen.

Innhold

Figurliste

Kodeliste

Tabelliste

1 Innledning

Utvikling av grafiske brukergrensesnitt til programvare gjøres hver dag. Mange opplever det som en stor og viktig del av utviklingsprosessen. Et brukergrensesnitt har som oppgave å presentere data for bruker samtidig som det skal være mulig for bruker å jobbe med systemet i form av interaksjon. Funksjonen til komponentene i brukergrensesnittet skal presentere systemets spesifikasjon på en komplett måte. For bruker oppleves grensesnittet som selve systemet. Funksjonaliteten som ligger bak er ikke synelig. Det er det heller ikke meningen den skal være. Fra medvirkende i en systemutviklingsprosess er brukergrensesnittet og design av dettet et eget fagfelt, og kun presentasjonsdelen av systemet. Det er ikke bare utseende, form og plasseringen som skal på plass. Hver komponent skal ha en funksjon og fungere sammen med de andre komponentene. En dedikert designer står gjerne for arbeidet med hvordan utseende blir seende ut. Utvikleren tar seg av utviklingen av systemet og logikken. Hvordan komponentene i brukergrensesnittet henger sammen og brukes mot det underliggende systemet er et arbeid som berører begge de to nevnte rollene, men tilhører fagfeltet interaksjonsdesign.

Utviklere kan velge å vrake i integrerte utviklingsverktøy (eng: IDE – Integrated Development Environment) som har støtte både for systemkode og grensesnittdesign. Få av disse er laget for å modellere interaksjonen mellom komponentene i brukergrensesnittet. Selv om man beskriver hvordan et grensesnitt skal se ut, for så å skrive den bakenforliggende koden som får det til å henge sammen, mangler metoden som forteller hvordan de forskjellige komponentene henger sammen. Det er heller ikke implisitt hvordan designkomponentene relaterer til funksjonaliteten i det underliggende systemet. Et eksempel på dette er Swing [W6] og programmeringsspråket Java [W7]. Swing er samling grafiske komponenter brukt til å konstruere brukergrensesnitt i Java. Komponentene har en definert funksjon (liste, tekstfelt, knapp etc). Instansiering av og samhandling mellom komponentene, samt funksjonskall til andre deler av systemet, må beskrives i programkoden. For Java finnes det flere verktøy som gjør dette arbeidet lettere. Flertallet av disse verktøyene har også funksjonalitet for å grafisk sette sammen et grensesnitt bestående av Swingkomponenter. Systemet kan så modelleres, men spørsmålet er om det finnes en god metode som spesifisere hvordan

grensesnittkomponentene henger sammen, og ikke minst, hvordan bruker skal bruke disse til manipulasjon av systemet.

Denne oppgaven tar mål av seg å undersøke om det finnes nok støtte for modellering av interaksjonen mellom komponentene i grensesnittet. Bygd på det vil det presenteres en problemstilling som omhandler hvordan man i et verktøy, i tillegg til å støtte den nevnte integrasjonsprosessen, kan finne en metode som bringer rollene designer, interaksjonsdesigner og utvikler nærmere hverandre.

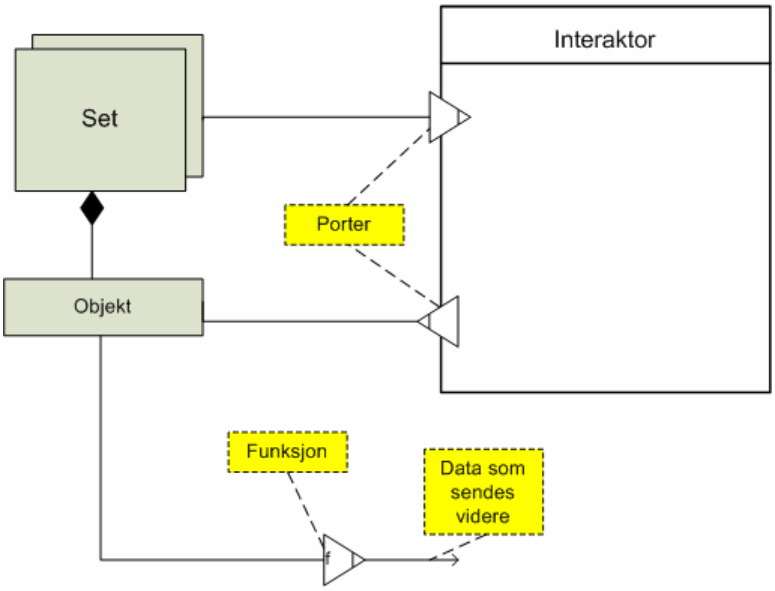
Flere arbeider har adressert problemet med interaksjonsmodellering. UML i seg selv har støtte for interaksjons- og tilstandsdiagrammer som skal vise flyten i og mellom de objekter et grensesnitt består av. Wisdom [Nunes, 2000] er en annen metode som forsøker å adressere deler av denne problemstillingen, men viktigst for denne oppgaven er DiaMODL [Trætteberg1, 2002], som er laget for å være en metode for interaksjonsdesign på grensesnittnivå.

1.1 DiaMODL

Dette avsnittet vil kort introdusere DiaMODL. En mer omfattende gjennomgang kommer i et eget kapittel [4], men en introduksjon er på sin plass da det vil bli referert til DiaMODL flere plasser i dette og kommende kapittel.

DiaMODL er et dialogmodelleringsspråk basert på abstraksjon av interaktorer. En interaktor representerer en eller flere grafiske komponenter og fungerer som grensesnitt mellom bruker og system. DiaMODL beskriver informasjonsflyt og interaksjon i en designrepresentasjon[1]. En dialogmodell beskriver hvordan bruker og system samhandler via et brukergrensesnitt. Dette skjer ved at funksjonaliteten og oppførselen til konkrete interaksjonsobjekter modelleres. Informasjonsoverføringen ivaretas av interaktoren, men bruker flere komponenter. Disse er: "(1) Et sett porter som definerer interaktorens eksterne grensesnitt til systemet, brukeren og andre interaktorer; (2) et sett koblinger som bærer data mellom portene; og (3) en kontrollstruktur som utløser eller forhindrer dataflyt mellom porter på interaktorens inn- og utside og langs koblinger."

[1] De grafiske komponentene som utgjør et brukergrensesnitt.



Figur 1 – Oversikt over DiaMODL-elementer

Figur 1 viser komponentene listet opp over. Interaktoren representerer en grensesnittkomponent. Den venstre siden av interaktoren vender inn mot system, mens høyre side, som er åpen, representerer grensesnittet mot bruker. Interaktoren har to porter på venstre side. Porten øverst, med tuppen inn i interaktoren viser at det kommer data fra systemet inn til interaktoren. Dette er data bruker kan se. Den nederste porten har motsatt funksjon. Data sendes inn til systemet. Dette skjer når bruker manipulerer grensesnittet. Hvilke interaktoren viser frem vises med koblingen fra et sett med objekter. Interaktoren presenterer dette settet i en valgt grafisk komponent, som i dette tilfelle er en liste. Den er ikke vist i interaktoren, men det vil senere bli vist at dette er en mulig funksjonalitet. Den utgående porten har en kobling til et enkelt objekt. Settet og objektet følger UML-notasjon, og viser at det ene objektet er den del av settet send inn til interaktoren. Dette stemmer med at bruker kan velge et element fra en liste. Det som skjer at interaktoren, viser et sett med objekter i en liste. Bruker velger et av elementene, og dette valget sendes tilbake til systemet. Koblingen fra det ene objektet fortsetter til noe som ligner en port, men som er en beregning. En beregning illustrerer at data som passerer gjennom på en eller annen måte blir endret eller behandlet.

Senere i oppgaven blir det presentert mer omfattende eksempler og en mer detaljert forklaring av DiaMODL, men denne oppsummeringen er nok for at et konseptuelt bilde skal kunne dannes.

1.2 Teori, utgangspunkt og mål

Som det vil bli vist er denne oppgaven basert på et forholdsvis snevert teoretisk og teknisk grunnlag. Det vil bli redegjort for aspekter innen moderne systemutvikling og interaksjonsdesign, men i all hovedsak er det DiaMODL som det viktigste teoretiske grunnlaget. Samtidig så var det helt fra starten av meningen å gjøre et arbeid som adresserte flere teoretiske felt. En identifiserbar merverdi for både roller og metoder for systemutvikling var viktig, samtidig som en konkret løsning var ønskelig.

Mesteparten av arbeidet med å spesifisere en problemstilling ble gjort i samarbeid med Anders Braaten [Braaten, 2006]. DiaMODL hadde flere sider som kunne undersøkes og fungere som grunnlag for en problemstilling. Det ble valg to forskjellige rettinger, men både sammen og hver for seg var de ment å kunne oppfylle ønske om en identifiserbar merverdi. Mer presist ville vi se på metoder som kunne forbedre kommunikasjonen mellom designer og utvikler. Vi hadde begge et sterkt fokus på dette fra starten av, da vi kom rett fra relevante jobber i databransjen. Vi hadde begge to erfaringer med prosjekter og utvikling, ofte basert på prototyping og interaksjonsdesign.

1.3 Organisering og arbeid

Dette avsnitter vil redegjøre for hvordan oppgaven er bygd opp, noe om arbeidsmetoder og til slutt hva som er felles for denne oppgaven og Braatens [Braaten, 2006]. Strukturmessig følger oppgaven oppsettet på en ”Vitenskapelig rapport” [Hartvigsen, 1998. s61]. Det største avviket er navngiving av kapiteler, noe som er gjort for å tydeliggjøre den prosessen oppgaven har vært gjennom[2]. For hvert kapittel vil først innholdet bli redegjort for, samt hva som er viktige med kapittelet, dets oppbygging og mål. På slutten av hvert kapittel vil i den grad det passer komme en oppsummering. Dette er spesielt viktig hvis neste kapittel bygger på det foregående.

En figur kan beskrive en problemstilling eller en løsning på en mer presis måten en tekst. Oppgaven baserer seg i stor grad på figurer, noe som også er naturlig da det er den eneste

[2] For informatikk kompliseres standardoppsettet på grunn av at faget har røtter i ulike forskninsdisipliner [19, s.89].

måten å presentere en DiaMODL-modell. Dette har spesielt vært viktig i jobben med å formulere den tekniske teorien og den konseptuelle problemstillingen.
Oppgaven er delt inn i følgende hoveddeler:

Innledning

Oppgavens innledning, dette kapittelet, et todelt. Først vil det innledende arbeidet, hypoteser og oppbygging bli presentert, deretter inneholder kapittelet en innledning til oppgavens problemstilling og teoretiske grunnlag

Teori

Oppgavens problemstilling og tekniske løsning baserer seg ikke direkte på teorien det vil bli redegjort for. Samtidig er det viktig å kunne ha noe å relatere både arbeid og formål til. Teorien redegjør for systemutviklingsparadigmer og metoder som har fungert som inspirasjon. Samtidig vil resultatet av oppgaven vil være et nyttig verktøy nettopp for de presenterte teoriene. Det er også viktig å ha god kjennskap til teorien da det gjør det lettere å identifisere hvilke momenter i et arbeid som lettest lar seg tilbakeføre som en merverdi. Teorien er delt i to. Først blir det redegjort for viktige aspekter ved moderne systemutvikling. Deretter blir det kort redegjort for interaksjonsdesign. DiaMODL hører til her, men er så viktig at det har blitt skilt ut i et eget kapittel.

Noe av teorien er felles for arbeidet med denne oppgaven og Braatens arbeid [Braaten, 2006]. Det må presiseres at det ikke ble jobbet som et team, men at det i enkelte perioder eksisterte et samarbeid. Med det menes at det på enkelte områder ble samarbeidet for å definere og løse problemstillinger, modellere mulige scenarios og lete etter et passende datamateriale. Vi jobbet sammen for å nå våre mål, selv om det til slutt viste seg å være forholdsvis forskjellige mål. Ingenting av teksten er felles, selv om Braaten er sitert. Begge oppgavene har kapittel som redegjør for det samme, men disse er skrevet hver for oss og har forskjellig fokus.

Problemstilling

Problemstillingen begynner hvor innledningen slutter, og presenterer i detalj en problemstiling delt opp i flere delproblem og spesifisert med et sett løsningskrav og avgrensinger.

Arbeid og implementasjon.

For å undersøke om en løsning på problemstillingen er mulig ble det utviklet en konkret implementasjon. Dette arbeidet er presentert over to kapitler. Først presenteres et teknisk grunnlag, samt det rammeverket som ble konstruert. Deretter kommet et kapittel som i detalj viser hvordan det ble implementert en mulig løsning.

Avslutning

Basert på det konkrete arbeidet blir det først drøftet hvordan dette legger grunnlaget for en generell metode. Regler for som en generell metode må implementere ble drøftet. Det samme ble retningslinjer for implementasjon av reglene. Deretter blir arbeidet, metode og valg av problemstilling drøftet og redegjort for i et eget kapittel. Til slutt blir det pekt ut flere mulige veier for et videre arbeid basert på hva denne oppgaven har funnet ut.

1.4 Hypoteser og antagelser

Antagelsene og hypotesene som er tatt med her er de som i etterkant viste seg å ha mest innvirkning oppgavens problemstilling. Det kan synes å være en overlapp, både på rekkefølge og innhold, men poenget er å vise en moding i problemstillingen over tid.

- Modelleringsverktøy er ofte omfattende og har høy terskel for riktig bruk. Et verktøy som er lettere å bruke, med mindre valg og standardisert kode vil ha større gjennomslagskraft.

- Et nytt verktøy blir bedre mottatt, hvis det på en enkel måte tilfører en merverdi som ikke har eksistert før.

- Komplekse brukergrensesnitt gjør det vanskelig å modellere interaksjon, og enda mer vanskelig for utvikler å sette seg inn i. DiaMODL vil kunne være med på å forenkle dette arbeidet.

- Modellering av brukergrensesnitt på interaksjonsnivå vil føre til en bedre dokumentasjon av brukergrensesnittet, muliggjøre iterasjon, raskere prototyping, og bedre systemets avhengighet mellom komponenter. DiaMODL vil kunne utvides til å støtte flere aspekter ved utviklingsprosessen.

- Kodegenerering er en verdifull del av et modelleringsverktøy. En slik funksjonalitet sammen med DiaMODL vil kunne være med på å utvikle det til et mer anvendt verktøy.

- Kodegenerering i sammen med DiaMODL-editoren åpner for at det vil bli utviklet tileggsfunksjonalitet som vil støtte andre arbeidsoppgaver enn det som opprinnelig var tenkt.

- DiaMODL vil med kodegenereringsfunksjonalitet senke terskelen for å prøve ut forskjellige utgaver av et brukergrensesnitt.

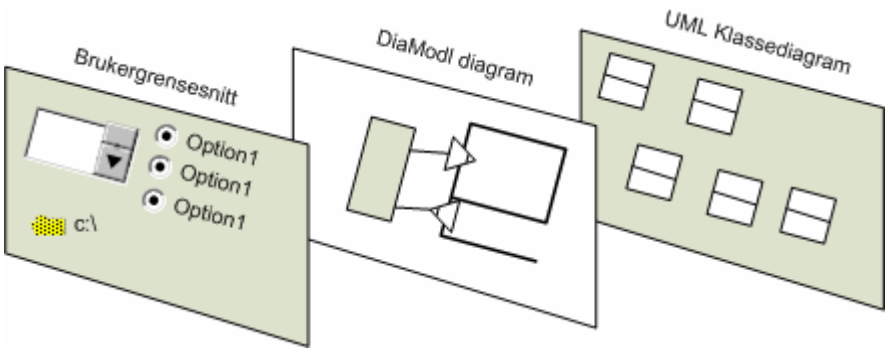
Flere av hypotesene stammer fra innledende arbeid sammen med Braaten [Braaten, 2006], og beskriver våre observasjoner av DiaMODL. Samtidig viser de utviklingen fra et generelt fokus på verktøy for systemutvikling til spesiell funksjonalitet i en DiaMODL-editor.

1.5 Grunnlag for oppgavens problemstilling

I [Trætteberg, 2003, s13] drøfter Trætteberg et mulig videre arbeid om verktøystøtte for DiaMODL. Dette punktet var avgjørende for valg av problemstilling for denne oppgaven. En problemstilling baser på eller annen form for verktøystøtte ville innebære noe konkret, noe det var sterkt ønske om. På en annen side, så ville implementasjon av en verktøystøtte blitt litt for snevert og lite innovativ sett fra en forskers synspunkt. DiaMODL berører som nevnt flere faglige områder, så en problemstilling som kunne berører flere av disse var en prioritet. I [Trætteberg1, s.187] drøftes en integrasjon av DiaMODL og UML. En slik integrasjon, kombinert med verktøystøtte dannet det grunnlaget som ville medført både et teknisk arbeid og muligheten for å utrede en teoretisk metode. Dette var den første grunnleggende problemstillingen. Ut fra den ble det utledet to mulige applikasjoner for kombinering av DiaMODL og UML.

Alternativ 1:

Utgangspunktet var tre logiske lag. Ikke applikasjonslag, men lag med forskjellig konkretisering av et brukergrensesnitt. Det første laget representerer brukergrensesnittet med tilhørende komponenter. Det siste, og ”bakerste” laget representere den konkrete implementasjonen i form av en UML-modell. Figur 2 viser hvordan DiaMODL som interaksjonsmodellen ligger som et lag mellom disse og fungerer som et bindeledd.

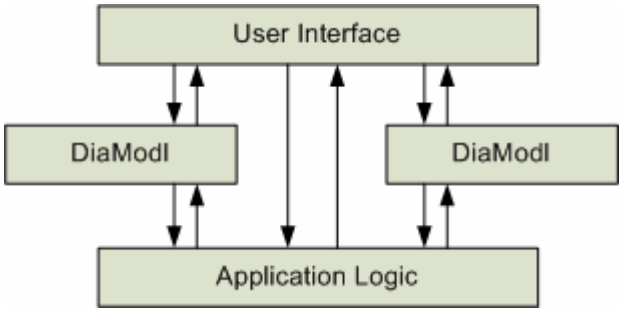


Figur 2 – Tre mulige lag i et brukergrensesnitt

Alternativ 2:

I denne applikasjonen blir modellering av konkret brukergrensesnitt og analysemodell (UML) utført på hver sin side i applikasjonen som er delt i to. Det grafiske grensesnittet på venstre side og analysemodellen på høyre side. Interaksjonsmodellen (DiaMODL) tegnes på et gjennomsiktig "matpapir" som legges over og binder de to arealene sammen. Man skulle kunne ta utgangspunkt i faktiske skjermbilder, eller benytte seg av abstrakte prototyper. Attributter i grensesnittet som bredde, høyde, tegnbredde og skrifttype er knyttet direkte til komponentene. Typisk vil det være flere sett med foiler (matpapir) i dette laget, ett for hvert skjermbilde.

En konkret applikasjon for DiaMODL har blitt utviklet i form av en DiaMODL-editor. Via en objektmodell som realisere DiaMODL-modellene er det også implementert et sanntids kjøretidsmaskineri. Faktiske brukergrensesnitt- (GUI) elementer kan brukes sammen med DiaMODL-elementer og vise med faktisk oppførsel. Maskineriet er designet så større eller mindre biter av brukergrensesnittet kan bli drevet av modellobjekter uten å påvirke andre deler. På den måten er det mulig å gradvis introdusere nye modellobjekter inn i en eksisterende applikasjon og prøve ut nye og mindre deler av et nytt design. Figur 3 viser DiaMODL tenkt som et lim mellom brukergrensesnitt og underliggende systemer. DiaMODL-elementer vil omfatte delere av brukergrensesnittet i form av verktøy for interaksjonsmodellering og samtidig være bindeleddet mot et systemet. Dette er konsistent med de to foregående forslagene til en applikasjon.



Figur 3 – Utviklingsarkitektur med DiaMODL [Trætteberg, 2003]

1.5.1 Uviklingsprosessen i dimensjoner

Design av brukergrensesnitt er en kreativ og omfattende prosess som ofte involverer flere deltagere. Designere kommer med forslag til brukergrensesnittet og løsningene må vurderes med hensyn på både funksjonalitet og brukbarhetskrav. Prosessens mål er å lage et brukergrensesnitt med en viss brukbarhet samtidig som det tilfredsstiller det målet man satte seg for selve systemet. Det er naturlig at flere typer designrepresentasjoner[3] brukes i denne prosessen. Man skal overføre designkunnskap til designere, måle fremgangen i prosjektet, og ikke minst arbeide seg frem mot et konkret mål.

Hvem er brukeren?

En utvikler må ha en forståelse av hvem brukeren er, hvilket miljø brukeren opererer i, målene brukeren vil nå og hvilke oppgaver brukeren trenger å utføre. For å være i stand til å forstå og bevare denne forståelsen av et spesifikt designproblem må designer kunne dokumentere dette i et passende format som representerer denne kunnskapen. Som nevnt over spiller en rekke formelle spesifikasjoner en viktig rolle i denne prosessen. Det samme gjelder også for den modellbaserte vinklingen på brukergrensesnittdesign [Trætteberg1, s.27: oversatt].

Designer og utvikler

Designer og utvikler er to roller som ikke nødvendigvis opererer sammen eller innenfor samme domene. Designers rolle og mål er å skape et brukergrensesnitt som tilfredsstiller brukers krav, mens utviklers rolle er å bruke designerens ervervede kunnskap sammen med sin egen kunnskap for deretter å sette sammen et komplett system. De forskjellige

3 Abstrakte tegninger, Prototyper og forskjellige Uml-diagram

rollene opererer i forskjellige dimensjoner, men de kan samarbeide over et felles sett verktøy.

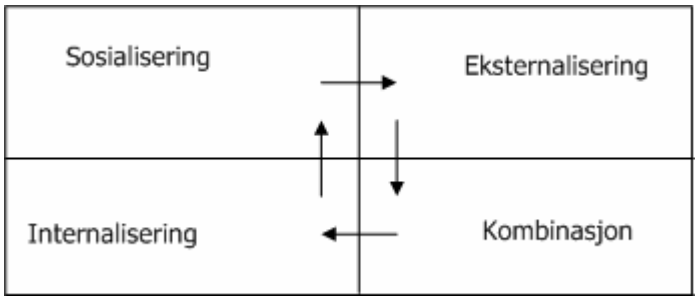
Problem og løsningsorientering

Forskjellen på problem- og løsningsorientering er ikke alltid like klar. Noen modeller kan bli tolket som både som en løsning på et spesifikt problem, og en spesifikasjon på et problem som blir løst senere i utviklingsprosessen. Et eksempel på dette er en abstrakt dialogmodell. Den kan bli regnet som en løsning på et problem spesifisert ut fra et sett med oppgaver, og på samme tid representere krav til en senere konkret løsning [Trætteberg1, s.28: oversatt].

1.6 Formalitet og dimensjoner

Forskjellen på taus og eksplisitt kunnskap er tett koblet til formelle og uformelle representasjoner. Eksplisitt kunnskap kan overføres til andre hvis disse har kunnskap nok til å forstå den. Denne forståelsen kan være naturlig språk eller annen halvformell notasjon [Nonaka, 1998 i Trætteberg1, s45]. I en utviklingsprosess er ofte kunnskap om systemet lettere å gjøre eksplisitt. Flere verktøy finnes for dette, hvor UML er det mest vanlige. For design av brukergrensesnitt, som er en mer kreativ prosess, ender ofte kunnskapen om hvordan man designer som taus kunnskap. Brukergrensesnittet er i seg selv vanskelig å dokumentere, men den kreative prosessen i seg selv utgjør kanskje et enda større problem.

Videre beskriver [Trætteberg1, s36] at jo mer formell en representasjon er, jo mer objektiv, kontekstavhengig og eksplisitt er kunnskapen. Kunnskapen er ekstern i forhold til opprinnelsen, uavhengig av kontekst, og kan bli lagret og overført til andre. Som en motsetning er ikke håndfast, eller taus, kunnskap personlig, kontekst spesifikk eller subjektiv. Den kan ikke uten videre overføres videre. Nonaka beskriver at denne tause kunnskapen kan gjøres eksplisitt gjennom en eksternaliseringsprosess, mens eksplisitt kunnskap gjøres taus, gjennom internalisering [Chandler, 1998]. Sistnevnte er ofte en nødvendighet for å kunne bruke eksplisitt kunnskap på en kreativ måte i stedet for mekanisk etter en oppskrift. I følge denne teorien innebærer enhver problemløsningsprosess kreasjon av kunnskap, og det er en konstant utveksling fra taus til eksplisitt kunnskap og omvendt. Figur 4 viser Nonakas figur for kunnskapsflyt.



Figur 4 – Nonakas kunnskapsflyt [Chandler, 1998, Trætteberg1, 2002]

Basert på dette presenterer Trætteberg i [Trætteberg1, s45] tre dimensjoner og to rettinger med til sammen få 6 bevegelser. I praksis er det bevegelser begge veier langs de tre aksene dimensjonene representerer. Bevegelsene beskrives slik:

Bevegelser i Perspektivdimensjonen (eng: Problem/solution)

Bevegelsen går fra problem som det ene ytterpunktet, til løsning som det andre ytterpunktet. Problem til venstre og løsning i midten, og som jeg skal vise, eksisterer det et abstraksjonsnivå mellom de to.

1. En løsning på et spesifikt problem blir løst ved å bruke en aller annen form for designkunnskap. Dette er den samme flyten som hvis man leser en side med tekst. Først definerer man et problem, deretter løses det innefor gitte rammer og med et gitt sett av verktøy. Man ser fremover, mot en konstruktiv rolle. Siden denne bevegelsen representerer skifte av fokus, kan man forventet begrenset støtte fra et modelleringsspråk.

2. Går man den andre veien, fra løsning til problem, blir det en analytisk aktivitet. Eksempel på dette kan være at man rekonstruerer en modell fra en ferdig løsning, gjerne for å analysere eller dokumentere.

Detaljeringsdimensjonen (eng: Granularity)

Denne bevegelsen går fra en grov oversikt til en fin detaljering, eller motsatt.

3. Detaljer blir utledet fra en spesifikasjon på et høyere nivå. Man går fra lite detaljer til detaljer. Dette representerer et arbeid med å konkretisere mer og mer fra en abstrakt spesifikasjon. Det blir et konstruktivt arbeid.

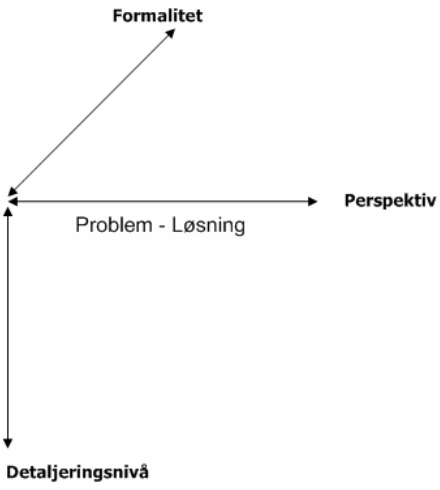
4. Når man går den andre veien, fra detalj til oversikt, så er det en generalisering.
Man gjør gjerne dette for å kunne sammenligne med andre ting på et høyere nivå. På lik linje med bevegelse 2, så blir dette en analytisk aktivitet.

Formalitetsdimensjonen (eng: Formality)

Dette er en bevegelse som viser om et problem eller løsning er mer eller mindre formell.

5. Fra mindre til mer er den "normale" bevegelsen, på den måten av vage ideer og kunnskap blir gjort om til mer presise beskrivelser. Siden en felles forståelse bland deltagere er viktig, kan man tillegge denne bevegelsen en *komunikativ* rolle.

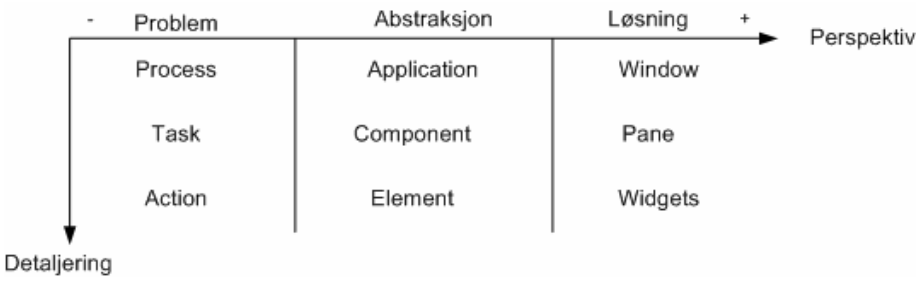
6. Motsatt vei går bevegelsen fra en formell representasjon til en mindre formell. Dette kan oppfattes som en unaturlig bevegelse men kan være nyttig i en kontekst hvor deltagerne ikke er inneforstått med en formell notasjon. Man går da til en mindre formell designnotasjon for å sørge for at alle får en felles forståelse.



Figur 5 – Designkunnskap i 3 Dimensjoner [Trætteberg, 2001]

Bevegelsene kan settes sammen og presenteres i figur 5. Figur 6 konkretiserer figur 5 ved å legge til eksampler på hva en bevegelse langs aksene betyr. Figuren viser ikke den formelle aksen da denne ikke har noe å si for valg av representasjoner.

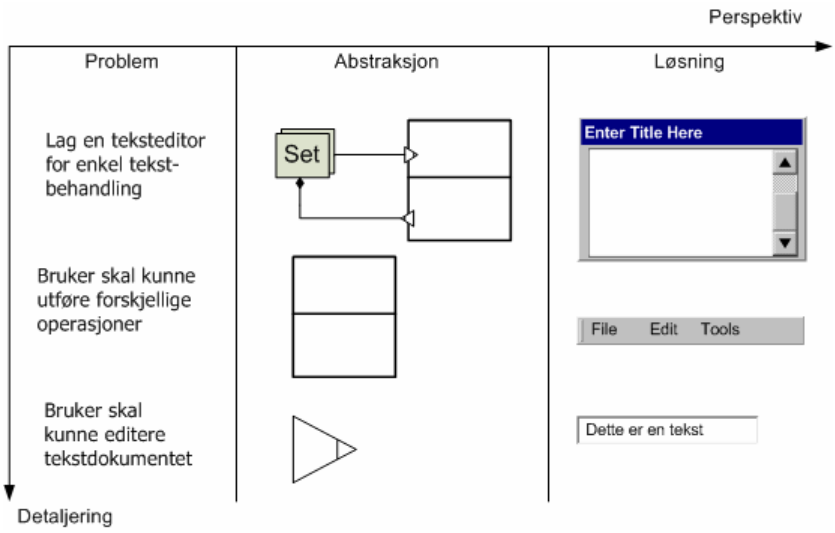
De tre vanlige brukergrensesnittmodellene (1) oppgave / domene, (2) abstrakt og (3) konkrete interaksjon kan plasseres langs perspektivaksen [Trætteberg1, s.36].



Figur 6 – Klassifisering av designkunnskap.

Figur 7 viser en enda mer konkret utgave av figur 5. Lang kolonnen *problem* ser man hvordan et spesifikt problem blir mer og mer detaljert desto lengre ned på detaljeringsaksen man kommer. Helt til høyre finner man det mest konkrete perspektivet; de faktiske komponentene man bruker til å løse problemet. Minst detaljert viser man at man skal ha et program, deretter en litt mer detaljert komponent, og helt til slutt en komponent som nøyaktig beskriver hva bruker skal kunne gjøre for å løse det minst detaljerte problemet. I dette tilfelle er det å lage en teksteditor for å redigere tekst.

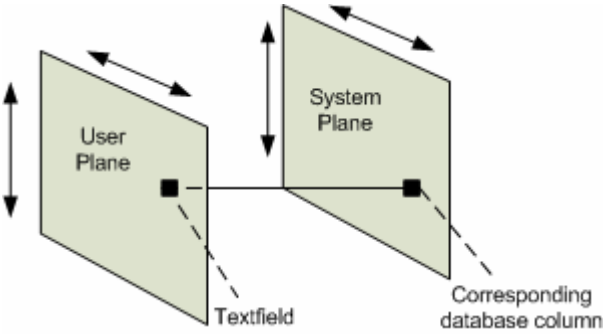
Kolonnen i midten er den mest spennende. Det er kolonnen for abstraksjon og inneholder DiaMODL-elementer i mer og mer detaljert grad. Oppsummert får man at man spesifiserer et problem ved å sette ord på det, deretter modelleres den abstrakte interaksjonen. Med det mener vi hva bruker *skal* kunne gjøre. Helt konkret ender dette opp med hvilke komponenter som velges for å implementere spesifikasjonen gitt ved abstrakt modellering.



Figur 7 – DiaMODL-elemetner i klassifisering av kunnskap

1.7 En 4. dimensjon

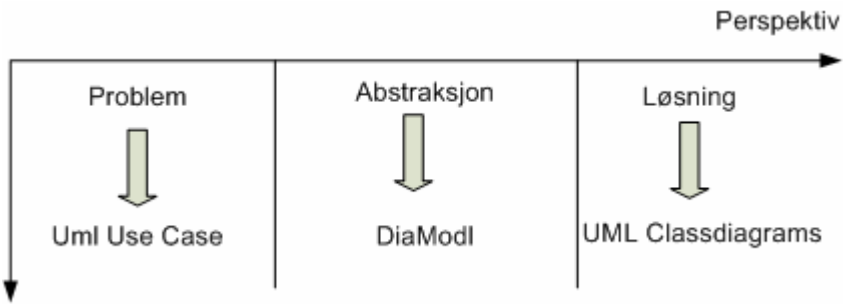
Trætteberg beskriver i [Trætteberg1, s.32] at en fjerde dimensjon synes å kunne integreres i presentasjonsrammeverket. Hovedmotivasjonen for å gjøre det er å kunne koordinere spesifiseringen og utviklingen av brukergrensesnittet med et underliggende system. Et skille mellom brukergrensesnitt og system gir både en praktisk og en konseptuel mening, men det avhenger også av applikasjonen. Hovedgrunnene for å gjøre det er (1) for å isolere bruker fra detaljer de ikke har interesse av og (2) man kan dele utviklingen inn i flere deler. Dette muliggjør parallell jobbing på flere deler av systemet. I det siste tilfelle vil en fjerde dimensjon være til hjelp da man kan bruke den til å koordinere utviklingen, redegjøre for forandringer og sikre konsistens. Et krav for at dette skal fungere er at brukerdimensjonen og systemdimensjonen er innrettet etter hverandre slik at man kan identifisere representasjoner som omhandler det samme. Figur 8 viser en tenkt modell med den fjerde dimensjonen – systemdimensjonen.



Figur 8 – En fjerde dimensjon [Trætteberg1, 2002]

1.7.1 UML i brukerplanet

UML har blitt, og er, standard for analyse og design av systemer, men som nevnt så er støtten for interaksjonsdesign noe mangelfull. Hvis man ser på de forskjellige perspektivene i brukerplanet; problem, abstraksjon og løsning så kan deler av disse representeres med UML. Figur 9 viser at et problem, eller problemdomene kan uttrykkes ved bruk av UML bruksmønsterdiagram (eng: usecase) Det vil senere bli vist at det finnes utvidelser av dette som kanskje er bedre egnet, men i første omgang er det brukermønsterdiagram som er det aktuelle. Et brukermønsterdiagram støtter variabel detaljering. Man kan ha forskjellige skjema som viser forskjellige detaljering av problemet. Disse skjemaene følger graderingene på detaljeringsaksen.



Figur 9 – DiaMODL i perspektivaksen

I den andre enden av perspektivaksen, løsningen, så viser figuren at det også her kan brukes UML i form av klassediagram. En konkret grensesnittklasse er som alle andre klasser og trenger ikke beskrives på annen måte.

Problemet oppstår derimot i midten. UML støtter dårlig abstraksjon av brukergrensesnitt. Abstraksjon er ment å representere interaksjonsdesignet. Med det menes oppførsel og sammenheng mellom komponenter i grensesnittet og brukers interaksjon med disse. Det

er her DiaMODL kommer inn da det er laget med dette for øyet. Selv om Trætteberg i [Trætteberg1, s.190] - "begrensninger og videre arbeid" – redegjør for et arbeid om mulig integrasjon mellom UML og DiaMODL er det ingen klar sammenheng mellom disse i figuren selv om DiaMODL delvis er basert på UML tilstandsdiagrammer. DiaMODL burde kanskje i fremtiden bli implementert som en del av UML. På en annen side er det kanskje viktigere å bygge opp en relevant praksis for bruken av DiaMODL, for deretter å utvide dets domene. UML vil være i endring og en mulig tilnærming mot hverandre fra begge parter vil kanskje være den mest ideelle fremgangsmåten. En mulighet er å se på hvordan DiaMODL og UML kan utfylle hverandre i en prosess hvor man opererer med to forskjellige plan og hvordan man kan bruke begge teknologiene til en parallell utvikling av forskjellige deler av et systemet.

1.8 Arbeidet med oppgaven

Forskning starter med et spørsmål [Hartvigsen, 1998] Man lurer på noe i forbindelse med noen man selv eller andre har gjort, og velger å forske på dette for å se om det er mulig. Men når en problemstilling er valgt dukker det opp flere nye valg. Hvordan skal man bruke problemstillingen til å definere hva man skal løse? Hvilke kriterier skal legges til grunn for at man har fått en tilfredsstillende løsning på problemstillingen? Skal man jobbe så bredt som mulig, eller skal man plukke ut deler av problemstillingen og jobbere dypere med dem?

Tidligere i dette kapittelet ble det presentert en rekke hypoteser og antagelser. Når det skulle utarbeides en problemstilling måtte det bli sett nærmere på hvilke behov det var viktigst å tilfredsstille. Antagelsene danner grunnlaget for et konkret problem, som både er ment å omfatte en teknisk løsning og en jakt på merverdi. De påfølgende kapitlene presenterer det teoretiske grunnlaget som gjorde det mulig å gjøre et arbeid som omfatter det å se på DiaMODL som er verktøy i en utviklingsprosess. Den presenterte teorien har vært den viktigste inspirasjonskilden, samt fungert som verktøy for å finne en løsning på den valgte problemstillingen.

2 Moderne Systemutvikling

Arbeidet med denne oppgaven hører til i domenene systemutvikling og interaksjonsdesign. Dette kapittelet vil redegjøre for noen aspekter som inngår disse domenene. Først og fremst er det de aspekter som er relevant for oppgaven. Med det menes innflytelse på enten problemstilling eller løsning.

2.1 Roller i utviklingsprosessen

Utviklingsprosesser brukes både for å forhindre at prosjekter blir utviklet ”ad hoc” og påse at suksess ikke avhenger av noen få dedikerte nøkkelpersoner, men av hele organisasjonen som helhet. I en utviklingsprosess kan man bruke ordet ”roller” i flere sammenhenger. Først er det rollen den faktiske prosessen har. Dette kan være forskjellig avhengig av hvilket stadium i utviklingsprosessen man befinner seg på. Den andre rollen, og den man gjerne tenker på når det er snakk om forskjellige roller er forskjellige personers roller, gjerne oppdelt etter fagområde.

Utviklingsprosessen kan deles inn i fire primære roller [Campos, 2004]:
(1) Sørge for å veilede i rekkefølgen til en utviklingsgruppes aktiviteter,
(2) spesifisere hvilke artefakter som skal utvikles og når de skal utvikles,
(3) dirigere oppgavene til individuelle utviklere og til gruppen som helhet og
(4) tilby kriterier for overvåking og måling av prosjektets produkter og aktiviteter.

Roller personer kan ha i en utviklingssyklus styres av hvert enkelt prosjekts parametere. Størrelsen kan ha innvirkning, likedan dets struktur og mål. I [Bambara, 2001] defineres og beskrives rollene prosjektleder, systemarkitekt, utvikler, designer, tester og deployer. Hvis man er på et lite prosjekt så kan en person inneha flere av disse rollene, men det er enkelte roller mange i praksis opererer med at ikke kan innehas av en og samme person. For eksempel bør ikke utvikler ha kundekontakt. Designer kan og bør i mange tilfeller ha kontakt med kunde, men det finnes negative sider ved dette. Designeren kan med kunden ved siden av seg føle press for å legge til mer enn det som var planlagt, noe som igjen får konsekvenser for utvikler. Prosjektleder, og systemarkitekt er de rollene som er mest nærliggende for kundekontakt, men dette avhenger av hvilket systemutviklingsparadigme man jobber under. Roller som tester og deployer kan ofte slås sammen, men tester og

utvikler bør være avskilte roller. Med mindre det er snakk om store prosjekter er ofte prosjektleder og systemarkitekt også en og samme person. Gjerne går dette på erfaring. Enten lang fartstid eller spesiell erfaring med teknologien bruker i det aktuelle prosjektet.

Oppgavens fokus på roller dreier seg i stor grad om rollene designer og utvikler. Grunnen til det er erfaring med at det ofte eksisterer en konflikt mellom disse rollene. Den jobben designer, og interaksjonsdesigner, gjør vil være bestemmende for deler av arbeidet til utvikler. Det sammen kan også være tilfelle hvis systemets funksjonalitet er bestemmende for utseende. En viktig antagelse er at det kan være mye å hente på å bringe designer og utvikler nærmere hverandre teknologisk.

2.2 Prototyping

Med prototyping menes her det man ofte kaller for hurtig prototyping (eng: rapid prototyping). Skillet mellom prototyping og hurtig prototyping går på hvor ofte man forkaster det arbeidet man har gjort og gjør det på nytt. I prototyping lager man gjerne en testapplikasjon en gang, og baserer løsningen på den. I hurtig prototyping gjør man kortere og oftere iterasjonen. Prototypene som lages er ment for å teste mindre og mer spesifikke problemstillinger.

Prototyping brukes til å konkretisere ideer på et tidlig stadium i utviklingsprosessen. Det er en fordel om så mange av prosjektets deltagere som mulig er med på deler av prosessen. Prototyping trenger ikke å være en frittstående metode, men er ofte en viktig del av andre systemutviklingsparadigmer.

Braaten beskriver i [Braaten, 2006] at en prototyp er et verktøy som brukes for å besvare spørsmål rundt spesifikasjon og design. Et vesentlig poeng i denne sammenhengen, er at hvilke midler, materialer eller verktøy som benyttes for å lage prototypene, ikke er viktig. Det essensielle med prototyper er hvordan de benyttes av interessentene for å utforske og demonstrere enkelte aspekter ved det fremtidige systemet. Prototyping kan gjøres på mange nivåer av formalisme og detaljnivå [1.7]. og fra en praktisk en praktisk side så kan arbeidet gjøres omfattende og avansert ved hjelp av verktøy, eller så enkelt som med penn og papir. I [Toxboe, 2005] beskrives det at tegning er viktig i de tidlige fasene av prototyping da det er en enkel måte å støtte den kreative prosessen til designeren. Man

kan lettere utforske flere muligheter uten å henge seg opp i detaljer. Dette setter fokus på det som er viktig på dette stadiet; den overordnede strukturen og interaksjonsflyten.

[Floyd, 1984, i Toxboe, 2005] skiller mellom tre hovedklasser av prototyping. (1) Prototyping for å utforske, (2) prototyping for å eksperimentere og (3) evolusjonær prototyping. Karakteristisk for det siste punktet er at det bryter ned tradisjonelt lineære utviklingstrinn og prioriterer læring, tilbakemelding og en kravspesifikasjon i stadig endring. Det argumenteres også her for at prototyping er et verktøy i systemutviklingen som gjør seg best når det kombineres med andre metoder.

En av metodene som bruker aspekter fra prototyping mye, og delvis baserer seg på samme arbeidsmetode er XP – Ekstrem programmering [Beck, 1999]. Xp sitt viktigste mål er å redusere risikoen ved utviklingsprosjekter. Toxboe mener i [Toxboe, 2005, s.10] at prototyping er en passende måte å gjøre dette på da man kan teste deler av prosjektet før man utvikler det. XP baserer seg også på det de kaller testkode (eng: spike solutions). Man lager kode for å teste om det faktisk fungerer før man koder på nytt i prosjektet. På den måten blir man ikke forstyrret av andre aspekter, og man vet at den aktuelle problemstillingen er løst. Denne måten å arbeide på skiller lite fra prototyping, men har kanskje tatt det enda lenger og gjør det i en mindre og mer detaljert skala.

Modeller kan forveksles med prototyper. I [Braaten, 2006] forklares det at den store forskjellen mellom modeller og prototyper som representasjonsartefakter for interaktive systemer, ligger i graden av abstraksjonen som artefaktet viser. Prototyper er av natur en mer konkret representasjon enn en modell, og gir derfor mer rom for direkte interaksjon med en bruker. Fordelene til en modell er at den kan gi en bedre forståelse for de underliggende mekanismer som definerer det interaktive systemet gjennom abstraksjon.

I [Gutierrez, 1989] beskriver Gutierrez detaljert forskjellige teknikker for prototyping i forskjellige problemkontekster. Det deles inn fire forskjellige situasjoner: S, P, E og U.

- En S-type, spesifikasjons (eng: spesification) situasjon er når det finnes en komplett og autoritær situasjon, hvor prototypen er ment å validere at denne spesifikasjonen blir fulgt.

- En P-type, praktisk (eng: practice) situasjon. I denne situasjonen er det ikke nok at prototypen tilfredsstiller spesifikasjonen. Grunnen til dette er at man rett og slett ikke kan teste spesifikasjonen på grunn av stadige endringer i den. Man må basere seg på en iterativ prosess som følger utviklingen i spesifikasjonen.

- En E-type, utviklingssituasjon (eng: evolution) er karakterisert av behovet for å kunne predikere effekten systemet vil ha på sine omgivelser. Prediksjon er vel og merke ikke noen sikker viten, og designet må utvikle seg gradvis. Utviklere må lage teknikker for å simulere systemet. Verktøy for å støtte frekvente endringer er også viktig.

- En U-type, usikkerhetssituasjon (eng: uncertainity) kjennetegnes ved en manglende modell av en applikasjon. En av grunnene til dette kan være at det er for mange ukjente faktorer til å klare å abstrahere problemdomenet.

På bakgrunn av disse problemkontekstene foreslås disse typene prototyping:

Utforskende prototyping

Denne typen prototyping fokuserer på basisproblemet kommunikasjon mellom spesialister og sluttbruker. Målet er å bruke prototyping til å utveksle ideer og enes om design.

Systemsimulering

Systemsimulering representerer en systemskisse som brukes til å verifisere et enda ikke eksisterende systems karakteristikk. I all hovedsak for å gjøre det lettere å kommunisere meninger og ideer.

Scenariobasert design

Kan ses på som en mer detaljert form for prototyping. Man simulerer hendelser bruker kan komme ut for med det ferdige systemet.

Eksperimentell prototyping

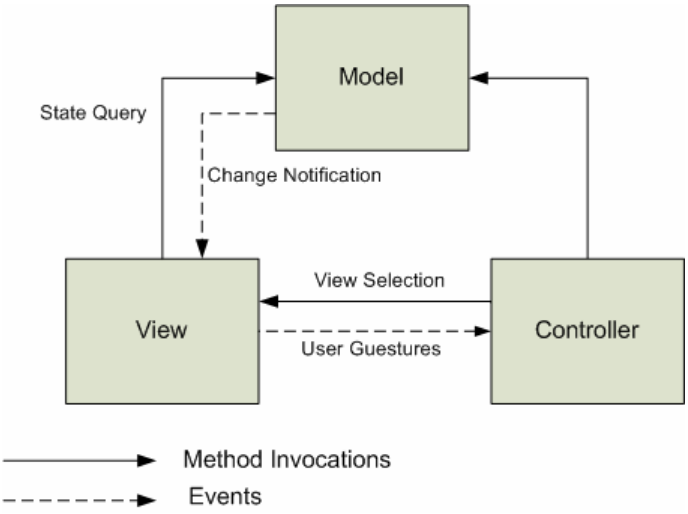
Brukes til å teste hypoteser og mulig problemer. Snarveier tas for å rask kunne produsere en modell som virker. Effektivitet og kompletthet blir ofret til fordel for testmuligheter

Oppsummert så er prototyper anvendbart på flere områder, både innen brukergrensesnittdesign og systemutvikling. Det er en større eller mindre del av mange utviklingsparadigmer og kan tilpasser hvert enkelt paradigme så vel som hvert enkelt prosjekt. DiaMODL er allerede et verktøy for blant annet prototyping. En utvidelse av funksjonaliteten til DiaMODL vil kunne brukes i flere av de presenterte scenarioene, i tillegg til å være et konkret utviklingsverktøy.

2.3 Model-View-Controller

Model-View-Controller (MVC) er en konseptuel modell som ble utviklet på Xerox PARC på 70-tallet, og brukt første gang i Smalltalk-80 [Coyle, 1994, s2]. MCV ble utviklet som et rammeverk for å kunne bygge omfattende applikasjoner med komplekse brukergrensesnitt. Kjernen i rammeverket er at man kan koble komponentene i brukergrensesnittet fra komponentene i den underliggende domenemodellen. Dette fører til en modulær struktur som det er lett å vedlikeholde, videreutvikle og endre [W1]. Selve applikasjonsmodellen kan kobles til flere forskjellige brukergrensesnitt uten at den trenger å bli endret. Et brukergrensesnitt kan også gjenbrukes mot flere datakilder, uten at koden som viser frem data trenger å tilpasses.

For å effektiv kunne bruke MVC, må de forskjellige komponentene og begrepene forklares. I MVC er manipulasjon fra bruker, applikasjonsmodellen og visuell tilbakemelding eksplisitt separert. Hver del blir kontrollert av tre spesialiserte objekter [Burbeck, 1992: oversatt]:



Figur 10 – Mode- View-Controller

Figur 10 viser de tre hovedkomponentene i MVC, som er:

Model

Modell-elementet i MVC (norsk: modell) representerer de underliggende data eller foretningsprosesser i applikasjonen. Det kan være alt fra kompliserte beregninger, eller enkle kall til en database for å hente lagrede data. Modell er et begrep som i forskjellige kontekst kan ha helt forskjellig mening. I forrige kapittel ble modell brukt om en fornekling og presentasjon av et system. I MVC er modell en representasjon av data eller underliggende system. DiaMODL bruker ordet modell om en helhetlig DiaMODL-modell, samtidig som en modell kan være et modell-element. Dette vil bli presentert senere. Det vil også bli vist at oppgaven tolker begrepet modell likt modellen i MVC.

View

Viewet (norsk: visning) viser frem innholdet som kommer fra modellen til bruker. View'et mottar data fra modellen, og inneholder selv bare informasjon om hvordan de aktuelle data skal presenteres. Tradisjonelt er det også viewet sitt ansvar å vise endringer som måte skje i modellen. Dette gjelder applikasjoner hvor modellen kan endre seg uten interaksjon fra bruker. Dette kan skje ved at view registrer seg til modellen og ber om å bli informert når en endring har skjedd. En annen løsning er at view'et selv ber om å få oppdaterte data når det måtte trenge det.

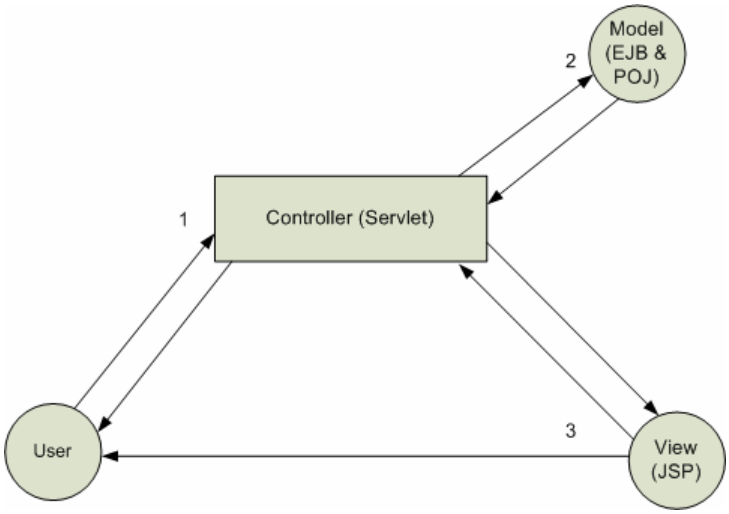
Controller

Controlleren (norsk: kontroller) oversetter kall som kommer fra bruker via viewet til aksjoner som skal utføres av modellen. Den ruter forespørsler til modellen med nok informasjon til at denne henter riktige data og utfører de handlinger den skal. I en frittstående GUI-applikasjon består brukerinteraksjon av manipulasjon av elementene i brukergrensesnittet. I en Web-applikasjon kommer gjerne interaksjonen som GET eller POST forespørsler til webserveren. Når jobben er utført av den forespurte modellen, ruter kontrolleren data tilbake til riktig view.

Fra applikasjoner installert lokalt på en maskin, samt applikasjoner med en klient – tjenere arkitektur, har fokuset i dag dreit mot applikasjoner med nettlesere som grensesnitt mot brukeren. Dette har ført til en endring i lagdelingen og arkitekturen i større system. Objektorientering har funnet sin plass også i webbaserte systemer. Utfordringen har vært at forskjellige logiske komponenter har måtte blitt delt mellom flere system. Modellen må deles opp mellom klient, samt en eller flere servere [Leff, 2001]. Brukergrensesnittet blir bygd opp på en webserver og sendt til bruker, mens applikasjonslogikken gjerne ligger på en annen dedikert enhet. Data hentes fra dedikerte databasetjenere, eller kommer via andre tjenester. Objektorientering relaterer mest til applikasjonslogikken, men for å binde sammen brukergrensesnitt og logikk har det blitt laget flere standarder, og det er i denne objektorienterte applikasjonsstrukturen MVC har fått en renessanse.

Struts er et utviklingsrammeverk helt og holdent basert på MVC. Det blir administrert av Apache Software Foundation (ASF) [Carnell, 2003]. ASF var og er en mer eller mindre løs forbindelse mellom personer dedikert til åpen kildekode[4]. Struts ble utviklet for å hurtig kunne utvikle brukergrensesnitt til webapplikasjoner skrevet i Java. Figur 11 viser hvordan MVC blir implementert i Struts. Numrene representerer sekvensen på kallene.

[4] Åpen Kildekode (eng: Open source), er en modell for spredning av programvare med den faktiske kildekoden vedlagt så bruker kan se og endre koden. Flere lisensmodeller for bruk av koden finnes.



Figur 11 – MVC i Struts

1. Når en bruker sender en forespørsel til en MCV-basert applikasjon blir forespørselen fanget opp av kontrolleren, som er implementert som en Servlet[5]. Kontrolleren undersøker forespørselen og ruter den videre til riktig komponent i modellen.

2. Modellen er det man ofte omtaler som foretningslogikken eller data i en applikasjon. Når den aktuelle komponenten i foretningslaget mottar forespørselen fra kontrolleren, blir komponentenes oppgave utført, Data og kontroll blir returnert fra modellen til kontrolleren.

3. Når kontrolleren mottar data fra modellen vil det basert på tilhørende metadata bli bestemt hvordan disse dataene skal vises for brukeren. Et view skal kun bestå av kode som presenterer data. Ingen behandling eller endringer på data skal gjøres. I Struts består et view av en JSP-side[6]. JSP er Javakode som generer html. Når koden er ferdig generert, blir den sendt til brukers klient, som er en nettleser. Kontrollen blir så gitt tilbake til kontrolleren som på nytt venter på input fra bruker.

[5] Java klasse som kjører i en applikasjonsserver. Behandler forespørsler via GET eller POST. Java Server Pages, en del av Sun's J2EE spesifikasjon.
[6] Java Server Pages. Også en del av Sun's J2EE spesifikasjon.

2.4 XP – Extreme Programming

Extreme Programming (XP, norsk : Ekstrem Programmering) er en utviklingsmetodologi og et regime for hvordan et program skal utvikles [W3]. XP er basert på observasjoner i utviklingsmiljøene om hva som gjorde utvikling raskere, eller i motsatt tilfelle hva som var forsinkende. XP er først og fremst basert på en eksaminering av etablerte metoder for programvareutvikling. Et sekundært mål er å senke kostnader på utvikling.
I [Beck, 1999] beskrives XP som en utforskende og smidig utviklingsmetode som søker å tilfredsstille en kunde gjennom tidlig involvering og kontinuerlige leveranser av programdeler. På 60 og 70-tallet, i programmeringens barndom var det opp til hver enkelt å programmere sine programmer som de ville. Da mange programmer ofte ble for komplekse for andre til å forstå, ofte også for programmereren selv, ble det på 80-tallet foreslått å underlegge programmering et sett med regler som skulle gjøre utviklingen mer standardisert. Etter hvert som man så dette fungerte til en viss grad ble det utviklet flere og flere regler i den tro at hadde man bare nok, så skulle man kunne utvikle et program bare ved å bruke reglene på en gitt problemstilling. Etter hvert har meningen om dette dreid dit hen at et for kompleks sett med regler gjør det vanskeligere å utvikle. Kompleksiteten førte til at utviklere tok snarveier. Så fort man gjorde noen utenom retningslinjene var man tilbake der man startet. Programmer ble langt i fra var feilfrie, vanskelig for andre å sette seg inn i samt feilsøke i. Det ble utviklet verktøy for å støtte programmeringen, men de ble ofte for komplekse i seg selv. XP er en reaksjon på dette og har som hensikt å snu opp ned på dette paradigmet, riktignok i konkurranse med andre metoder.

Historien viser at en høy prosentandel av utviklingsprosjekter feiler. Starter man på et nytt og ukjent problem, så er dette forbundet med en risiko. Jo mer ukjent, jo større risiko. Et av formålene med XP var å adressere denne risikoen for å oppnå en større suksessrate.

Hva er Extreme Programming?

Aktivitetene i XP er ikke sekvensielle. Det vil si at har mye felles med modellene *Iterativ utvikling* og *Spiralmodellen*. Forskjellen ligger i at XP er ment å være mer dynamisk med hensyn på hvor ofte man gjentar de fire basisaktivitetene. Slik kan XP tilpasses hvert enkelt prosjekt og utvikler. XP baserer seg på små grupper med utviklere. 2-12 er et

antall ofte brukt, men det er gjennomført prosjekter med opp til 30 utviklere. Man baserer seg på ordinære utviklere. Kodeeksperter gjør ikke nødvendigvis jobben mer effektiv. Det er prosessen som gjør XP effektiv. På prosjekter med høy risiko er et så lavt antall utviklere som mulig å foretrekke. Det er også viktig å legge vekt på at XP ikke bare omhandler utviklere, men hele prosessteamet, med prosjektledere og kunder. Alle jobber sammen samtidig og mot samme mål. Å kunne spørre spørsmål, forhandle og komme med ideer i en flat struktur er nødvendig for at XP skal bli en suksess. En annen viktig faktor i XP er testing. Testene skal teste spesifikk funksjonalitet, og skrives før koden som skal testes.

Hvordan bruker man Extreme Programming?

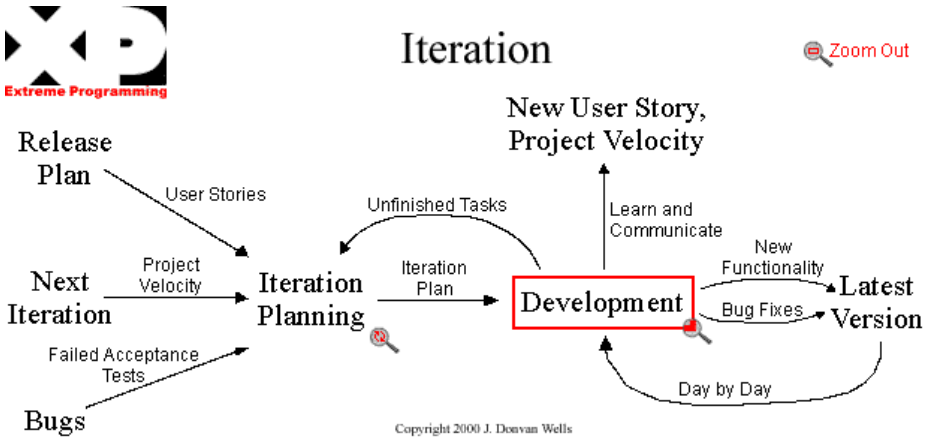
XP's prinsipper er delt inn i fire områder:

- ***Planlegging***
    - *Brukerhistorier*
      Brukerhistorier har samme formål som UML-bruksmønsterdiagram [W8], og brukes til estimat av publiseringsplanlegging. Brukerhistoriene erstatter tradisjonelle kravspesifikasjoner.

    - *Publiseringsplanlegging av små og frekvente publiseringer*
      Møte for å fastsette publiseringsplan. Overordnet for prosjektet. Utgangspunkt i brukerhistorier. Høy frekvens av kodepublisering gjør det mulig å isolere små frittstående biter med funksjonell kode.

    - *Måling av arbeid(eng: project velocity)*
      Et mål på hvor mye arbeid som er gjort i prosjektet. For å sammenligne med estimater.

    - *Iterasjonsplanlegging*
      For hver iterasjon holdes det et møte. Hver iterasjon er 1-3 uker lang. Kartlegging av kritiske områder. Den viktigste funksjonaliteten først. Figur 12 viser flyten i iterasjonsplanleggingen.



Figur 12 – XP Iterasjonsplanlegging

- *Flytting av personer*

  Personer flyttes mellom roller i prosjektet for å unngå tap av kunnskap og flaskehalser i koding.

- *Morgenmøter*

  Så kort og konsist om dagens gjøremål, problemer og løsinger som mulig. Viktig at hele utviklingsteamet er med.

- *Forandring av regler som ikke fungerer*

  *H*vis metodologien i XP ikke passer til det arbeidet som gjøres, så ikke nøl med å omrokkere, fjerne eller legge til elementer. Dette er noe av kjernen i XP

- ***Design***

  - *Så enkelt design som mulig*

    Et enkelt design tar bestandig kortere tid å fullføre enn et komplekst et. Gjør alltid det enkleste som fungerer.

  - *Systemmetaforer*

    Velg en systemmetafor som hele temaet kan forholde seg til. En som gjør

at de involverte holder samme fokus.

- *CRC-kort (kort for eng: Class, Responsibilities and Collaboration)*
  beskriver en klasses oppgave, metoder og avhengigheter. Øker fokus på Objektorientert utvikling.

- *Testløsninger*
  Lag testløsninger for å finne løsninger på problemer. Gjør at man kan ignorerer andre ting en det faktiske problemet. Beholdes ikke etter en løsning er funnet men implementeres på nytt i prosjektet.

- *Innføring av funksjonalitet*
  Ikke legg til ny funksjonalitet før det faktisk er nødvendig. Lite av det vil bli brukt til slutt hvis man innfører det på et for tidlig tidspunkt.

- *Ombygging av systemet (eng: refactoring)*
  Fjern redundans og sørg for å ha et så enkelt og strukturert system som mulig til enhver tid. Fører til bedre kode og bedre kodedesign.

- ***Koding***
  - *Kundens tilgjengelighet*
    Kunden skal være tilgjengelig så ofte som mulig, og titte utvikler over skuldrene hvis mulig. Han skal være en del av utviklingsteamet. Senker risiko for unødig kode og funksjonalitet.

  - *Kodestandard*
    Teamet skal være enige om kodeformatering. Fører til konsistens i koden, og gjør det enklere å flytte folk mellom oppgaver.

  - *Enhetstesting*
    Man vet hva man skal lager, og tester for å teste dette lages før noe kode

skrives. Deretter lager man en kode som feiler for å teste testen. Deretter koder man til testen går gjennom. Enhetstesting gjøres på så lavt nivå som mulig. Å kode testen først hjelper til å holde fokus på hva som faktisk skal løses.

- *Parprogrammering*
  Dette er kanskje den mest kjente delen av, og den folk flest forbinder med XP. En utvikler skriver koden, mens den andre ser på, kommer med meninger, ser feil, og har bedre overblikk. Man veksler på hvem som koder.

- *Integrering. Ofte, men kun et par av programmerere av gangen.*
  Integrer kode som er skrevet med resten av prosjektet med korte mellomrom. Kun et par gjør dette av gangen. Deretter testes hele prosjektet. Sikker at feil som fører til mye redundant arbeid ikke oppstår.

- *Eierskap til kode*
  Alle på teamet skal kunne komme med ideer til alle deler av prosjektet. Alle skal kunne endre og rette på andres kode. Med ette hindrer man flaskehalser og lite effektiv kode.

- *Optimering til slutt*
  funksjonalitet er viktigere en effektiv kode. Ikke gjør optimering før prosjektet er ferdig og funksjonaliteten er på plass.

- *Ikke jobb overtid*
  Overtid øker ikke alltid effektiviteten. Legg heller på flere personer hvis nødvendig.

- ***Testing***.
    - *All kode skal enhetstestes*
      All kode enhetstester under og etter utvikling

    - *All kode skal passere enhetstester før den gis ut.*
      Ingen kode integreres med resten av systemet før den har passert en enhetstest.

    - *Lag tester ved oppdagelse av feil (eng: bugs)*
      Hvis en feil oppdages, ikke prøv å rett på den uten å ha en test som kan bekrefte at den er rettet.

    - *Akseptansetester*
      Dette er tester som baserer seg på brukerhistoriene. Sikrer at kunden får det han skal ha.

Hvordan relaterer XP til DiaMODL? For å oppsummere XP, så ble metodologien laget som et svar på problemdomener hvor krav skriftet ofte. Kunden har sjelden eller aldri et helt klart bilde på hva han vil ha. Her kommer DiaMODL inn. En modellering kombinert med faktiske grensesnittkomponenter kan hjelpe bruker til et klarere bilde av systemet. Man kan også ha et systemet hvor det er forventet at funksjonaliteten vil måtte endre seg med forholdsvis korte mellomrom. I noen tilfeller er systemene i en konstant endring. Det er i slike tilfeller hvor tidligere metoder feiler, XP er ment å fungere [Beck, 1999: oversatt].

2.5 Brukersentert design og utvikling

I [W18] beskrives brukersentert design både som en filosofi og en prosess. Som filosofi plasseres mennesket, i motsetning en ”ting”, i sentrum. Prosessen fokuserer på kognitive faktorer (som oppfattelse, minne og problemløsning), som trår i kraft når mennesket omgås og bruker ”ting”. DiaMODL kan i en utviklingsprosess setter bruker i sentrum, samtidig som DiaMODL-editoren [6.2] er verktøyet eller ”tingen”.

Brukersentert design (og utvikling) eksisterer i mange fasonger. I [Toxboe, 2005] redegjøres det for dette. Fra å være en integrert del av en utviklingsprosess har brukersentert design utviklet seg til å også omfatte egne verktøy for dette. Dette er verktøy som ikke omfatter implementasjonsplaner. De er utelukkende laget for å innpasse bruker i designprosessen. Som denne oppgaven vil vise, kan DiaMODL, med tilhørende editor, brukes som et slikt verktøy.
Brukersentert design plasser personen, ikke systemet, i midten. Dette gjøres ved å besvare spørsmål om mulige brukere, deres mål og oppgaver. Svarene kan brukes til å rette systemet inn mot spesifikke brukergrupper. Det er ikke gitt at det trenger å være programvareutvikling. Alle konkrete ting en bruker kan interaktere med kan være underlagt en brukersentret prosess [Toxboe, 2005: oversatt].

I [2.1] ble det redegjort for roller i en utviklingsprosess. Hovedmålet for alle involvert i en utviklingsprosess er at systemet de utvikler er brukervennlig. I [Gulliksen, 2001] påpekes det at dette er spesielt viktig for designerrollen. Det arbeidet designeren har er den, som for bruker, har mest innvirkning på systemets brukervennlighet. Videre defineres brukbarhet på samme måte som i ISO 9241 (Ergonomi i programvare brukt i et kontormiljø med visuelle terminaler), del 11 : Guide til brukbarhet. ”Brukbarhet er i hvor stor grad en spesifikk bruker med sin programvare kan oppnå gitte mål for effektivitet og tilfredshets innen en spesifikk kontekst”. Kort fortal, så er det et mål på hvor effektiv brukeren blir med systemet og hvor fornøyd han eller henne er. For å oppnå de gitte målene er en brukersentert prosess en nødvendighet.

I [Beck, 1999] redegjøres det for et sett forskjellige metoder for å oppnå brukbarhet. Det faktiske designet av et grensesnitt er, i praksis, en fase i utviklingsprosessen som naturlig er ustrukturert. I mange tilfeller får også denne fasen for lite oppmerksomhet. En av grunnene til det er at det kan være vanskelig å separere kreativt designarbeid fra faktisk programmering. DiaMODL passer inn i designfasen og bidrar til å isolere den aktiviteten, samtidig som det blir lettere å inkludere bruker i prosessen. [Beck, 1999] foreslår forskjellige metoder som bidrar til større fokus på grensesnittdesignet. Ikke alle relaterer direkte til brukersentert utvikling i form at bruker faktisk deltar på designprosessen.

Håndtverksmetoden

Denne metoden ser på hvert prosjekt som unikt. Programvare blir utviklet under overvåkning av en dyktig grensesnittekspert. Tilhenger av denne metoden mener ofte at strukturering av grensesnittdesign ikke er mulig. Talent blir viktigere av metode. Metoden kan fortsatt inkludere bruker på en bra måte, men i forhold til DiaMODL, som er en konkret en metode er den lite funksjonell.

En forbedret systemutviklingsprosess

Denne metoden prøver å introdusere MMI-teknikker inn i reportuaret til en vanlig systemutviklingsprosess ved hjelp av oppgaveanalyser. Bruker kan trekkes inn grensesnittsimuleringer brukt som et analysegrunnlag. DiaMODL, og spesielt DiaMODL-editoren passer på mange måter inn i denne metoden som et verktøy

Kognitiv konstruksjon

Metoden prøver å oppnå optimalt design ved å innføre teorier fra kognitiv psykologi. Metoden er mer teoretisk enn de andre metodene, og personer tilhørende en tradisjonell utviklingsprosess vil ha vanskelig for å inkludere metoden i sitt arbeid.

Teknologisk vinkling

Metoden er annerledes enn de andre metodene, da den har systemutviklingsprosessen i seg selv som utgangspunkt. Verktøystøtte er et viktig moment, spesielt verktøy for prototyping. DiaMODL med tilhørende editor vil passe godt inn i denne metoden, spesielt hvis funksjonaliteten utvides.

En konkretisering som passer inn i alle metodene over, presenteres i [W18] som retningslinjer for brukbarhet. For å oppnå dette er det ikke nok å følge reglene. Involvering av bruker for å verifisere hvert trinn er viktig:

Synlighet

Synlighet er når brukeren kan danne seg en korrekt kognitiv modell av “tingen”. Dette oppnås ved at manipulasjonsmetoden er tydelig og entydige.

Gjenkjennelse

"Tingens" brukergrensesnitt, i dette tilfelle programvare, må være gjenkjennelig. Mer bestemt så må kontroller (knapper og andre GUI-elementer) har samme funksjonalitet selv om de brukes flere plasser i applikasjonen.

Tilbakemelding

Når brukeren gjør noe, skal programvare gi en tilbake så bruker ser at han har gjort noe. Dette kan være riktig eller galt, men det må systemet gi tydelig beskjed om.

Navigasjon

Det skal være enkelt å navigere. Spesielt gjelder dette websider, men gjelder også vanlig programvare.

Feil

Feilmeldinger må være forståelige for bruker. Det må bli fortalt hvorfor feilen oppsto og hvilke valg som finnes.

Tilfredshet

Brukeren skal bli tilfreds med det han ser. Med det menes at det skal være minst mulig forstyrrende elementer.

Språk

Om grensesnittet inkluderer språk, skal det legges vekt på at teksten er lett forståelig og ikke gir rom for feiltolkinger

Regler for hvordan punktene brukes er essensen i brukersentert utvikling. Punktene kan også kjennes igjen fra andre systemutviklingsparadigmer. Spesielt er brukermedvirkning fokusert på i XP [2.4]. Reglene for bruk er presentert under:

1. Involver brukere fra starten. Retningslinjer er bra, men ikke nok til å utvikle full brukbarhet for en spesifikk applikasjon.
2. Kjenn brukerne. Finn ut hvor mye erfaring de har, i hvor stor grad de vet hva de vil ha, og hvor kjent de er med sitt domene. Og ikke minst hvor vant de er med bruk av datamaskiner.

3. Analyser oppgaver og mål. Observer brukeren og prøv på bakgrunn av observasjonen å besvare spørsmål som går på hvordan brukeren bruker prototypen (prototyping av brukergrensessnitt er antatt), hvilken informasjon brukeren trenger og hvordan denne reagerer når han står fast.
4. Ikke fastsett et endelig design for fort. Jo mer funksjonalitet som innføres, jo mer vil det ha å si for brukergrensesnittet. Dette leder naturlig over i neste punkt:
5. Test brukbarhet kontinuerlig. Bruker skal være med hele veien. XP predikerer at om nødvendig skal brukeren sitte og se på utviklingen foregår.

Brukersentert utvikling og design kan brukes på flere måter og vektlegges forskjellige i forskjellige faser av et prosjekt eller fra prosjekt til prosjekt.. Oppgaven vil presentere et teknisk arbeid som ikke direkte trenger involvering av bruker. Derimot vil resultatet av arbeidet kunne fungere som et viktig hjelpemiddel i en brukersentert prosess.

3 Interaksjonsmodellering

Interaksjonsmodellering er fasen hvor man syr sammen delene av et system består av for se hvordan disse til sammen utgjør nyttig oppførsel [W19]. Mer konkret er det arbeidet med å bestemme hvordan interaksjon med og i et system skal foregå. Braaten beskriver i [Braaten, 2006] at det innen feltet interaksjonsdesign har vært brukt flere typer modeller for å beskrive hvordan interaksjonen foregår mellom bruker og system via et brukergrensesnitt. Det er tre mye brukte angrepsmåter: Kognitiv modellering, arkitekturmodellering og interaksjonsmodellering. Innen interaksjonsmodellering fokuseres det på fire forskjellige modeller: (1) scenariomodellering, (2) tilstandsmodellering, (3) transisjonsmodellering, og (4) dialogmodellering.

Modellbasert brukergrensesnittdesign har vært en forsket på det siste tiåret uten at det har ført til noen særlig industriell suksess. I følge [22 i Braaten, 2006] fokuserer ikke modellbasert verktøy nok på formalismen som trengs for å automatisk generere brukergrensesnitt. En ny generasjon brukersentrerte modelleringsverktøy må dukke opp for å styrke markedsakseptansen til modellbasert design. Den formaliserte tilnærmingen hindrer verktøy i å gi tilstrekkelig støtte til tanke- og designoppgavene som utviklere må utføre for å lage brukbare og effektive brukergrensesnitt. I forbindelse med interaksjonsdesign er det spesielt interessant å lage modeller som definerer oppførselen til brukeren og det aktuelle systemet; I tillegg til selve samspillet dem imellom. DiaMODL er et slikt verktøy, og vil bli presentert i et eget kapittel [4]. Dette kapittelet omhandler to metoder som ikke direkte støtter interaksjonsmodellering, nemlig UML og Wisdom. Grunnen til at UML blir presentert er at DiaMODL baserer seg på UML, og tilbyr samtidig støtte for interaksjonsdesign. Samtidig er UML brukbart for spesifikasjon og dokumentasjon i et hvert utviklingsprosjekt. Wisdom prøver å adressere mye av det som er nevnt i dette avsnittet, og var lenge utgangspunktet for arbeidet med oppgavens problemstilling. Som det vil bli vist ble Wisdom en inspirasjonskilde, og men også gjenstand for mangler og kritikk.

3.1 UML

Etter at objektorientert programmering slo gjennom har man lykkes i å samles om et felles sett beskrivelsesteknikker for å designe og dokumentere en objektorientert

prosessen. Denne samlingen beskrivelsesteknikker er kjent som UML. UML står for "Unified Modeling Language" og er en notasjon, dvs. en samling med symboler og diagrammer som kan brukes til å lage en visuell modell av et system. Det som er nytt med UML i forhold til tidligere slike notasjoner er U'en, nemlig unified. Det betyr at man har blitt enige om en standard. Tidligere har det vært flere standarder og man kunne oppleve at ulike miljøer bruker to ulike symboler for samme ting eller at samme symbol hadde ulik mening. Med andre ord en unik mulighet for unødvendige misforståelser. I oktober 1995 ble versjon 0.8 av UML lansert. I tillegg til notasjonen var også en utviklingsmetode inkludert. Senere så man at det kunne være nyttig å definere UML som en standard for notasjon uten at dette ble knyttet til en utviklingsmetode, og f.o.m. UML 1.0 omfattet standarden kun notasjonen. Den seneste versjonen, UML 1.1 ble presentert for The Object Managenement Group (OMG) [W5] i september 97 og akseptert som standard måneden etter.

UML er en meget omfattende og komplekst spesifikasjon. Få bruker hele spesifikasjonen fullt og helt. Deler av UML er mer populær enn andre og sett fra utviklers synspunkt også mer brukbare. Dette avsnittet vil gjøre rede for UML som helhet, men spesielt de delene av UML som vil bli brukt eller drøftet.

3.1.1 Bruksmønsterdiagram

Et bruksmønster (eng: UseCase) definerer en eller flere oppgaver som systemet skal kunne håndtere. Hvert bruksmønster skal definere en egen og avsluttende oppgave for systemet. Det skal også beskrives hvilke aktører som skal benytte seg av hvert bruksmønster. Systemets samling av bruksmønstre lages ofte sammen med en eventuell kunde og brukes til å fange opp systemets kravspesifikasjon. Ingenting av det som beskrives i bruksmønsterdiagrammet referer direkte til implementert kode. [W15]. Følgende definisjoner eksisterer på et bruksmønsterdiagram:

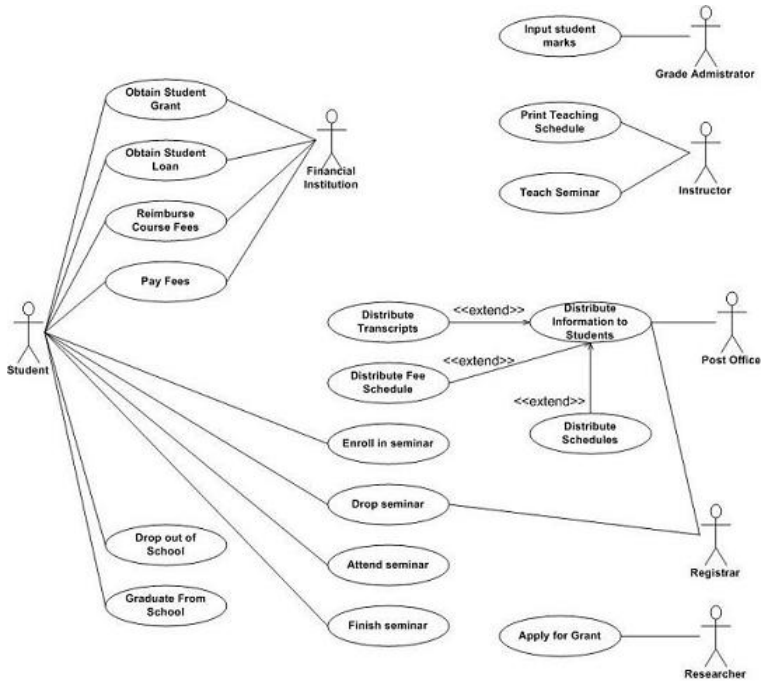
1. Bruksmønsterdiagrammet beskriver en sekvens med hendelser som omfatter kravene bruker har til et system. Bruker representeres som en aktør, og hendelsene tegnes som ellipser med koblinger seg i mellom.

2. En aktør kan være en person, organisasjon eller et annet system som på en eller flere måter samhandler med det aktuelle systemet. En aktør er tegnet som en strekfigur.

3. Koblinger eksistere mellom aktører og hendelser. Disse tegnes som linjer. En kan også bruke piler for å tydeliggjøre hvilken vei en kobling går. Koblingene representerer ikke dataflyt, men hvem som starter en relasjon.

4. Systemavgrensing kan tegnes som bokser rundt en samling komponenter. Dette for å vise at systemet kan være delt opp eller avgrenset i flere deler. Systemavgrensing er ikke ofte bruket i bruksmønsterdiagram, men muligheten er der. Det samme gjelder for pakker. Får man får store og omfattende diagram, kan disse deles opp og legges i pakker som skal representere en samling nærliggende bruksmønsterdiagram. En kan også organisere pakkene etter hvordan de forskjellige diagrammene representere de forskjellige delene av systemet.

Figur 13 viser et eksempel på et bruksmønsterdiagram.

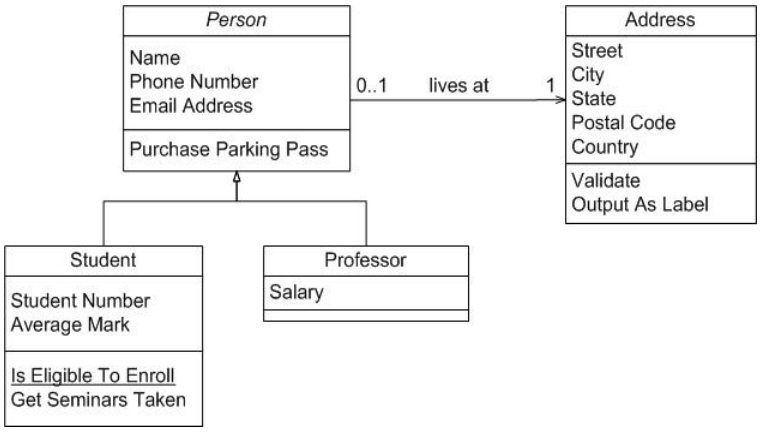


Figur 13 – UML Bruksmønsterdiagram [Campos, 2003]

3.1.2 Klassediagram

Klassediagram [W4] er det som oftest er forbundet med UML. Diagramtypen beskriver og dokumenterer objektorienterte systemer, dets klasser, metoder og sammenhengen mellom klassene. Klassediagram har et stort bruksområde. Brukken spenner fra spesifikasjon og systemdokumentasjon til en bruk for raske overblikk over systemet. Oftest brukes det i designfasen, men flere utviklingsverktøy kan også lage diagrammene basert på eksisterende kode, noe som fungerer som dokumentasjon. Dette er en viktig funksjonalitet, da den spesifikasjonen man starter med ofte ikke stemmer med det ferdige produktet.

Figur 14 viser et klassediagram med av fire klasser. To av klassene arver egenskaper fra en annen klasse. Avhengighet mellom klasser og tilhørende kardinalitet vises også.



Figur 14 – UML klassediagram [Campos, 2003]

3.1.3 Sekvensdiagram

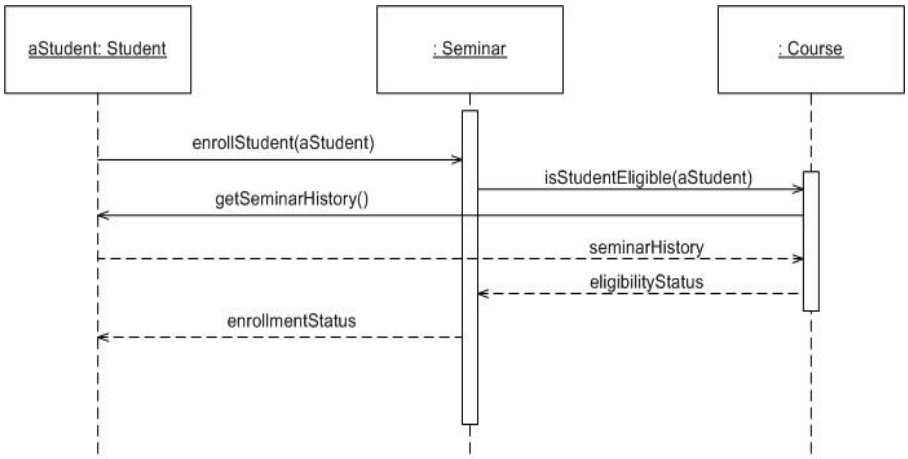
Sekvensdiagram modellerer den logiske flyten mellom komponenter i et system. Man får et visuelt bilde av hvordan klassene samhandler. Dette kan brukes til både dokumentasjon og for å validere logikk. Sekvensdiagram viser oppførselen til et system, i motsetning til klassediagram som viser den logiske sammensettingen av klassene systemet består av.
Dette kan modelleres på flere nivå [Campos, 2003]:

Brukerscenario:
En deskriptiv måte å vise hvordan systemet blir brukt. Kan utledes på analysenivå fra bruksmønsterdiagram. Forskjellen fra bruksmønsterdiagram er at sekvensdiagram ikke

bare viser de oppgaver systemet skal utføre, men hvilke komponenter i systemet som utfører oppgavene.

Metodelogikk:
Sekvensdiagram kan brukes til å dokumentere og utforske logikken i et kompleks system. Oftere er det lettere å se sammenhenger når man får det presentert visuelt.

Servicelogikk:
En service er en høynivå metode som kan bli kalt fra flere klienter. Eksempler på dette kan være transaksjoner, web-metoder [Kreger, 2001], eller metoder i et objektorientert miljø.

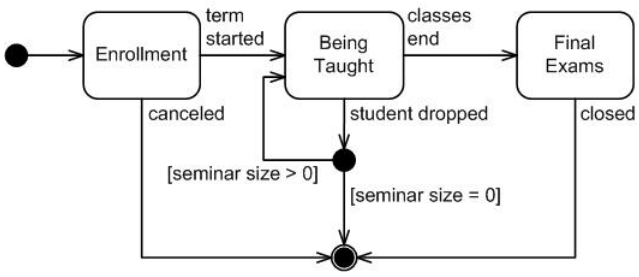


Figur 15 – UML Sekvensdiagram

Figur 15 viser et enkelt sekvensdiagram. Boksene på toppen av diagrammet representerer klasser eller instanser av klasser. Objekter svarer på forespørsler fra andre objekter og aktører, så begge delerer er ofte inkludert i sekvensdiagrammene. De prikkete linjene horisontalt i diagrammet representerer objektets levetid i de aktuelle scenarioet. De hele strekene viser faktisk kall mellom objektene, mens de vertikale boksene representerer aktivitet i klassen. Altså den tiden den utfører et arbeid startet ved et kall til en metode.

3.1.4 Tilstandsdiagram

Objekter har både oppførsel og tilstand. De *gjør* ting og de *vet* ting[7]. Noen objekter er mer komplekse enn andre, og både gjør og vet flere ting. I noen tilfeller blir objekter blir så omfattende og avanserte at de blir vanskelige å forstå ved å kun se på koden eller et klassediagram. For å forstå slike klasser bedre, og spesielt de som oppfører seg forskjellig avhengig av hvilken tilstand de er i kan man bruke et tilstandsdiagram (eng: Statecharts). Det er en diagramtype som beskriver oppførselen til et objekt ut fra hvilken tilstand objekter befinner seg i. Diagrammet beskriver tilstander og overgang mellom tilstandene. Som i UML aktivitetsdiagram har objektene en initial og en avsluttende tilstand. Førstnevnte er tilstanden objekter er i når det blir instansierer. Sistnevnte er tilstanden til objektet når ingen overganger til andre tilstander er mulig. En overgang mellomtilstander blir startet ved en hendelse. En hendelse kan være både intern og ekstern.



Figur 16 – UML tilstandsdiagram [Campos, 2003]

Figur 16 viser et tilstandsdiagram. Diagrammene beskrive en klasse for forskjellig nivå. Figuren viser et diagram som er forholdsvis høynivå. Det vil si det ligger nærmere opp mot analysenivå enn hvis man faktisk modellerte de faktiske tilstandene til en klasse. Dette fordi det viser konkrete tilstander sett fra de involverte personenes synsvinkel. Samtidig viser det bruken av tilstandsdiagram på en bra måte da det involverer de fleste notasjoner brukt i denne diagramtypen.

3.2 Wisdom

Dette kapittelet vil redegjøre for Wisdom. Grunnen til at det todelt. For det første var Wisdom en stor inspirasjon i starten av arbeidet med oppgaven. Det så tilsynelatende ut

[7] Objekter har verdier, og metoder som kan gjøre er arbeid på disse verdiene.

som noe vi kunne jobbe videre på. Det ble ikke sånn da DiaMODL har støtte for interaksjonsdesign, noe Wisdom ikke har.

I [Nunes, 2000] presenteres Wisdom som en lettvekts, UML basert, metode for modellering av programvare. Metoden er ment å fremme ny forståelse for et UML basert prosessrammeverk som fremmer en hurtig og evolusjonær modell for prototyping. Modellen skal sømløst integrere den moderne livssyklusen krevd av et lite utviklingsteam. For å få til dette baserer Wisdom seg på en modellarkitektur som promoterer et sett med UML baserte modeller for å støtte brukersentrert utvikling og grensesnittdesign.

Wisdom introduserer nye modeller for å støtte bruker og rolle modellering, interaksjonsmodellering, dialogmodellering og presentasjonsmodellering. I tilegg foreslår Wisdom en ny arkitektur på analysenivå som støtter integrasjonen av arkitektonisk viktige elementer i brukergrensesnitt. Til slutt er notasjonen i Wisdom laget for å forenkle bruken av UML, og definerer derfor et nytt sett av elementer å modeller. Disse elementene er laget for å støtte Wisdom arkitekturen og prosessen. [Nunes, 200, s.1: oversatt]

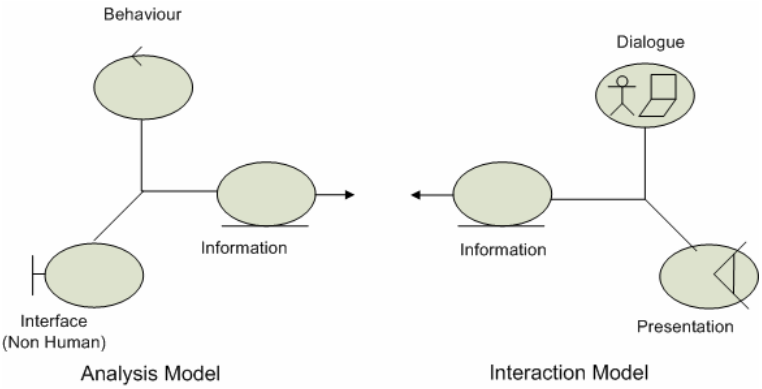
Wisdom er en konseptuel arkitektur på samme måte som MVC og UPA [10 i Nunes, 2001]. Det vil si at den ikke tar hensyn til konkret design eller implementasjon. Wisdom introduserer et brukersentrert perspektiv på analysenivå. Dette gjør den ved å sørge for følgende punkter [Nunes, 2001: oversatt]:

1. Arkitekturen bygger på brukergrensesnittkunnskap som fanger essensen av eksisterende vellykkede og gjennomtestede arkitekturmodeller.
2. Arkitekturen integrerer den eksisterende analysemodellen i UPA, samt bestreber å samarbeide om artefakter og sporbarhet mellom fagfeltet MMI og systemutviklingsmodeller.
3. Arkitekturen prøver å fremme en ansvarsdeling mellom systemets interne funksjonalitet og brukergrensesnittet.

4. Til slutt, så retter arkitekturen seg etter standarden UML både på semantisk og notasjonsnivå.

Wisdom foreslår to nye dimensjoner: *analysemodell*, interaksjonsmodell *og* sammenhengen mellom de to [Figur 17]. Interaksjonsmodellen omfatter dimensjonene, informasjon, dialog og presentasjonsdimensjonene. Analysemodellen deler informasjonsdimensjonen med interaksjonsmodellen, i tilegg til at den omfatter det fysiske grensesnittet og dets oppførsel. Det blir altså presentert to informasjonsrom som deler en dimensjon som binder arkitekturen i systemet sammen med brukergrensesnittet.

I [Nunes, 2000] oppsummeres det med at de har vist hvordan UML kan bli brukt å til representere en konseptuel arkitektur for interaktive systemer. Det er foreslått flere tilnærminger til både konseptuelle og implementerbare arkitekturer. Analysemodellen i UPA kritiseres for å være for systemsentrert. Wisdom er en naturlig utvikling av UPA's informasjonsrom, og tilfører notasjon både for presentasjon for bruker og dialog med denne. I tillegg presenteres en notasjon for en arkitekturmodell. På den måten bygger Wisdom bro mellom disse to domenene, og legger grunnlag for noe som kan bli en verktøystøttet utvikling.



Figur 17 – Arkitektur i Wisdom

Analysemodellen opererer med tre analyseklasser, standardisert i UML som stereotyper Figur 18 viser de tre analyseklassene i kolonnen til høyre [Nunes, 2000: oversatt].

Entitetsklassen (eng: Entity)

Dette er klasser i analysemodellen som er ment å representere varig informasjon. Det kan for eksempel være et grensesnitt til en database eller klasser som representerer modellen i MVC [3.1.5]. Resten av systemet henter data fra disse klassene.

Kontrollklassen(eng: Controller)

Har samme rolle som i MVC. Den skal koordinere, sekvensere transaksjoner og kontrollere dataflyten mellom andre klasser. Man regner med at dette er klasser som håndterer foretningslogikken i systemet.

Grenseklassen(eng: Boundary)

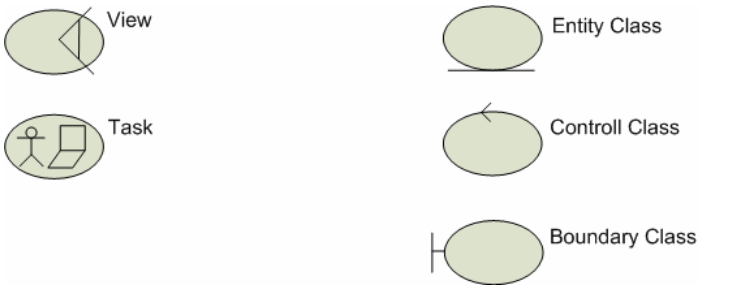
Dette er klasser som brukers til å håndtere utveksling av data mellom kjernesystemet og andre deler av systemet. Undersystemer eller andre nærliggende system kan også være mulige klienter. Den nye interaksjonsmodellen til Wisdom tar for seg den overordnende organisering av brukergrensesnittet i interaktive systemer. Modellen strukturerer interaksjon med bruker, og utsetter håndtering av brukergrensesnittstiler, teknologier for implementasjon og andre begrensinger til de etterfølgende design- og implementasjonsklassen. De to stereotypklassene brukt til dette er:

Presentasjon(eng: View)

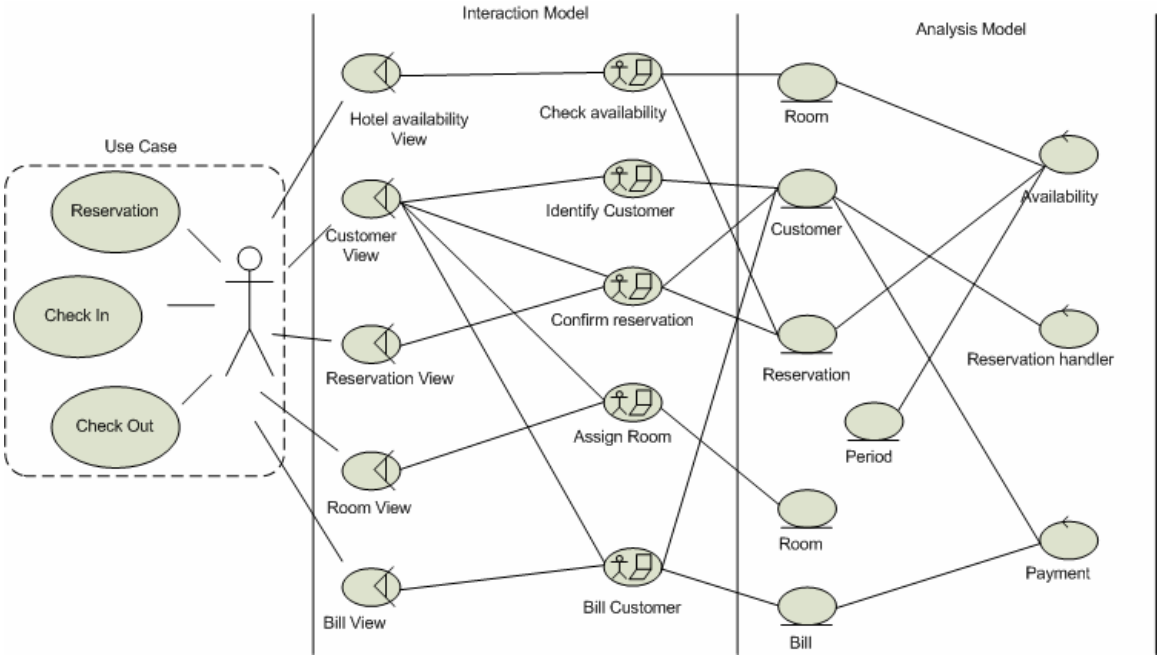
Dette er klasser som skal representere en komponent eller skjermbilde som presenterer noe for bruker. De sammen komponentene fanger også brukers interaksjon. Dette er klasser som mer spesifikt enn et UML brukermønsterdiagram spesifiserer hva bruker skal kunne gjøre med systemet.

Oppgave(eng: Task)

Oppgaveklassene brukes til å modeller strukturen til dialogen mellom bruker og system. De har ansvar for å kartlegge dataflyt som går frem og tilbake mellom system og bruker, eventuelt å sekvensere og validere denne.



Figur 18 – Wisdom-elementer komponenter



Figur 19 – Bruk av Wisdom

Figur 19 viser et praktisk eksempel på hvordan Wisdom blir brukt. Til venstre er et UML brukermønsterdiagram med en aktør og oppgaver systemet skal være i stand til å utføre. Likhetene med UML brukermønsterdiagram slutter hvis vi beveger oss til høyre for aktøren. Her går det koblinger fra aktøren til interaksjonsdelen av Wisdom modellen. Den viser hvilke presentasjonsbilder aktøren blir presentert for. Hvert presentasjonsbilde har en eller flere koblinger til en eller flere oppgaveklasser. Den vertikale linjen representerer skillet mellom interaksjonsmodell og analysemodell.

Analysemodellen viser entitetsklasser som representerer de data en oppgaveklasse skal arbeide mot. For eksempel arbeider klassen ”Bill Customer” (no: Fakturer kunde) mot entitetsklassen ”Bill” (no: Regning). Helt til høyre i analysemodellen finner vi kontrollerklassene. Det er de klassene som tar seg av hvordan ting gjøres i systemet. I

Eksemplet med fakturering av kunde har man klassen ”Payment”(no: betaling). Denne håndterer logikken rundt nettopp betaling.

3.2.1 Kritikk av Wisdom

I forbindelse med den delen av oppgavens arbeid som retter seg mot interaksjonsmodellering og applikasjonsstøtte for dette er det mulig å rette litt kritikk mot Wisdom. Ikke fordi det ikke er et solid arbeid, tvert i mot, men det adresserer ikke nødvendigvis de problemene det uttalte at det gjør. Den kritikken man kan rette mot Wisdom går på dets rolle i et systemutviklingsscenario. Det baserer seg på UML bruksmønster diagrammer, og utvider disse til å støtte modellering av et sett handlinger en bruker skal kunne gjøre med et system og dets brukergrensesnitt.

Wisdom er ment å berøre området interaksjonsmodellering. Men modellen er ikke egnet til å si noe om hvordan den faktiske interaksjonen skal foregå. Modellen presenterer et sett komponenter bruker skal arbeide med og vektlegger at implementasjonen skal legges til realiseringsfasen. Dette medfører lite eller ingen merverdi for bruker eller designer som skal enes om hvordan systemet skal se ut, og oppføre seg i bruk. Utvikler kan ha bruk for å se hvilke kjernekomponenter systemet måtte bestå av for å realisere scenarioene i et UML bruksmønsterdiagram, men dette er mulig med eksisterende UML diagrammer. Hvis interaksjonsmodellering er et konkret mål med Wisdom, ikke bare et utvidet verktøy for konseptuel modellering, vil ikke da DiaMODL være bedre egnet?

Etter hvert som Wisdom ble gått gjennom ble det samtidig jobbet parallelt med en problemstilling. Det ble da tydelig at DiaMODL, og en eventuell utvidelse av det vil tilføre mer merverdi til flere av rollene i et utviklingsscenario. Wisdom ble derfor ikke valgt som utgangspunkt. Sett fra en annen side, nemlig praktisk erfaring, er Wisdom en høyaktuell kandidat til å supplere UML bruksmønsterdiagram. Det vil på samme tid det tar å skissere et bruksmønsterdigram kunne gi mye mer informasjon om resten av systemet. Man vil på samme tid som man begynner å spesifisere hva systemet skal kunne gjøre, være i stand til å spesifisere grunnkomponenter i det. Det er verdifullt.

4 DiaMODL

UML har blitt industristandard hva modellering av komponenter i systemer angår. Brukergrensesnittmodellering og modellbasert brukergrensesnittmodellering har ikke kommet like lang [Trætteberg2, 2002], og selv om det finnes flere prosjekter som har som mål å utvide UML slik at interaksjonsmodellering støttes har det ikke blitt standardisert [Nunes, 2000]. DiaMODL prøver å adressere noen av de utfordringene som finnes innen feltet dialogmodellering. Dette kapittelet vil redegjøre for denne funksjonaliteten.

4.1 Hva er DiaMODL

Dialogmodellering hører til feltene interaksjonsdesign og interaksjonsmodellering, hvor det er en av flere metoder [Braaten, 2006]. I motsetning til konkret interaksjonsmodellering som fokuserer på de forskjellige dialogenes sammenheng med interaksjonskomponenter, beskriver en dialogmodell hvordan en bruker opererer på et system via systemets grensesnitt. Dialogmodellen beskriver også dataflyten mellom komponenter, aktivering av komponentene og i hvilken sekvens de brukes.

I hybriden DiaMODL er disse to perspektivene integrert i et og samme modelleringsspråk.
DiaMODL baserer seg på UML i form av tilstandsdiagrammer og delvis på Pisa-interaktorer [6 i Trætteberg2]. DiaMODL er ment å være fleksibelt i den betydning at det kan brukes til modellering av konkrete brukergrensesnitt eller for en mer abstrakt modellering. Statecharts ble valgt som utgangpunkt da noe av målet med arbeidet var å gjøre det enklere å lage verktøy som støttet DiaMODL-modellering ved bruk av UML [Trætteberg1, 2002].

4.2 DiaMODL bruk og eksempler

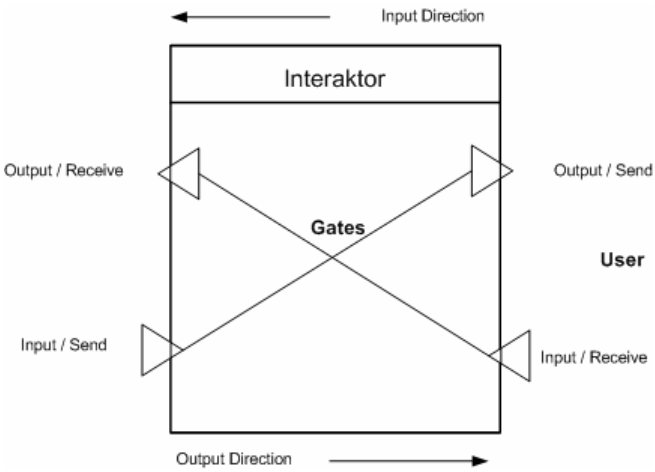
DiaMODL består av hovedkomponenter interaktorer, beregninger/porter (eng: Computations/Gates), koblinger (eng: Connections) og funksjoner. I tillegg vil det bli redegjort for begrepene sett og objekt. Bruken av UML Statecharts for å formalisere oppførsel ble valgt fordi det forenklet arbeidet med å nå et av to mål med DiaMODL,

nemlig å kunne gjøre det enklere å kunne støtte en eksekvering av modellerte brukergrensesnittkomponenter (eng: runtime execution).

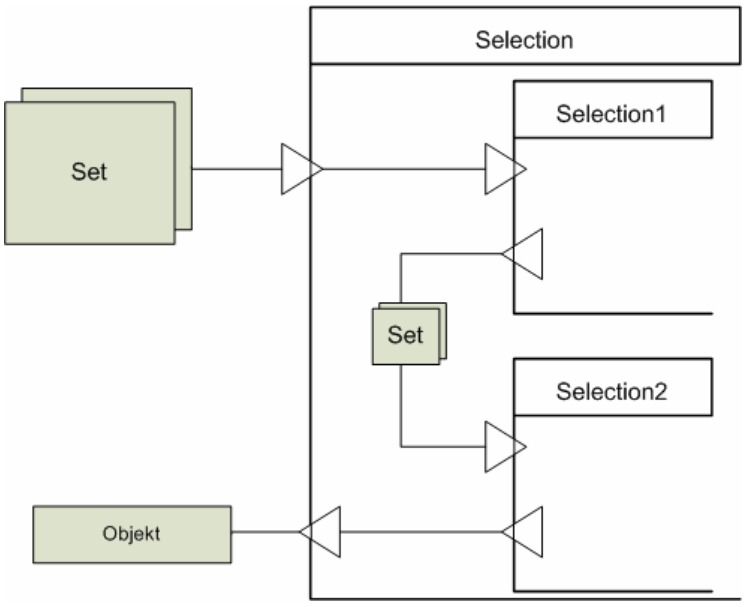
Interaktorer

En interaktor representerer interaksjonen med bruker eller andre deler av et system. Den tar i mot input fra bruker, og sender dette videre inn i systemet, på samme måte som den kan presentere data den får fra systemet til brukeren. Trætteberg beskriver i [Trætteberg, 2003] at en interaktor kan sees på som en spesifikasjon av et designproblem som kan løses ved å komponere eksisterende interaktorer sammen. Den kan også ses på som en spesifikasjon av et løst problem i håp om at det kan samsvare med fremtidige designproblemer. I prinsippet er en interaktor uavhengig av hvordan den er laget og hva den består av. Den er en spesifikasjon av en implementert oppførsel. En annen måte å si det på er at det er en ”sort boks”, hvor innholdet er skjult for bruker. Funksjonaliteten er derimot ikke skjult. Bruker vil være eksponert for informasjon fra interaktoren, samtidig som han vil kunne sende data inn. Systemet som interaktoren eventuelt sender data til er heller ikke eksponert for noe av interaktorens implementasjon. Dette gjør det mulig å sette sammen interaktorer, bestemme dataflyten mellom dem og således modellere et system bestående av standardkomponenter. Dette er samme tankegang som flere brukergrensesnittverktøy bruker i dag (eng: toolkits). Man setter sammen standardkomponenter med en ferdig definert funksjonalitet, og programmerer det underliggende systemet mot grensesnittet til disse komponentene.

Informasjonsoverføringsaspektet i DiaMODL er strukturert med og bygd opp av interaktorer som igjen består er definert av: 1) Et sett av porter som definerer interaktorens eksterne grensesnitt til systemet, brukeren og andre interaktorer, 2) et sett av koblinger som bærer data mellom portene, og 3) kontrollstrukturen som utløser eller forhindrer dataflyt mellom porter på interaktorens inn- og utside og langs koblinger. [Trætteberg, 2001: oversatt]. Figur 20 viser en interaktor og dens mulige funksjonalitet. Den venstre siden er systemsiden, hvor data ut fra og inn til systemet modelleres. Brukersiden er til høyre og representere som kommer fra eller blir vist til bruker.



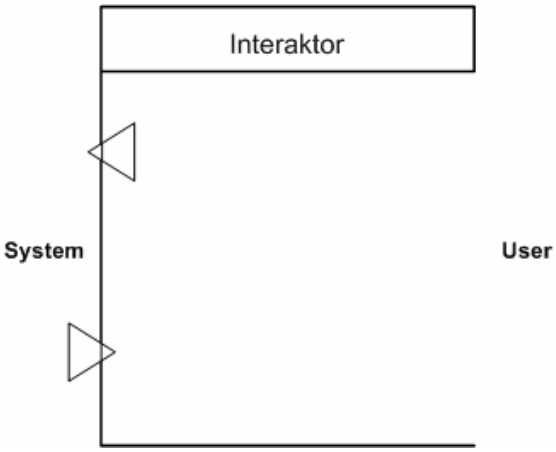
Figur 20 – DiaMODL interaktor [Trætteberg, 2001]



Figur 21 – Sammensatt (nøstet) DiaMODL interaktor

Figur 21 viser en interaktor bygd opp av andre interaktorer. I dette tilfelle en sekvens av seleksjoner, først fra et sett, så fra et valgt undersett. Dette kan modellers som over. Eksempelvis kan man tenkte seg at fra det første settet man velger fylker. Når man velger et fylke, får kommer det opp en liste med byer. Av disse velges en enkelt by.

Notasjonen i figur 20 er nå endret til slik den kan ses i figur 22. Dette for å gjøre det mer tydelig at interaktoren er åpen i den ende enden for kommunikasjon med bruker. Den venstre siden, som er systemsiden, har beholdt den opprinnelige formalismen.

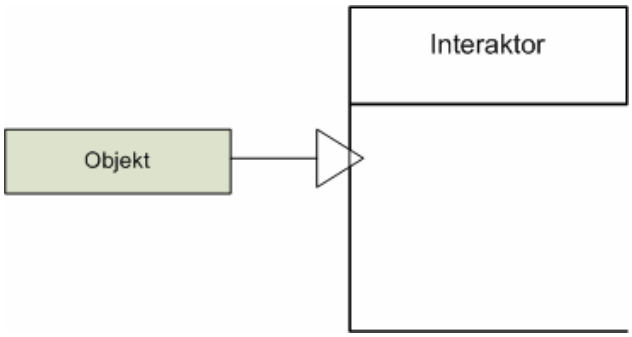


Figur 22 – Interaktor – ny versjon

Porter

Tilhørende interaktorer er et sett med porter. En port er en beregning med spesiell funksjonalitet, og skiller seg fra en beregning den brukes til å transportere verdier over grensen til en interaktor [Trætteberg2, 2002]. En interaktor kan både motta verdier og sende fra seg verdier gjennom portene, da disse interaktoren representerer koblingspunkter for interaktorens dataflyt.

Figur 23 – Port (eng: Gate)



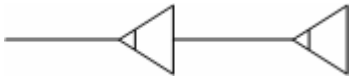
Figur 24 – Dataflyt fra et objekt (systemet) gjennom en input/recieve port

Figur 23 og 24 viser en port og bruken av den. Den brede delen representerer basen som mottar data, mens tuppen i den spisse enden sender fra seg data. Med mindre porten ikke er tillagt ekstra funksjonalitet skjer det ingen endring av data ved passering. Portene på

figuren sender data inn til en interaktoren. Motsatt vei vil porten sende data ut fra interaktoren og inn til systemet eller andre interaktorer.

Koblinger

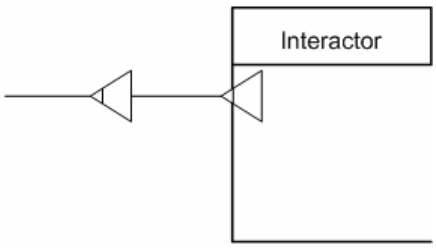
Koblinger gjør det mulig for data å transporteres mellom interaktorer. Den representeres bare ved en strek som går mellom en av de ovenfor nevnte komponentene. Figur 25 viser dette.



Figur 25 – Kobling

Beregninger

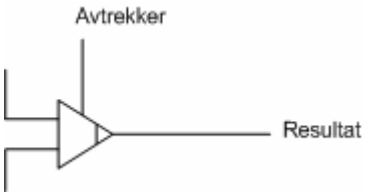
Beregninger skiller seg fra porter ved at den tilbyr en metode for å beregne nye verdier i tillegg til å sørge for informasjonsflyt ut og inn av en interaktor som porter gjør. Den representeres på samme måte som en port, men skilles fra porten ved at den ikke har noe tilhørighet til en interaktor. Figur 26 viser praktisk bruk av en beregning.



Figur 26 – Bruk av beregning

En beregning kalkulere et ”typet” resultat fra et sett med argumenter. I dialogmodellen er argumentet verdier som kommer via koblinger som kommer inn på funksjonens flate side. Det kalkulerte resultatet blir sendt ut fra funksjonens spisse side. [Trætteberg1, s119: oversatt]. På den brede siden kan man motta verdier fra mer enn en kilde. På oversiden av funksjonen har man en tilkobling fra en *avtrekker* (eng: trigger). Dennes funksjonalitet er åpning og lukking av funksjonen for gjennomgående trafikk. Dette fungerer på to måter. Enten er den åpen eller lukket, eller den er lukket hele tiden for så å åpne seg kun den tiden det tar å slippe gjennom data. Oppsummert kan en beregningens rolle se slik ut:

1. Motta verdi
2. Kalkulere resultat
3. Sende fra seg kalkulert verdi

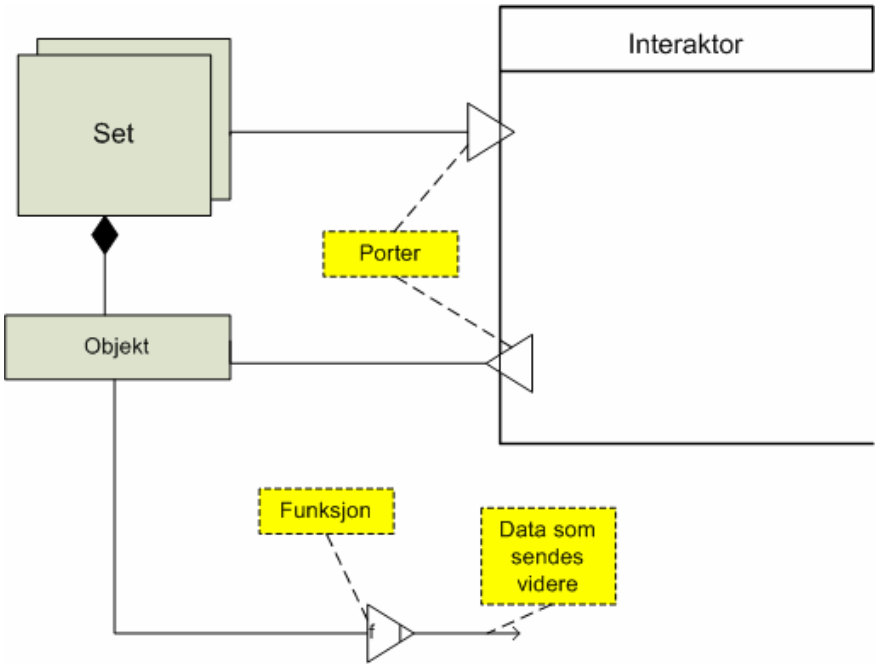


Figur 27 – DiaMODL spesifisering av funksjon

Funksjonen i figur 27 tar i mot to tallverdier. Når en knapp blir trykket (avtrekkeren), så kalkuleres summen av de to verdiene, og sendes til videre.

Sett og objekt

Både figur 21 og 24 viser bruken av komponentene *sett* og *objekt*. Dette er ikke spesifikke komponenter men notasjoner for å illustrere mengden data som flyter inn, ut og i mellom interaktorer. Et sett brukes for å illustrere en mengde flere enn ett objekt. Mer bestemt en samling av en type objekter. For eksempel en liste over byer. Et objekt er entallsrepresentasjonen av objektet by.

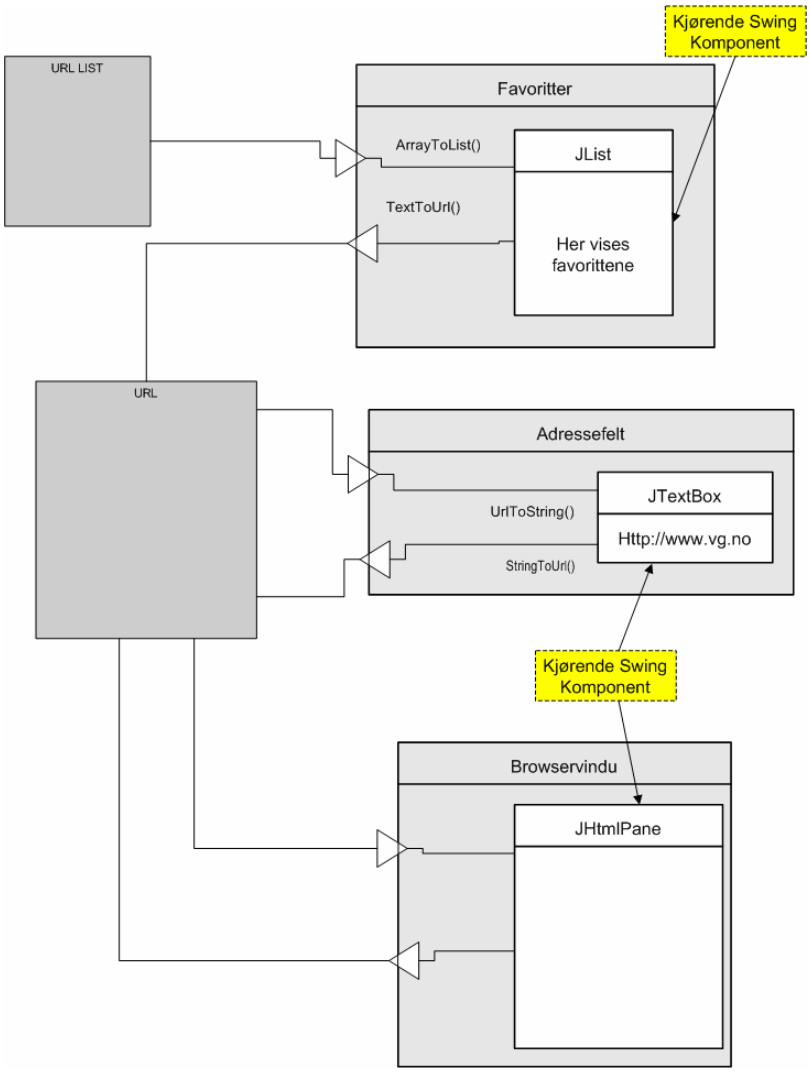


Figur 28 – Gjennomgang av DiaMODL-elementer

Figur 28 viser en interaktor får inn et sett med data. Eksempelvis kan dette vises i interaktoren som en liste over elementene i settet. Bruker velger et av elementene som så sendes inn i systemet fra interaktoren. Man har et sett med objekter, og det ene objektet som blir valgt hører til det opprinnelige settet. UML-notasjon er brukt sammen med DiaMODL for å vise dette. Den viser også hvordan en funksjon er lagt på en kobling. Denne funksjonen kan brukes til å gjøre endringer på de data som kommer fra den ene koblingen, blir endret, for deretter å bli send videre.

4.2.1 Begrepet Placeholder

Et sett er en samling objekter som er input til, eller output fra en interaktor. Objektene er implementasjonen av interaktorens spesifikasjon [Trætteberg, 2001]. ”Placeholder” (no: noe som holder eller omfavner noe) er et felles og forenklende begrep. Konkret er en placeholder bare en mindre formell representasjon av objekter og sett. Den skiller ikke mellom dem og kan representere begge entitetene. Dette skille ble definert for under arbeidet med oppgavens problemstilling og så førte det til en diskusjon rundt objekter, sett og placeholders faktiske funksjonalitet. Figur 29 viser et eksempel på bruken av forskjellige placeholdere. Samtidig er figuren et eksempel på hvor enkelt man kan modellere forholdsvis omfattende funksjonalitet. En interaktor gir designeren stor frihet og gjør det mulig å tenke seg at hele systemer, eller delsystem, blir modellert som en interaktor for å skjule kompleksitet. De placeholdere brukt i dette eksemplet er ”URL List” og ”Url”. Oversatt til sett og objekt så er førstnevnte et sett med objekter som representerer en Url.



Figur 29 – Nettleser modellert i DiaMODL

Figuren inneholder ingen beregning, men portene er tillagt funksjonaliteten å representere url-objektene som tekst. Dette er data som forandres i det de flyter gjennom portene og kan derfor betraktes som beregninger.

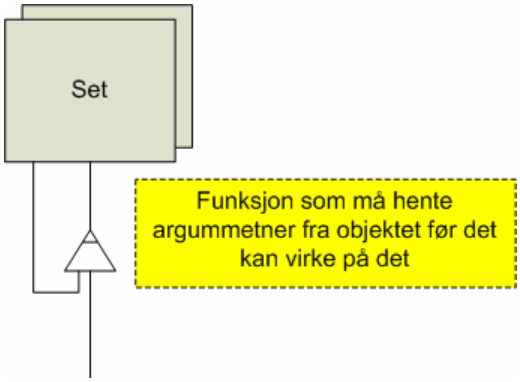
Modellen i figur 29 har tre interaktorer. De representerer tre hovedkomponenter i en nettleser. Øverst er en liste med favoritter. Deretter kommer en interaktor som representerer en felt hvor man kan skrive inn en nettadresse. Til slutt, og nederst, er interaktoren som representerer vinduet hvor nettsiden blir vist. Hvordan de forskjellige komponentene i hver enkelt interaktor er implementert i ikke synelig for bruker, noe som skjuler unødvendig kompleksitet. Interaksjonen mellom komponentene er det som er viktig i en modelleringssammenheng. Placeholderne i modellen har to funksjoner; den øverste representerer enn liste over nettadresser, Url'er, som sendes inn til interaktoren

som viser de frem som en liste over lagrede favoritter. Den andre holderen representerer en enkel url.

Url-objektets verdi kan bli satt på tre måter. (1) Bruker velger en adresse fra favorittlisten, (2) bruker skriver inn en adresse i adressefeltet og (3) man aktiverer en lenge i nettleseren. Portene og koblingene mellom interaktoren viser at hvis man velger en lenke fra favorittlisten oppdateres holderen for url-en. Den blir vist i både adressefeltet og i nettleseren. Skriver man inn en adresse i adressefeltet, blir nettleseren oppdatert, men ikke favorittlisten. Det motsatte skjer hvis man aktiviserer en lenke i nettleservinduet, adressefeltet blir henter den aktuelle adressen fra holderen og viser frem denne.

4.2.2 Utvidelse av beregninger

For å vise at man gjøre en operasjon på et objekt eller et sett[8], har det blitt foreslått å endre DiaMODL-notasjonen ved å innføre et bruk av beregninger koblet direkte til et annet element. En slik notasjon viser at data hentes fra et element. Deretter blir det gjort beregninger på verdiene som er hentet. Til slutt blir verdiene sendt tilbake til elementet. Diskusjonen som førte til dette hadde utspring i et ønske om å kunne representere at en beregning har mer kompleks oppførsel en kun en øyeblikkelig beregning av verdier som passerer.

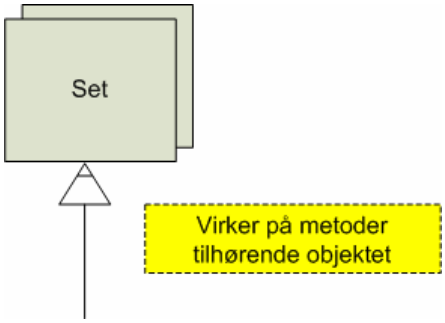


Figur 30 – Beregning med utvidet funksjonalitet - opprinnelig notasjon

Utfordringen var å finne en notasjon som viser denne funksjonaliteten uten at det forveksles med porter eller beregning. Figur 30 viser en mulig notasjon, men denne kan

[8] Selv om placeholder er det generelle begrepet som ofte er enklest å bruker er det noen ganger viktig å skille mellom settet og et faktisk objekt.

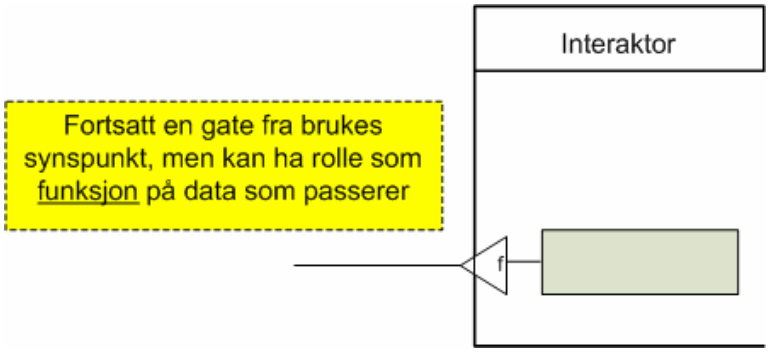
lett blandes sammen med interaktorer. Det som ble foreslått var å sette en gate helt inntil kanten på det aktuelle objektet eller settet (figur 31). På figuren står denne beregning på undersiden av objektet, men det er viktig å merke seg at dette ikke har noen betydning for funksjonaliteten.



Figur 31 – Beregning med utvidet funksjonalitet - foreslått notasjon

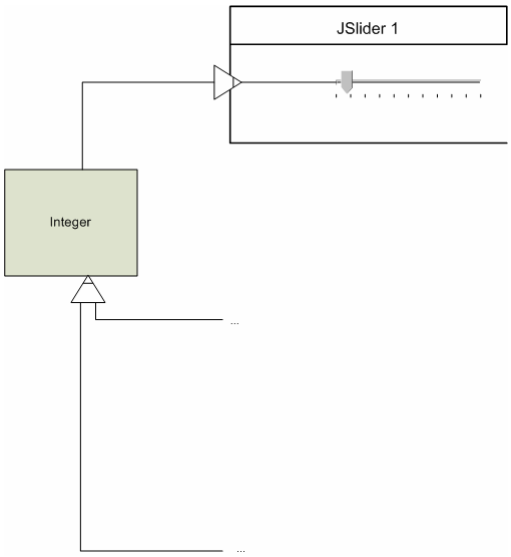
Et annet alternativ var å bruke en gate merket som en funksjon (figur 32). Faren med denne notasjonen er en kompromittering av en port, siden denne notasjonen allerede er brukt for å vise at en port har en beregningsmetode. En port med denne funksjonaliteten er fortsatt en port, og må ikke bli oppfattet som noe eget selv om den handler på data som passerer gjennom. Eksempelvis kan dette være en Integer.parseInt()-funksjon, brukt hvis interaktoren skal sende fra seg et tall mens tallet for brukeren skriver inn er tekst (Figur 33).

Figur 32 – Gate med funksjon



Figur 33 – Bruk av gate med funksjon

Den diskuterte funksjonaliteten er ikke implementert i spesifikasjonen til DiaMODL men er allikevel av interesse da den tilfører en økt funksjonalitet. Figur 34 viser en mer komplett DiaMODL-modell med denne funksjonaliteten.



Figur 34 – Bruk av beregning med utvidet funksjonalitet

5 Problemstilling

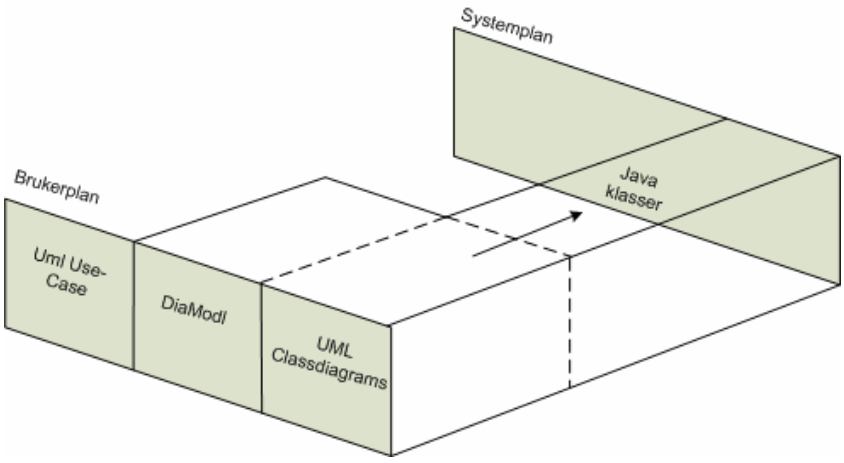
Programmering er ikke forskning [Hartvigsen, 1998], men kan brukes som er verktøy for å undersøke om en problemstilling er mulig å løse. Utgangspunktet for arbeidet med denne oppgaven var et ønske om å kunne utføre et praktisk arbeid relatert til DiaMODL. Et naturlig valg var å se på en eller annen form for verktøystøtte. I [Trætteberg1, s188] beskriver Trætteberg et behov for å kunne bruke DiaMODL sammen med konkrete GUI-(brukergrensesnitt) komponenter. Med det menes muligheten til å bruke konkrete grensesnittkomponenter sammen med DiaMODL-representasjonen av de samme komponentene. DiaMODL-editoren [6.2] representerte den verktøystøtten som det fra starten av et ønske om å utvikle. Samtidig er den naturlig som utgangspunkt for en ny og mer konkret problemstilling.

Videreutvikling basert på DiaMODL-editoren attraktivt av flere grunner. For det første kombinerer editoren abstrakt modellering og konkrete GUI-elementer. Dette er viktig når oppførselen til en prototyp skal spesifiseres. Imidlertid mangler editoren muligheten for å lage en kjørbar prototyp som realiserer funksjonaliteten i DiaMODL-modellene. Generert kode vil være et viktig verktøy for utvikler, og samtidig føre til at utvikler og designer kan jobbe tettere. En annen effekt er at generert kode vil tilføre editoren egenskaper som støtter en iterativ designprosess. For det andre vil en videreutvikling være med å korte ned veien fra DiaMODL som en spesifikasjon til et konkret og reelt produkt. En trede effekt er muligheten for bedre inkludering av bruker, noe det legges vekt på i en brukersentert prosess. Først fordi bruker kan se GUI-elementene allerede i modelleringsfasen Deretter kan det genereres en kjørbar prototyp generert på bakgrunn av modellen. I tillegg til å se den konseptuelle modellen vil da bruker kunne se et konkret produkt.

Mangelen på kodegenerering er det sentrale i oppgavens problemstilling. Forrige avsnitt beskriver momenter som gjør at en kodegenerering basert på DiaMODL-modeller dekker ønsket om å kunne tilføre en reell merverdi både til DiaMODL-editoren og rollene i en systemutviklingsprosses.

5.1 Teoretisk bakgrunn for problemstillingen

I oppgavens innledning ble begrepene brukerplan og systemplan presentert. Som vist er UML-klassediagram er den konkrete representasjonen som binder brukerplanet og systemplanet sammen. En logisk sammenheng mellom UML og DiaMODL utenfor brukerplanet ville ført til at DiaMODL hadde en kobling til systemplanet, selv om denne ville være indirekte. Implisitt ledet dette mot en oversetting av DiaMODL til UML. Spørsmålet er bare hvor gevinsten i dette ligger? DiaMODL er basert på UML[9], og er ment som en forbedring innen et spesifikt domene UML ikke støtter[10]. Hva vil man da vinne på å gå direkte tilbake til UML? Det måtte i så fall være som en representasjon av et helhetlig system som omfatter både den logiske delen av systemet og koden som representerer interaksjon og utseende. Figur 35 viser at man med muligheten for å editere DiaMODL-modeller i sanntid også berører den konkrete delen av brukerplanet (UML klassediagrammet). Med det menes at DiaMODL er en abstrakt notasjon, men implementasjonen av DiaMODL i DiaMODL-editoren er konkret. Objektene som i editoren representerer DiaMODL-modellen kan modelleres i et klassediagram. Figuren viser at DiaMODL på grunn av dette strekker seg mot systemplanet samtidig som det grenser til den konkrete delen av brukerplanet.



Figur 35 – DiaMODL og systemplanet

Figur 2 [s.8] viser hvordan DiaMODL er brukt som et mulig ”lim” mellom brukerplanet og systemplanet. Trætteberg har i [Trætteberg1, 2002] valg å ikke gå i detalj hva bruker-system dimensjonen angår, men viser med sitt arbeid at de verktøy han presenterer for

[9] DiaModl er basert på Uml tilstandsdiagrammer (Statecharts).
[10] Interaksjonsdesign [1, s1].

arbeid med brukerplanet også relaterer til systemplanet. Oppgavens innledning presenterer systemplanet som en fjerde dimensjon. Denne fjerde dimensjonen viser hvordan systemplanet kan bli sett på som et underliggende system som komponentene i brukerplanet kobles mot. Konkretiseres dette kan man se på brukerplanet som et sett med kode for å håndtere brukers interaksjon med et brukergrensesnitt og dets komponenter. Systemplanet vil da representere et sett med kode som ikke har noen direkte kobling til hendelser i brukergrensesnittet. Det tar kun seg av eventuell logisk funksjonalitet.

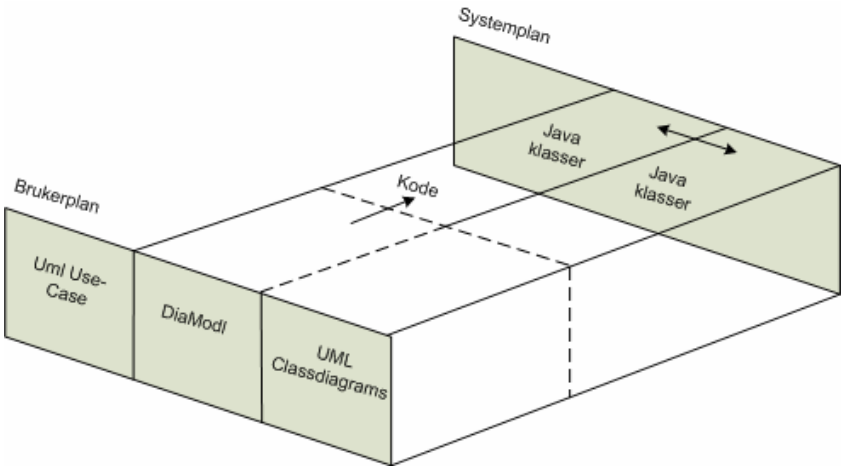
En annen måte å tolke sammenhengen mellom de to planene på er at brukerplanet representerer de abstrakte komponentene som må realiseres for at det skal eksistere et grensesnitt mot brukere av systemet. Funksjonaliteten i systemplanet går på tvers brukerplanets tre dimensjoner, og av den grunn er det naturlig å betegne systemplanet som en fjerde dimensjon. Systemplanet vil med denne tolkingen representere hele kodebasen som skal til for at et brukergrensesnitt, inkludert underliggende logikk, skal kunne realiseres med det utseende og den funksjonaliteten som angis. DiaMODL representerer både lag 1 og 2 i figur 2. Tolkes systemplanet som over kan man også regne inn den koden som må til for at brukergrensesnittet skal kunne realiseres som et kjørende program. Dette er kode som kan modelleres med klassediagram, noe som fører til at DiaMODL også berører lag 3 i figuren.

5.2 Problemstilling

En metode for å rask kunne gå fra modell til utvikling er kodegenerering. Med det menes at kode genereres basert på en modell egnet for dette. Spesielt finner man denne funksjonaliteten i verktøy for objekt- og database modellering. Et klassediagram viser komponentene (klassene eller pakkene) et objektorientert system består av. Disse kan være basert på eksisterende kode, men kan også modelleres først og deretter realiseres som kode.

For DiaMODL, som i utgangspunktet er en abstrakt representasjon, eksisterer det ikke noen tilsvarende kodemodell. En DiaMODL-modell viser interaksjonen mellom elementer i et brukergrensesnitt. Hvordan man løser dette i form av kode er ikke gitt selv om DiaMODL-elementer eksisterer som objekter i DiaMODL-editoren.

Overordnet kan problemstillingen formuleres slik: Er det mulig å utarbeide en metode som beskriver hvordan en DiaMODL-modell kan oversettes til kjørbar kode? Går man et skritt videre er det naturlig å se om en slik metode vil kunne realiseres på et generelt nok vis til at kode kan genereres for alle DiaMODL-modeller.



Figur 36 – DiaMODL og systemplanet forts.

Figur 36 viser en utvidelse av figur 35, og er beskrivende for oppgavens problemstilling. Forskjellen fra figur 35 er at figuren viser at DiaMODL kan kobles til systemplanet ved å oversette den abstrakte modellen til kode. Koden på systemplanet vil igjen kunne modelleres i et klassediagram. Resultatet er at DiaMODL både kan brukes som en abstrakt representasjon av et brukergrensesnitt samtidig som det er et konkret grunnlag for kode på systemplanet.

5.3 Delproblem og løsningskrav

Dette avsnittet vil utdyping problemstillingen. Utdypingen vil bli presentert som et sett punkter. Punktene vil også fungere som løsningskrav. Overordnet fokus at koden som genereres gir et bra programmatisk utgangspunkt for utvikler. Samtidig skal den genererte koden representere det arbeidet designeren har gjort på best mulig måte. Disse to aspektene kunne havnet i konflikt med hverandre da det på en side er ønskelig med så kvalitetsmessig bra generert kode som mulig, mens det på den andre siden skal være mulig å hurtig kunne generere kode som realiserer et design.

Punktene under dekker det arbeidet som vil bli redegjort for i kapittel 6 (Konstruksjonsrammeverk) og 7 (DiaModlGen, kodegenerering av DiaMODL-modeller). I tillegg fungerer punktene som grunnlag for retninger i et eventuelt videre arbeid:

- Den genererte koden skal så langt det er mulig implementere oppførselen spesifisert i modellen. Dette betyr at det for hvert modellelement må genereres kode som implementerer tilsvarende oppførsel. Dette innbærer:
    - En BeanPlace (Interaktor) må oversettes til en klasse som kan presentere et GUI-element.
    - En beregning eller port må oversettes til noe som beregner en ny verdi fra en eller flere andre verdier.
    - En kobling må oversettes til kode som lytter på en endring i kilden og videreformidler denne til et mål.
    - En interaktor/tilstand grupperer elementer som aktiveres/deaktiveres som en enhet. I praksis er dette implementasjonen av DiaMODL-editorens støtte for Statechart XML [5.4]
- Det må utvikles og presenteres en algoritme som sikrer at alle modell-elementer blir besøkt og brukt i kodegenereringen[11].
- Basert på utfordringer og mangler funnet i arbeidet må det utledes regler for en generell genereringsmetode, samt hva som må endres for at denne skal kunne implementeres.
- Det må presenteres og drøftes retningslinjer for et videre arbeid.

5.4 Avgrensing

Et problem kan virke uoversiktlig så lenge det ikke avgrenses. Ser man på et problemet som en helhet kan det være vanskelig å se hvordan det skal angripes og løses. En vanlig strategi, som også denne oppgaven baserer seg på, er å dele opp problemet i mindre delproblem som hver for seg kan løses. I oppgavens tilfelle dreier dette seg om hvordan man med programkode kan løse delproblemer som igjen kan brukes til utforsking av muligheter. En naturlig fremgangsmåte er å dele opp den koden på en slik måte at hver del gjenspeiler et konkret delproblem.

[11] Traversering i dette tilfelle til si at alle objekter i en modell må kunne besøkes, alle parametere må kunne hentes ut og det må være mulig å finne sammenhengen mellom objektene.

Punktene i forrige avsnitt spesifiserer ikke hvor detaljert hvert av dem skal løses. En overordnet avgrensing for oppgaven er at hvert punktet skal løses på en måte som gjør det mulig å løse neste punkt. Oppgaven presenterer løsninger som i flere tilfeller er valg ut blant en rekke andre løsninger. Konkret kan dette være eksempler som er vurdert og undersøkt, men som det ikke har blitt funnet nødvendig å implementere støtte for. Det kan også være ting som ikke har nok relevans i forhold til problemstilling og løsningskrav.

Under er det presentert flere punkter som er med på å avgrense oppgaven:

- DiaMODL-editoren og det tilhørende kjøretidsmaskineriet er en omfattende applikasjon og begge deler er i kontinuerlig utvikling. Det gjør det viktig å fokusere på den eksisterende funksjonaliteten og bruke den som utgangspunkt. Oppgaven er basert på DiaMODL-editoren som den eksisterte når arbeidet med utviklingen av kode for å generere begynte. Endringer som har blir lagt til i ettertid har ikke blitt tatt hensyn til[12].
- Alle modeller kan ikke testes empirisk. Det gjør det nødvendig å velge ut momenter i løsningen som til en akseptabel dekker den viktigste funksjonaliteten.
- Arbeidet med å implementere kode som dekker de spesielle eksemplene vil kun dekke kun et utvalg av brukergrensesnittkomponenter. Hvordan dette kan utvides vil bli gjort rede for.
- Flere plasser i den implementerte koden ville det være ønskelig å ha en generell funksjonalitet. De plassene hvor dette ikke er av avgjørende betydning for funksjonaliteten vil det være brukt forenklede løsninger.
- Presentasjonen av en mulig generell løsning vil være basert på problemer funnet i eksemplene. Problemene vil bli vist i mer komplekse DiaMODL-modeller, noe som vil sannsynliggjøre at løsningen er generell nok, selv om dette ikke lar seg bevise.
- Funksjonalitet som ikke har nok relevans vil bli utelatt. Grensetilfeller vil bli kort redegjort for.

[12] Med unntak av støtte for tilstander (statecharts) da dette ikke medførte vesentlig endringer.

Med dette er en konkret problemstilling presentert sammen med et sett krav som må oppfylles for at problemene skal kunne anses som løst. Avgrensing legger føringer på omfanget av hvert punkt. Resten av oppgaven vil fokusere på løsning av problemstillingen, resultater av dette og føringer for et mulig videre arbeid.

6 Konstruksjonsrammeverk

En løsning som innebærer en implementering blir ofte todel, noe som også gjenspeiles i de presenterte løsningskravene. I oppgavens tilfelle er hovedarbeidet genereringskoden, eller mer bestemt genereringsmetoden. For å kunne implementere denne trengs det et rammeverk. Dette kapittelet vil redegjøre for implementeringen av DiaMODL-editoren, et konstruert rammeverke brukt til en løsning og viktige funksjonelle aspekter.

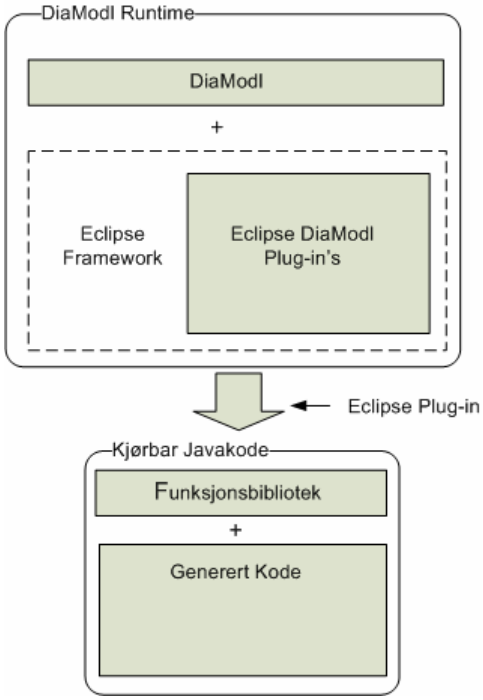
De funksjonelle kravene presentert i problemstillingen relaterer i stor grad til arbeidet med rammeverket og den logiske struktureringen av både generert og genererende kode. Begge må være oppdelt i en logisk struktur. Denne strukturen vil, og bør, gjenspeile de oppgavene som må fullføres for at en kodegenerering skal være mulig.

6.1 Utviklingsrammeverk

Selv om det som blir vist her er den beste løsningen er det også mulig å vurdere andre tekniske plattformer som grunnlag for en kodegenerering. En DiaMODL-modell modellert i editoren kan lagres. Dette skjer i en XML-fil som beskriver strukturen til modellen. Rammeverk for å håndtere XML-filer finnes for mange språk, noe som åpner for muligheten til å kunne bruke disse til kodegenerering. Ulempen er at objektmodellen DiaMODL består av vil måtte være representert i det valgte språket. Med en direkte utvidelse av DiaMODL-editoren vil objektmodellen være tilgjengelig i utgangspunktet. En manuell implementering er da ikke nødvendig. Eclipse, og Java, utgjør naturlige verktøy å jobbe med da. Eclipse er en utvidbar plattform og Java er objektorientert og har et komplett sett med brukergrensesnittelementer. Totalt sett vil dette føre til enklest arbeidsforhold, raskest mulig utvikling og mest mulig fokus på løsning.

Figur 37 viser en skjematisk fremstilling av hvordan en teknisk løsning kan bygges opp. Den øverste rammen representerer DiaMODL-editoren. DiaMODL-kjøretidsmaskineriet (eng: Runtime) realiseres ved at DiaMODL som modelleringsmetode implementeres som utvidelser til rammeverket Eclipse. Den underste firekanten representer den kjørbare javakoden. Hovedsakelig er dette generert kode, men om ønskelig kan et manuelt sett

med kode brukes sammen med den genererte koden. Figuren viser dette som ”Funksjonsbibliotek”[13]. Dette vil være nødvendig som støtte i arbeidet med å utvikle de spesielle tilfellene til en generell metode. Det kan også brukes til å dekke funksjonalitet som DiaMODL-editoren enda ikke støtter. Med det menes at en utvikler kan selv kan skrive kode som manuelt kalles fra den genererte koden for å tilføre funksjonalitet som ikke er mulig å spesifisere i DiaMODL. Pilen mellom de to firkantene representerer oppgavens løsning.



Figur 37 – Problemstilling

Løsningskravene i kapittel 6 presisere at det må presenters en struktur som den genererende koden skal være bygd opp etter. Videre sies det også at det på bakgrunn av denne strukturen skal presenteres et komplett program som generer kode av en DiaMODL-modell. For at dette skal være mulig må det eksistere et rammeverk implementasjonen kan bygge på. DiaMODL-editoren er implementert som en utvidelse til Eclipse. En tilsvarende utvidelse ville være passene for den genererende koden. Utvidelsen har fått navnet “DiaModlGen”.

Med utviklingsrammeverk menes i oppgavens tilfelle det miljøet og det verktøyet som blir brukt i arbeidet med løse oppgavens problemstilling. Sett fra en utviklers ståsted er et

[13] Funksjonsbibliotek i betydningen en samling ekstra funksjonalitet.

rammeverk en kombinasjon av verktøy og annen kode som kan brukes som et hjelpemiddel for å lage egen kode. I oppgavens tilfelle er koden som utvikles både avhengig av det rammeverket DiaMODL-editoren utgjør og det verktøyet DiaMODL er implementert i.

Utviklingsverktøyet Eclipse er i seg selv et rammeverk hvor funksjonaliteten er implementert som en rekke utvidelser (eng: plugins). I [W2] beskrives Eclipse som en plattform strukturert som en kjerne som står for kjøringen av programmet, samt et sett med tilleggsfunksjonalitet i form av utvidelser. En utvidelse tilfører rammeverket funksjonalitet ved å koble seg til forhåndsbestemte utvidelsespunkter. For eksempel er det grafiske brukergrensesnittet i Eclipse en utvidelse som ligger på toppen av rammeverket. Eclipse fremstår som et helhetlig program for brukeren, men det er ikke et samlet javaprogram som startes. Ved oppstart aktiviseres rammeverket, og dette laster dynamisk de utvidelsene det trenger.

I Eclipse er utvidelser en egen type prosjekter. For å kjøre eller teste den implementerte utvidelseskoden kjøres prosjektet som en "RunTime Workbench". I motsetning til kompilering og kjøring av normale javaprogram som startes som frittstående program, kjøres utvidelser ved at en ny instans av Eclipse starter. Utvidelsen er da med i denne instansen på lik linje med andre medfølgende utvidelser. Den ferdige implementasjonen av oppgavens utvidelse medfører ingen store visuelle endringer i DiaMODL-editoren. Den eneste forskjellen er en knapp på menylinjen med tittelen "Generer". Etter endt modellering trykkes det på knappen på for å kjøre genereringskoden. I [R26] beskrives det hvordan Eclipse er bygd opp. Rammeverket er delt i tre hovedkategorier:

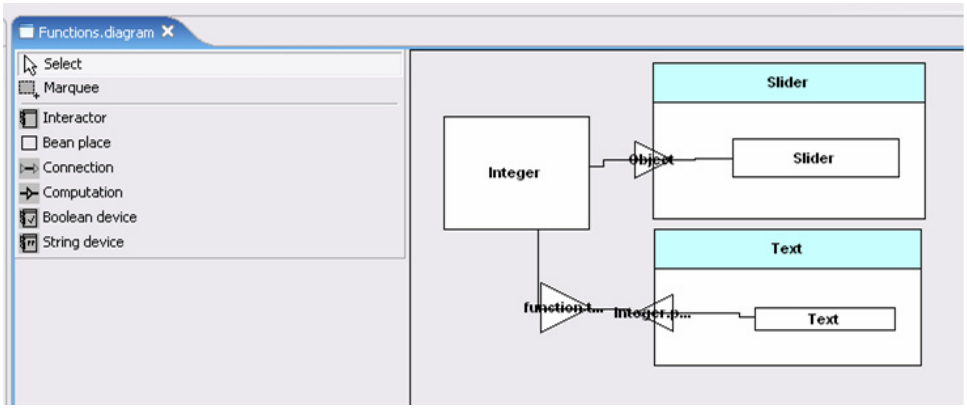
- Kjernen som tar seg av kompilering og manipulering av javakode.
- Brukergrensesnittdelene som håndterer den visuelle delene av Eclipse.
- Feilfinningsdel (eng. debug) som brukes til finne feil spesifikke for Java.

Skal man lage kode som utvider rammeverket gjøres dette ved å definere utvidelser som kobles til plattformens utvidelsespunkter. Disse er et sett med veldefinerte utvidelsespunkter hvor nye utvidelser kan kobles til for å tilføre systemet en utvidet

oppførsel [R26: oversatt]. Sett fra systemets perspektiv er ikke utvidelsen som blir laget annerledes en systemets egne utvidelser. En manifestfil ("manifest.mf") beskriver hvordan koden skal settes sammen og avhengigheter mellom deler av pakken. Filen "plugin.xml" er den som er viktigst for utvikler. Den beskriver de definerte tilleggene.

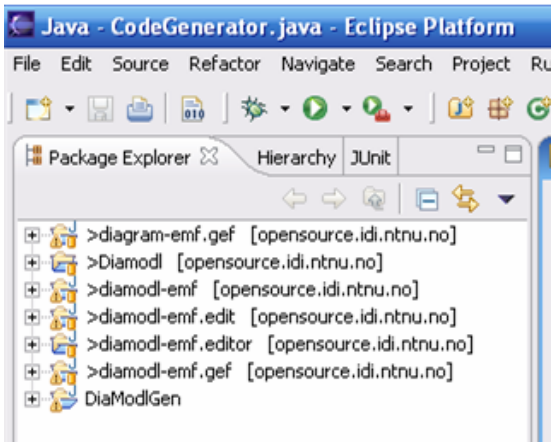
6.2 Utvidelsen DiaModlGen

På lik linje med utvidelsene beskrevet i forrige avsnitt er DiaMODL-editoren en utvidelse til Eclipse. Den er igjen avhengig av andre utvidelser, blant annet av en grafisk utvidelse som gjør det mulig å lage en editor hvor en utvikler kan modellere med DiaMODL-komponenter, lage koblinger mellom komponentene, legge til funksjonalitet, og til slutt få modellen til å kjøres i modelleringsvinduet [5.1]. Figur 38 viser DiaMODL-editoren.

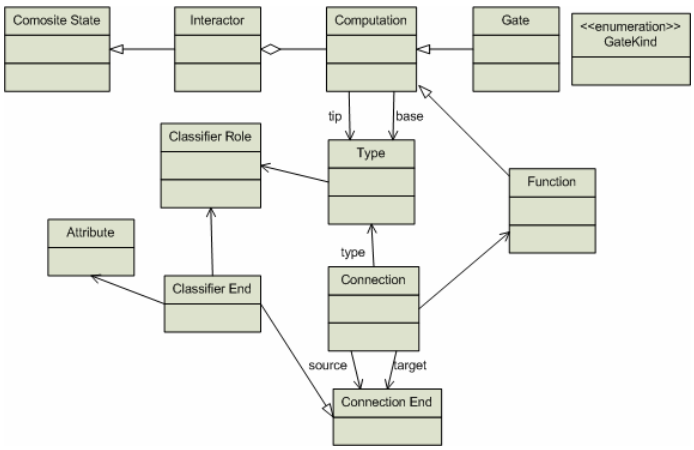


Figur 38 – Modellering i DiaMODL-editoren

Til høyre er området hvor modelleringen forgår. Feltet til venstre viser tilgjengelige modelleringskomponenter. Modellen lagres som en XML- fil etter endt modellering og kan således jobbes med i flere omganger. Figur 39 viser pakkene, eller i dette tilfelle utvidelsene, som oppgavens prosjekt DiaModlGen er avhengige av. Pakken DiaMODL kan ses sammen med andre utvidelsene som brukes til å integrere DiaModlGen med andre utvidelser. Figur 40 viser metamodellen av DiaMODL som editoren implementerer og bruker som grunnlag for modellering.



Figur 39 – Import av DiaMODL-utvidelser



Figur 40 –DiaMODL Metamodell

I oppgavens tilfelle starter Eclipse med det rammeverket som utgjør DiaMODL-editoren. For å kjøre oppgavens kode er det på et grafisk utvidelsespunkt laget lagt til en knapp på menyen i Eclipse. Når denne knappen blir aktivert blir DiaModlGens kode kjørt. XML-filen i kodeliste 1 brukes av rammeverket når en utvidelse skal instansieres. De samme avhengighetene som vises i figur 39 kan kjennes igjen her. I tillegg kan det legges merke til "<actionSet>". Her beskrives hvilket utvidelsespunkt som skal brukes og hva som skal legges til. I dette tilfelle beskriver filen at det er et menyelement med tilhørende navn og ikon som skal legges til. "<id>" viser hvor utvidelsens kode befinner seg.

```
<plugin
   id="diamodlgen"
   name="DiaModlGen Plug-in"
   version="1.0.0"
   provider-name=""
   class="diamodlgen.DiaModlGenPlugin">
   <runtime>
      <library name="diamodlgen.jar">
         <export name="*"/>
      </library>
   </runtime>
   <requires>
```

```
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="diamodl-emf"/>
      <import plugin="org.eclipse.emf"/>
      <import plugin="org.eclipse.emf.common.ui"/>
      <import plugin="org.eclipse.gef"/>
      <import plugin="no.hal.uiml"/>
   </requires>
      <extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Logisk navn"
            visible="true"
            id="diamodlgen">
         <menu
               label="DiamodlGen"
               id="DiamodlGenMenu">
         </menu>
        <action
               label="Generate Java"
               tooltip="Generate javacode from model"
                        icon="icon.png"
               class="diamodlgen.RunAction"
               menubarPath="DiamodlGenMenu/DiamodlGenGroup"
               toolbarPath="DiamodlGenGroup"
               id="diamodlgen.RunAction">
         </action>
      </actionSet>
   </extension>
</plugin>
```

Kodeliste 1 – Plugin.xml

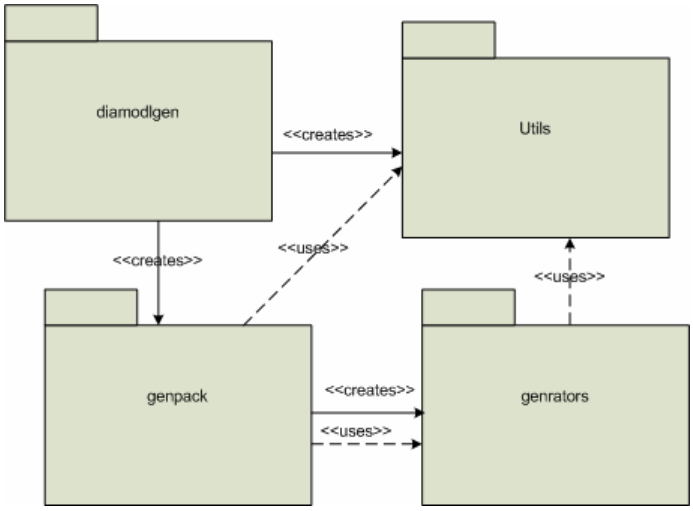
6.3 Genereringsrammeverk

Dette avsnittet vil presentere strukturen, klassene og koden som til sammen utgjør rammeverket for genereringsalgoritmen. Det vil bli gjort rede for hvordan klassene er satt sammen strukturelt, logikken i genereringsklassene og hvordan programmet utgjør et bra utgangspunkt for en videreutvikling.

DiaModlGen er det inn i fire pakker [W12]. Hver av pakkene har avskilt funksjonalitet og kan brukes av hverandre gjennom offentlige metoder [W6]. Pakken DiaModlGen inneholder funksjonaliteten for å instansiere utvidelsen. Pakken inneholder blant annet klassen DiaModlGenPlugin. Den inneholder ikke funksjonalitet som tilfører oppgaven noe og vil ikke bli beskrevet her. Dens viktigste funksjon er at den kaller metoden run() i klassen RunAction.

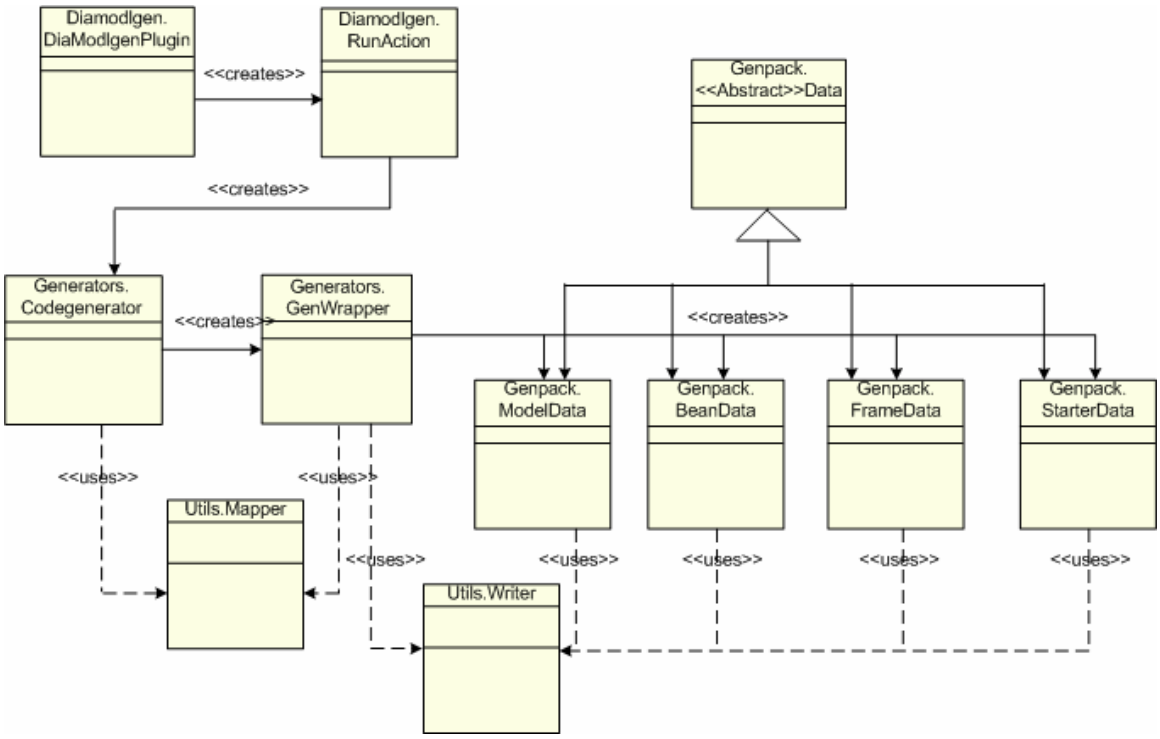
Pakken Utils inneholder klasser med støttefunksjonalitet som brukes av flere andre klasser. I all hovedsak er det klassen Mapper og Writer som ligger her, men pakken kan også brukes til andre fremtidige støtteklasser. Det vil bli redegjort for begge klassene i kapittel 7.

Pakken Generators inneholder klassene som hver for seg skriver den geneerte koden til fil. Det ekisterer en klasse for hver type klasse som skal skrives ut. Disse kalles en av innpakkingsklasse (eng: wrapper) i pakken Genpack. Denne sørger for et sett med metoder for å opprette og sette parametere i de genererende klassene. Genpack inneholder også klassen CodeGenerator Denne mottar en liste med traverserte elementer fra RunAction, og inneholder logikken for å styre hvilken kode som blir generere. Den bruker det settet av metoder GenWrapper tilbyr. Figur 41 viser en oversikt over pakkene som utgjør DiaModlGen.



Figur 41 – DiaModlGen pakkediagram

Figur 42 viser klassene som er beskrevet i dette avsnittet og sammenhengen mellom dem. Man kan se at klassene i Genpack arver fra en abstrakt klasse Data. Denne klassen inneholder funksjonalitet som er felles for alle klassene i pakken Genpack



Figur 42 – DiaModlGen klassediagram

6.3.1 Klasser for kodegenerering

Klassene i pakken Genpack representerer de klassene som til slutt blir generert. Funksjonsmessig er det lagt lite logikk til disse klassene, men desto mer arbeid er lagt ned for at de lett skal kunne utvides. Basert på variabler som er satt i klassene genereres kode ved hjelp av et sett med standardvariabler. Superklassen Data, som klassene arver fra, inneholder felles variabler, metoder og tekst.

Den generelle, og felles, strukturen for alle klassene er som følger:

- Et sett med variabler som inneholder tekst brukt i genereringen. Inkludert i disse er variabler arvet fra superklassen.
- Aksessmetoder for å sette og hente ut disse variablene.
- En metode som ved hjelp av noe logikk fyller denne listen basert på innholdet i variablene.
- Metode som sender klassen til utskrift.

Det store arbeidet med klassen, som også vil bli den mest omfattende delene av en eventuell utvidelser ligger i logikken som håndterer variabler med koden som skal

genereres, samt hvordan denne struktureres. Kodeliste 2 viser deler av en genereringsklasse.

```
public class BeanData extends Data{
        private int classType           = 0;
        public String modelClassName ="";
        public String modelName = "";

         //...Variabler brukt under generering...
        private String stringType            = "Text";
        private String intType               = "Value";
        private String strToInt              = "Integer.parseInt";
        private String intToStr              = "\"\"+";
        private String modelDataType         = "";
        private String componentDataType     = "";
        ...
       //Constructor og aksessormetode
        public BeanData(WriterTool writer,...

        public String getUseAdditionalValue() {
                return useAdditionalValue;
        }
        public boolean isUseMethodFunc() {
                return useMethodFunc;
        }
        public void setUseMethodFunc(boolean useMethodFunc) {
                this.useMethodFunc = useMethodFunc;
        }
        //Legger ved hjalp av logikk alle variablene i listen som skal skrives ut
         public void addPrints(){
                ...

                printList.add(tx_class);
                printList.add(fileClassName);
                if (useMethodFunc){
                            printList.add(methodFunction);
                            printList.add(tx_leftSemi);
                        }

                        printList.add("guiComponent.get");
                        printList.add(GUIComponentMethod);
                        printList.add(tx_leftSemi);
                        printList.add(tx_rightSemi);
                        if (useMethodFunc){
                            printList.add(tx_rightSemi);
                        }
                       if (classType != 1){
                        printList.add(tx_newLine);
                        printList.add(tx_newLine);
                        printList.add(update1);
                        printList.add(tx_leftBrack);
                        printList.add(tx_newLine);
                        printList.add(update2+modelName+")");
                        printList.add(tx_leftBrack);
                        if (classType==2){
                            printList.add(tx_newLine);
                            printList.add("if (!actionHappend){");
                            printList.add(tx_newLine);
                            printList.add("updateHappend = true;");
                            printList.add(tx_newLine);
                        }
                //...Klassen avsluttes
```

Kodeliste 2 – Genereringsklasse

Logikken som setter sammen den sammenhengende teksten arbeider så langt det er mulig kun med variabler. Det er også så langt det har vært mulig blitt satt logiske navn på variablene basert på hvilken del av koden de skal generere.

6.4 Potensielle utfordringer

Konstruksjonen av implementasjonsrammeverket avdekket tre ting som tidlig fremsto som mulig utfordringer i en kodegenerering. Kapittel 7 vil vise hvordan utfordringene er løst.

6.4.1 Statecharts og DiaMODL

Et tillegg til DiaMODL-implementasjonen er støtte for SCXML (StateChart XML). En DiaMODL-modell lagres som et dokument i XML format. SCML kan implementeres i XML-filene for å støtte behandling av modell-elementers tilstand. Med tilstander menes at for eksempel en port kan slås av og på, eller være aktiv ikke aktiv. I [W3C, 2006] beskrives SCXML som et XML-dokument med støtte for tilstander basert på tilstandsdiagrammer (eng: statecharts). I utgangspunktet var hovedbruksområdet støtte for VoiceXML, men det presenteres også andre mulige bruksområder. Et eksempel er et multimodell brukergrensesnitt, hvor et objekts tilstand representerer informasjon som presenteres for bruker, enten via lyd eller bilde. En tilstandsendring skjer etter bruker har respondert på objektets presentasjon [2 i W3C, 2006: oversatt].

I [W16] gis det eksempler på hvordan statecharts i XML kan se ut. Alt som kan representeres i UML tilstandsdiagrammer kan også representeres som SCXML. Foretningsprosesser, navigasjon, interaksjon og dialogmodellering er mulig bruksområder. Kodeliste 3 viser en enkel SCXML notasjon. Når et objekt blir aktivert, endres tilstanden og en tekst skrives ut.

```
<state id="custom" final="true">
      <onentry>
       <my:hello name="world" />
      </onentry>
     </state>
```

Kodeliste 3 – XML Statecharts

Implementasjonen av SCXML i DiaMODL-editoren vil ha betydning for hvordan DiaModlGen generer kode.

6.4.2 Funksjoner

Begrepet funksjoner opptrer i forbindelse med bruk av beregninger og porter i en DiaMODL-modell. Porter og beregninger kan utføre handlinger på data som kommer inn,

og sende disse videre. Funksjoner er den koden som blir lagt inn i portene og beregningen for å utføre beregningene. Per dags dato er det to typer kode som støttes. Det ene er standard javafunksjoner (også benevnt som MethodFunctions). Dette er funksjoner som er bygd inn i språket Java. Den andre muligheten er bruk av scriptspråket Pnuts, som i [W11] beskrives som et lettvekts pråk laget spesielt for Java. Det vil si at når det skal lages en funksjon, så må denne enten finnes i Java, eller skrives i Pnuts. Det er ikke mulig å definere egen funksjoner med skrevet i Java. Det er heller ikke mulig å vi en funksjon kalle annen egenkonstruert kode. Dette er normalt ikke noe problem når modeller skal konstrueres i DiaMODL-editoren, men som det vil bli vist kan dette medføre en utfordring i forbindelse med kodegenerering.

6.4.3 SWT vs Swing

I oppgaven vil begrepene SWT og Swing blir bruk. Begge to er grafiske bibliotek brukt i Java. Forskjellen på dem vil medføre en utfordring i forbindelse med kodegenerering, men som det vil bli vist er de så like at en programmerer som er vant med det ene biblioteket ikke vil ha problemer med å sette seg inn i, og bruke, det andre.

Swing [W7] er det grafiske referansebiblioteket til Java. SWT [W14] er et tilsvarende grafisk bibliotek utviklet av IBM som en del av plattformen Eclipse [W2]. SWT ble originalt utviklet for Eclipse, men kan også brukes utenfor rammeverket. Hovedmålet med SWT var å lage et enkelt grafisk bibliotek som koblet et programs utseende tettere til den plattformen det kjører på. Enkle komponenter i SWT blir kjørt av den underliggende plattformen, mens mer avanserte komponenter blir emulert i Java. Swing tilbyr flere funksjoner, er mer elegant og tilbyr et høyere abstraksjonsnivå. Ser man generelt på det er SWT tilsynelatende enklere å ta i bruk men blir mer komplisert når det skal lages avanserte brukergrensesnitt [W14: oversatt]. Tabell 1 viser at Swing og SWT har de samme komponentene, så det skulle ikke være noen i veien for at utvikler selv kan velge hvilket bibliotek han skal bruke.

Komponent	SWT	Swing
Button	X	X
Advanced Button	X	X

Label	X	X
List	X	X
Progress Bar	X	X
Sash	X	X
Scale	X	X
Slider	X	X
Text Area	X	X
Advanced Text Area	X	X
Tree	X	X
Menu	X	X
Tab Folder	X	X
Toolbar	X	X
Spinner	X	X
Table	X	X
Advanced Table	X	X

Tabell 1 – Sammenligning SWT og Swing

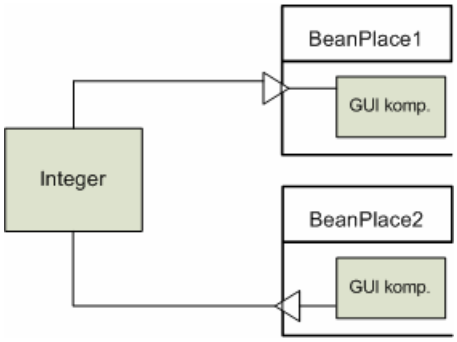
DiaMODL-editoren er en utvidelse av rammeverket Eclipse. Utvidelsene som er tatt i bruk for å muliggjøre en grafisk modellering er også en del av Eclipse, og for å kunne vise konkrete GUI-elementer i DiaMODL-editoren er det av den grunn nødvendig å bruke SWT- biblioteket til dette.

7 Kodegenerering med DiaModlGen

I kapittel 5 ble det presentert et problem som igjen ble delt opp i mindre delproblem. Dette kapittelet vil presentere metoden som er brukt for å løse disse problemene. I tillegg til en problemstilling, presenterte kapittel 5 et sett med løsningskrav som må være oppfylt for at problemet skal kunne sees på som løst. Det er viktig å skille mellom selve problemet som er av en generell art og løsningskravene som legger føringer på et konkret arbeid. Dette, og neste kapittelet, vil presentere arbeidet med løsningskravene. Omfanget av arbeidet ble avgrenset i kapittel 5, og sammen med løsningskravene utgjør avgrensing en mal for hvordan arbeidet er bygd opp.

Den første delen av dette kapittelet vil vise hvordan DiaMODL er realisert i DiaMODL-editoren i form av en objektmodell. Kjennskap til denne er nødvendig da den genererte koden vil være basert på informasjon om objektmodellen og dens sammensetting. Objektmodellens komponenter, struktur og sammenhenger vil bli gjort rede for sammen med en drøfting om hvor mye av informasjonen det er nødvendig å bruke. Deretter vil det bli presentert to DiaMODL-modeller med tilhørende objektmodeller. Disse er valgt ut på bakgrunn av enkelhet og funksjonalitet. Varianter av disse vil bli brukt gjennom hele kapittelet. Med forankring i eksemplene og objektmodellen vil det bli presentert hvordan kode kan genereres. Denne koden vil bli implementert i rammeverket presentert i forrige kapittel. Funn av mangler og utfordringer i den genererte koden vil legge grunnlaget for utredning av en generell metode for kodegenerering.

7.1 DiaMODLs objektmodell



Figur 43 – Utgangspunkt for DiaMODL objektmodell

Kapittel 5.1 presenterer DiaMODLs metamodell[14]. Med metamodell menes at dette er en overordnet og beskrivende modell. Den implementerte objektmodellen som representerer DiaMODL i DiaMODL-editoren baserer seg på denne, men ser noe annerledes ut.

Figur 43 viser en enkel DiaMODL-modell. Den består av to interaktorer som hver har en grafisk komponent. Den grafiske komponenten er instansiert av et BeanPlaceobjekt[15]. Den ene interaktoren sender fra seg data, noe som skjer ved at BeanPlacen sender verdien den henter fra den grafiske komponenten til porten som sitter på interaktoren. Data sendes videre fra porten til en modellrepresentasjon. Den andre interaktoren henter data fra denne modellen inn via sin port og setter verdien på sin grafiske komponent. Mellom komponentene i modellen går det streker som representerer koblinger. Subjektene i dette avsnittet: interaktor, BeanPlace, kobling, port, beregning og modell representere objekter i DiaMODLs objektmodellen. De kan også delvis kjennes igjen fra metamodellen. Objekter er instansierte klasser, og den klassemodellen DiaMODL består av er listet opp under. Klassene er gruppert i den rekkefølgen de arver, så en klasse kan være vist flere ganger.

- *Gate → Computation → Valueplace → ModelElement*
- *BeanPlace → ValuePlace*
- *Interactor → State → ModelElementContainer*
- *Transition → ModelElementAssosiation*
- *Connection → ModelElementAssociation*

I toppen av hierarkiet finner man den abstrakte klassen ModelElement. Alle andre klasser arver fra denne, enten direkte eller indirekte via andre klasser. Flere av klassene i hierarkiet er abstrakte, det vil si at de ikke kan instansieres som objekter. Objektene i DiaMODLs objektmodell er instanser av klassene nederst i hierarkiet og har konkret funksjonalitet. I figuren er det vist en modellrepresentasjon. Denne finnes ikke i oversikten over klasser, noe som vil få betydning senere i arbeidet en generell kodegenerering. Som en start er det nok å vite at modellklassen i figuren er en BeanPlace uten en grafisk komponent. Klassen er kun brukt som holder for en verdi. Dette gjøres

[14] "En presis definisjon av konstruksjoner og regler nødvendig for å kunne lage en semantisk model" [eng : oversatt, W17].
[15] Den grafiske komponenten vises i modellen, men det er klassen beanplace som gjør dette mulig.

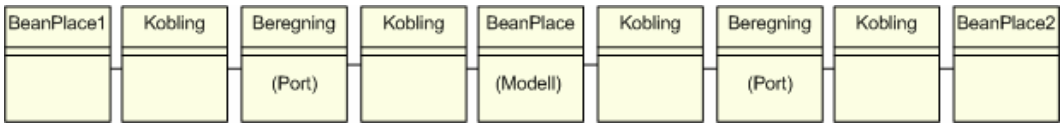
ved å sette klassens parameter ”valueType” til valgt datatype. Et annet unntak når det gjelder konkretisering er klassen Transition. Den ligger også nederst i klassehierarkiet men er ikke synlig i en DiaMODL-modell og vil derfor ikke bli tatt hensyn i dette kapittelet.

7.2 En utflating av objektmodellen

Forrige avsnitts oversikt over DiaMODLs klasser er et hierarki hvor klasser arver fra, og er avhengige av andre klasser. Flere av klassen er abstrakte og derfor ikke synlige i en modell. Det er heller ikke tydelig at alle klasser tilfører direkte funksjonalitet til modellen. Spesielt blir dette tydelig hvis man ser på klassen interaktor. Den tilsynelatende funksjonaliteten er at den er en holder, eller et rammeverk, for porter. I tillegg holder den på et BeanPlaceobjekt. Portene presenteres som en del av interaktoren, og BeanPlacen ser ut til å tilhøre interaktoren på samme måte. I den fullstendige objektmodellen er dette på mange måter riktig, men hvis man tar bort interaktoren og lar portene være igjen sammen med BeanPlacen, ser man at det går koblinger fra BeanPlacen til portene. Videre derifra går det koblinger videre til andre deler av modellen. Både BeanPlacen, koblingene og portene er objekter på lik linje med interaktoren. Tar man bort interaktoren i modellen kan de se ut som man faktisk ikke mister funksjonalitet siden BeanPlacene fortsatt er koblet til portene via koblingsobjekter. Dette leder til en antagelse om at det kan være mulig å overse deler av objektmodellen og allikevel klare å hente ut nok informasjon til at en kodegenerering er mulig. En slik utflating vil være fordelaktig i da man får et mindre antall objekter å forholde seg til. Man trenger kun å ta vare på de objektene som representere en konkret funksjonalitet, noe som kan kalles en utflating av modellen. Med det menes at kun nødvendige objekter blir tatt med og behandlet som de ligger på samme nivå. Under er en oversikt over de klassene som er nødvendig å ta vare på etter utflatingen. Som man kan se så er dette de samme klassene som er visuelt synlige.

- Beregning
- Port (Underklasse til beregning)
- Kobling
- BeanPlace
- Modell (Underklasse av BeanPlace)

Figur 44 viser objektene som vil være igjen etter en utflating av modellen i figur 43. Strekene mellom objektene er assosiasjoner, eller avhengigheter mellom objektene. Alle modellens deler er instansierte objekter, så strekene i objektmodellen må ikke forveksles med strekene i en DiaMODL-modell hvor de representerer koblingsobjekter. Koblingsobjektene har et utgangspunkt og et mål (eng: target og source), noe som gjør det mulig å angi retning.



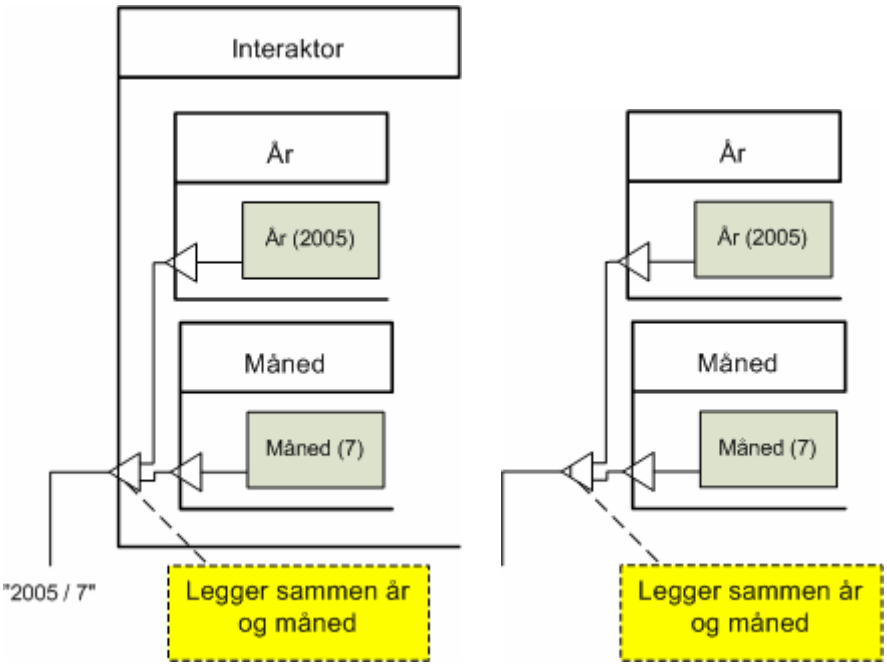
Figur 44 – Utflatet representasjon av DiaMODLs objektmodell

Som presentert i kapittel 4, kan man mellom en port og et annet objekt i modellen ha en eller flere beregninger. En beregning er superklassen til en port. Tar man bort modellenes interaktor vil det ikke være mulig å skille mellom porter og beregninger. Begge objektene ser ut til å sitte mellom andre modellobjekter, bundet sammen via koblinger. Når interaktoren er med brukes porten til å slippe data ut og inn av det område en interaktor definerer. Samtidig kan porten gjøre noe med dataene som passerer gjennom. Det sist er en funksjonalitet den arver fra beregningen. Siden interaktoren ikke er tilstede vil det kun være behov for å forandre data. Resultatet er at en beregning og en port vil kunne representeres på samme måte. Det vil senere bli vist at dette har betydning for kodegenereringen.

En utflating vil føre til at informasjon går tapt. Oppgavens kodegenerering har et begrenset behov for informasjon og utflatingen er et valg som er tatt på bakgrunn av ønsket om å presentere en enkel og oversiktelig metode. På en annen side er det ikke alltid ønskelig med en utflating. Dette gjelder spesielt interaktoren, som har en definert rolle. Den skal avgrense et område og representere en logisk enhet bruker eller andre interaktorer kan kommunisere med. En interaktor kan i tillegg til å være en holder for en grafisk komponent, være en holder for andre interaktorer. Den ytterste interaktoren vil da utgjøre grensen for en logisk enhet, og de nøstede interaktoren vil ikke være synelig for andre interaktorer på utsiden. Funksjonalitet pakkes inn og kan bare brukes gjennom et grensesnitt. I kapittel 5 ble det presentert støtte i DiaMODL for en XML statechart-implementasjon. I praksis vil det medføre en mulighet for å kunne bestemme om en interaktor skal være aktiv eller ikke. Denne funksjonalitet vil ligge i den ytterste

interaktoren, den som utgjøre grensen for en logisk enhet.. Nøstingen og en komplett objektmodell vil være nødvendig hvis denne funksjonaliteten skal være mulig.

Figur 45 viser en DiaMODL-modell hvor interaktoren er fjernet. Forskjellen i funksjonalitet mellom de to modellene er ikke synelig på annen måte enn at porten som tilhørte interaktoren nå kan ses på som en beregning. I modellen hvor interaktoren er tilstedet representer den en enhet hvor man i to tekstfelt kan sette inn år og måned. Interaktoren vil gjennom porten sende fra seg summen av de to verdiene. Selv om man mister denne avgrensingen i modellen hvor interaktoren ikke er med blir resultatet det samme; En beregning legger sammen verdien av data som kommer fra to interaktorer.



Figur 45 – Fjerning av interaktor

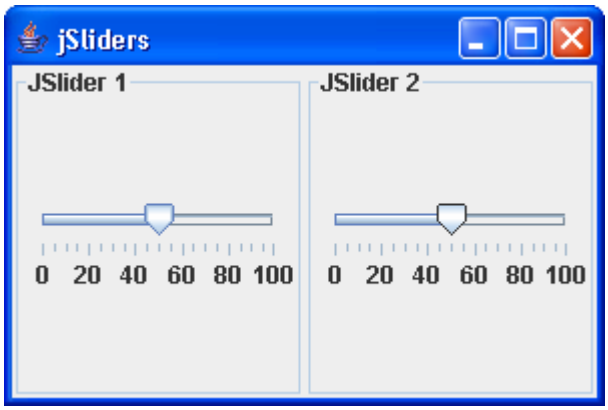
Som det vil bli vist senere i kapittelet så er ikke interaktoren helt uten verdi i en kodegenerering. Når man i DiaMODL-editoren skal plassere en BeanPlace vil interaktoren naturlig følge med. Dette gjør at interaktoren er med i de presenterte modellen selv om det ikke er nødvendig å ta hensyn til den. Foreløpig er dette en sannhet med modifikasjoner. Kodegenereringen vil kun bruke data fra objektene i den utflatede

modellen, men vil samtidig bruke interaktoren for å avgjøre hvilken funksjonalitet de genererte klassene skal ha[16].

7.3 Innledende modelleksempler

For at forsking skal være mulig, er det viktig å ha et grunnlag eller utgangspunkt. For at en metode for kodegenerering skal kunne utledes, vil dette avsnitte presentere to eksempler som vil bli brukt gjennom hele kapittelet. Eksemplene er valg ut fra kravet om at et løsningskonsept skal kunne presenteres på en klar og tydelig måte. Samtidig skal de kunne representere et størst mulig utvalg av utfordringer og konsepter løsningen må kunne dekke. Til sist er det et viktig moment at det arbeid som gjøres med eksemplene må kunne fungere som et godt grunnlag for utledningen av en generell metode.

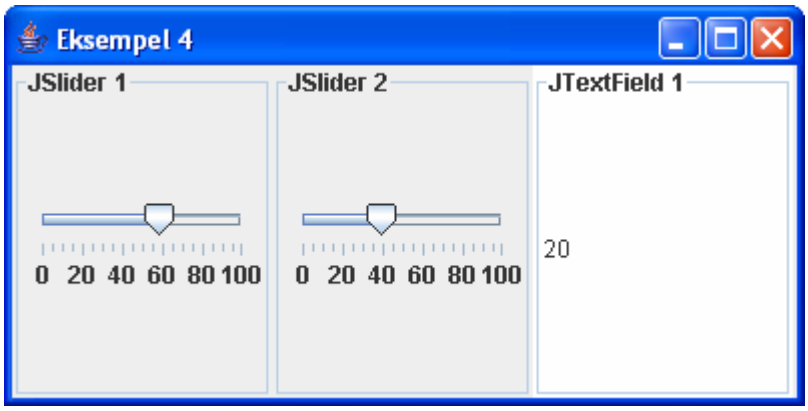
Figur 46 viser et skjermbilde av det enkleste eksemplet (heretter Eksempel1) som kommer til å bli brukt. Programmet består av to Java Swing JSlider [W6] komponenter. Når den ene flyttes på viser den andre samme verdi. I dette tilfelle er det verdien av JSlider2 som bestemmes av verdien til JSlider1.



Figur 46 – Skjermbilde Eksempel1

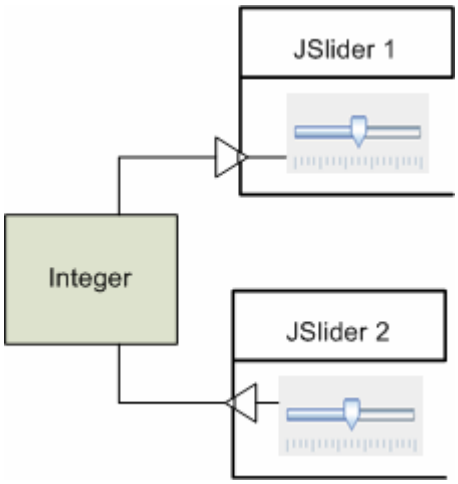
Figur 47 viser skjermbilde til et program med noe mer funksjonalitet (heretter Eksempel2). JSlider1 sin verdi blir satt av summen av verdien til JSlider2 og JTextField. Det viktigste med dette eksemplet er at det må dekke funksjonalitet for å legge sammen data, samtidig som det være mulig å konvertere mellom datatyper.

[16] Som det vil bli vist er klassen som blir generert for de spesielle eksemplene for å representere beanplacen i den genererte koden en hybrid mellom modellens interaktor og beanplace.



Figur 47 – Skjermbilde Eksempel2

Figur 48 viser DiaMODL-representasjonen av Eksempel1. javakomponenten i interaktoren JSlider2 sender fra seg komponentens verdi gjennom en port til en modellrepresentasjon som lager verdien. Modellen holder en verdi av typen Integer, som er den samme datatype den grafiske komponenten sender fra seg. Ingen konvertering mellom datatyper trengs. JSlider1 mottar data gjennom en inngående port og setter komponentens verdi lik modellens.

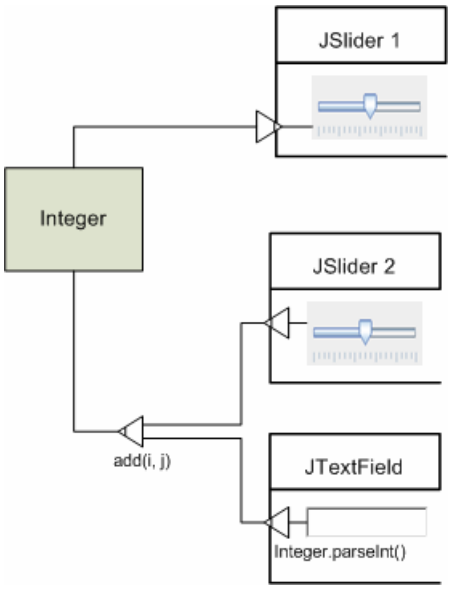


Figur 48 – DiaMODL-modell av Eksempel1

Eksempel1 er det samme som er brukt i kapittel 7.1, men er her instansieringen av de grafiske komponentene tydeliggjort. Modellens tilhørende objektmodell er også presentert i 7.1, og den er den som er av betydning når det skal traversere og genereres kode.

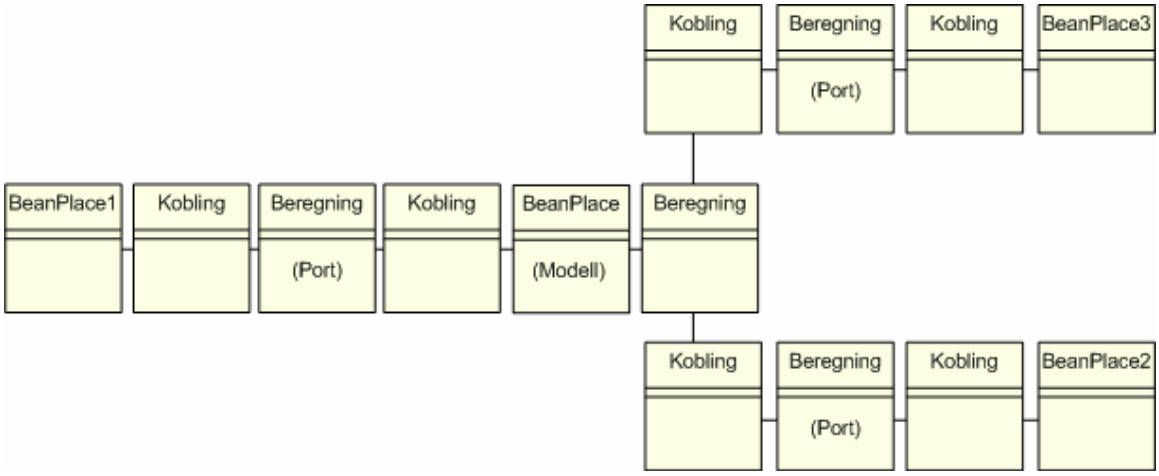
Figur 49 viser DiaMODL-representasjonen av Eksempel2. En interaktor med en JSlider mottar en verdi fra en modell. Forskjellen fra forrige eksempel er at det her er to

interaktorer som oppdaterer modellens verdi gjennom en beregning. Beregningen har en funksjon som legger sammen verden de to interaktorene sender fra seg. Den ene interaktoren er lik komponenten i Eksempel1 og har en grafisk komponent som sender fra seg en verdi lik modellens verdi. Den tredje interaktoren har en grafisk komponent av typen JTextField [W6]. Denne komponentens datatype er ikke den samme som modellen. Det må gjøre en konvertering, noe som skjer i en utgående port.



Figur 49 – DiaMODL-modell av Eksempel2

Figur 50 viser Eksempel2's objektmodell når denne er flatet ut. Interaktorene er ikke med og er erstattet av objektet BeanPlace. Fra disse går det koblinger til portene før to koblinger møtes i en beregning. Porten til interaktoren JTextField har en funksjon, og det samme har beregningen. Objektmodellen viser også en BeanPlace som fungerer som en modell.



Figur 50 – Objektmodell for Eksempel2

Begge eksemplene små og lite kompliserte. Eksempel1 inneholder det minimum av komponenter det er naturlig å inkludere i en kodegenerering. Eksempel2 mer komplekst men fortsatt lite og enkelt. En DiaMODL-modells viktigste funksjonalitet via eksemplen dekket, noe som er en forutsetning for at en kodegenerering skal bli en suksess. Den viktigste hendelsene er at en interaktor sender eller mottar data. Dette er dekket i begge eksemplene. Konvertering av data er dekket i Eksempel2. Det samme er også en annen sentral funksjonalitet, nemlig bruken av beregninger.

Ut fra objektmodellen er det mulig å identifisere en forskjell på en port og en beregning selv om interaktoren er utelatt. I en komplett DiaMODL-modell vil man kunne ha nøstede interaktorer. I et slikt tilfelle ville det være mulig at den ytterste interaktorens port mottok data fra flere av de nøstede interaktorene. Siden en utflatet objektmodell ikke omfatter interaktorer, vil det ikke være aktuelt at en port har andre innkommende koblinger enn den som kommer bra BeanPlacen. En port vil, som det ligger i ordet, være porten som data fra en BeanPlace passerer ut eller inn i gjennom. En beregning kan, og vil ofte, motta data fra flere BeanPlacer. Hvis man bare skulle forholdt seg til en utflatet objektmodell ville det være naturlig å bruke denne forskjellen til å behandle porten og beregningen på forskjellig måte men forskjellen er så ubetydelig at dette ikke vil være aktuelt. Det vil også medføre problemer hvis en komplett modell senere skal brukes til kodegenerering.

7.4 Traversering av objektmodellen

For å kunne generere kode som har den samme funksjonaliteten som en DiaMODL-modell må det være mulig å forsikre som at alle objekter i modellen er besøkt. Basert på

den utflatede objektmodellen presentert i avsnitt 7.2 vil dette avsnitte vise hvordan dette lar seg implementere i form av en traverseringsalgoritme.

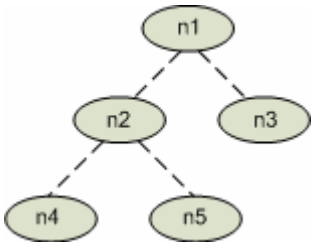
Hvis alle objekter er besøkt, vil det også innebære at alle koblinger mellom objekter er undersøkt. Dette gjør det mulig å finne ut hvordan objekter henger sammen. Hvis man vet dette vet man også hvor et objekt får data fra eller sender data til. For hvert objekt som besøkes kan relevante data hentes ut. Til sammen gjør dette at det er mulig å realisere en DiaMODL-modells funksjonalitet i generert kode.

En objektmodell kan ha et ubegrenset antall objekter. Det er heller igjen begrensninger på hvilke objekter som kan være koblet sammen. For at en traversering skal være mulig må man se mer nøye på hvordan objektmodellen faktisk er bygd opp og henger sammen. Koblingsobjektene er sentrale da de er de objektene som gjør en traversering mulig. Mellom hvert objekt i modellen er det et koblingsobjekt. Dette objektet har en parameter som representerer koblingens utgangspunkt og en parameter for koblingens mål (eng: target og source). Parameteren er en referanser til de objektene koblingen skal binde sammen. Tilsvarende har objektene som ikke er koblinger en liste med referanser til et antall koblingsobjekter. Denne listen vil være like lang som antall koblinger til eller fra et objekt. Starter man i et objekt kan man ved å hente en av objektets koblinger også få tak i objektet i den andre enden av koblingen.

Siden det mellom hvert objekt i modellen vil være et eller flere koblingsobjekt, og koblingsobjektene vil ha et modellobjekt i hver ende, vil koblingene fra nå av bare bli vist som en strek mellom de andre modellobjektene.

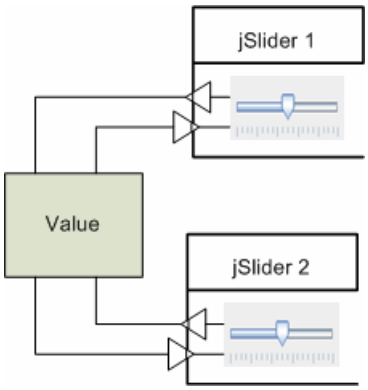
7.4.1 Traverseringsalgoritme

Figur 51 viser et binært tre. Binære trær (eng: binary trees) er en av de viktigste og mest grunnleggende av datastrukturene [W13, 5.1]. Et tre starter i en toppnode som har null, et, eller to barn. Hvert barn kan ha en foreldrenode. Eksemplet viser et tre som består av en toppnode med to barn. Et av barnene har selv to barn. Det kan ikke være koblinger mellom barnene.



Figur 51 – Binært tre

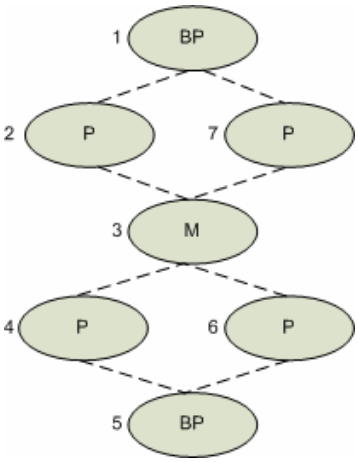
I arbeidet med å skrive en algoritme for traversering av objektmodellen ble det sett på hvilke traverseringsalgoritmer som fantes for traversering av binære trær. Den enkleste måten er å iterere seg gjennom ved bruker av en eller flere løkker, men det finnes flere teknikker for å gjøre traverseringene enklere og mer effektiv. I [W13, 5.1.12] presenteres flere potensielle rekursive algoritmer. Grunnlaget for å undersøke traverseringsalgoritmer for binære trær er en åpenbar likhet mellom strukturen i en objektmodell og et tre. Figur 53 viser objektmodellen til modellen i figur 52 strukturert på samme måte som et binært tre. Likheten er slående, selv om mange av et binært tres regler ikke fulgt [W13: 5].



Figur 52 – DiaMODL-modell brukt i traversering

Under er en liste over hvordan objektene i en objektmodell vil bli merket:

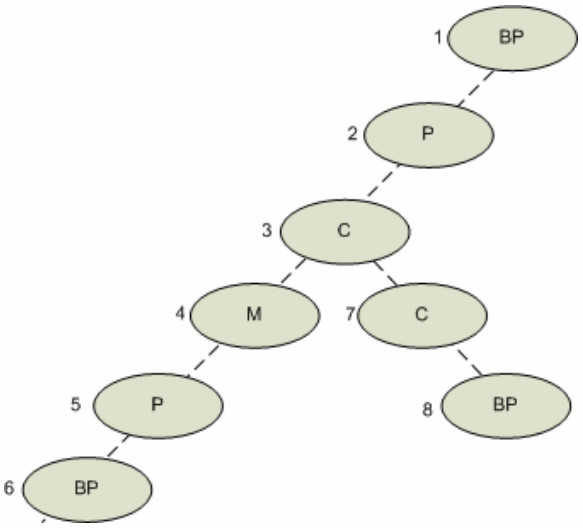
- BP: BeanPlace. Den grafiske komponenten i en interaktor.
- P: Port
- M: Modell
- C: Beregning (eng: computation)



Figur 53 – Eksempel1s objektstruktur

Startes traverseringen i en ende, det vil si i et BeanPlaceobjekt, så ser man at objektet har to barn. Dette stemmer med figur 52 hvor man ser at BeanPlacen har koblinger til to porter som sitter på interaktoren.. Fra portene går det koblinger til eller fra modellobjektet. Videre fra modellen går det igjen to koblinger til porter som tilhører modellens andre BeanPlaceobjekt. Tallene ved siden av objektene viser rekkefølgene elementene blir traversert i, noe som bestemmes av valgt traverseringsalgoritme.

Figur 54 viser objektmodellen til Eksempel2 strukturert som et binært tre. Traverseringsrekkefølgen er ikke absolutt, da den vil være avhenging av hvor i modellen traverseringen starter. Startes det en annen plass vil rekkefølgen være annerledes, men koblingene mellom objektene vil uansett være lik. I figuren starter traverseringen i BeanPlaceobjektet JTextField1. Fra den går det en kobling til en port. Deretter kommer man til en beregning. Dette stemmer med modellen og man kan se at beregningen har tre koblinger. En fortsetter til modellen, som igjen har en kobling vider til porten tilhørende BeanPlaceobjektet JSlider1. Den andre kobling som går fra beregningen går til porten foran JSlider2. Også denne figuren viser traverseringsrekkefølgen.



Figur 54 – Eksempel2s objektstruktur

Det er to hovedutfordringer som må håndteres når det skal utvikles en traverseringsalgoritme. For det første må den være effektiv, og for den andre må det være sikkert at alle noder blir besøkt. I motsetning til et binært tre som er bygd opp etter strenge regler kan modellene være av en sånn art at en vanlig rekursiv algoritme for traversering av et binært tre ikke vil fungere. Figur 54 viser en av de største forskjellene mellom et tre og modellen; En node kan i tilegg til å ha mange barn ha mange foreldre. Dette gjør at når man kommer til et element så må det sjekkes om det er besøkt fra før samtidig som man må sjekke om neste potensielle element er besøkt eller ikke. Dette gjøres ved å for hvert element sjekke om elementene i den andre enden av en kobling er besøkt. Siden en kobling er retningsbestemt kan dette gjøres både for utgående og inngående koblinger.

Kodeliste 4 viser metodene run()og traverseModel(). Metoden ligger i klassen som først blir instansiert når DiaModlGen blir kjørt. Metoden run() blir kalt av rammeverket presentert i kapittel 6. Metoden henter objektmodellen som representerer DiaMODL-modellen og legger denne i en liste. Deretter blir metoden traverseModel() kalt. Metoden er den som står for selve traverseringen, og implementerer traverseringsalgoritmen. Klassen instansierer også flere lister som blant annet brukes til å holde styre på hvilke elementer som er besøkt sammen med en egen liste for elementer som skal sendes til videre behandling. I den siste listen blir elementer lagt første gangen de blir besøkt. Støttemetodene for dette er ikke vist. Metoden run()har også variabelen "selection", som er en liste over hvilke elementer i modellen som er markert av bruker. Siden vi trenger

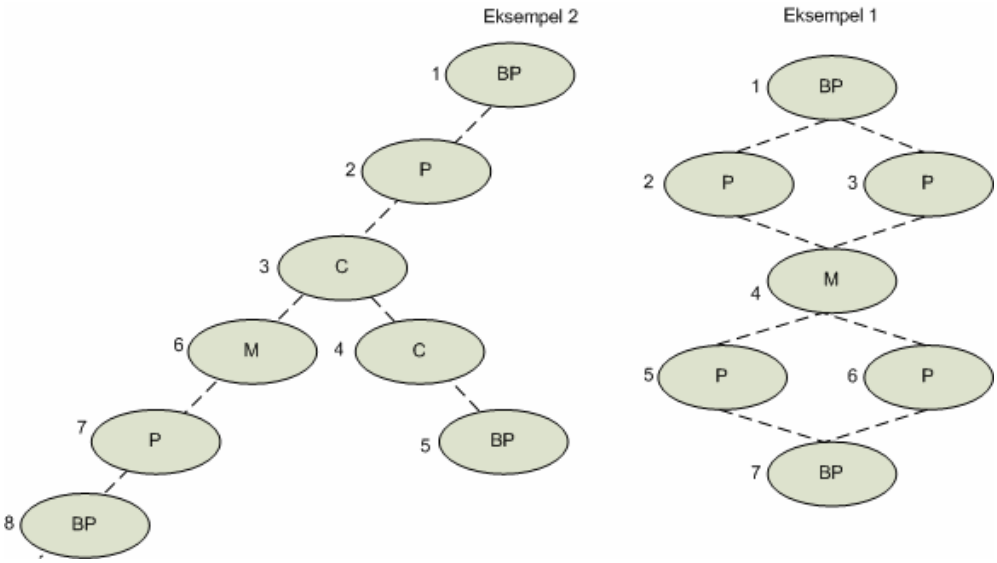
bare et element for å traversere gjennom hele modellen hentes bare det første elementet i denne listen ut som startpunkt for traverseringen.

```
public void run(IAction action) {
    ...
    List selection = ((StructuredSelection)getSelection()).toList();

    for (int i = 0; i < selection.size(); i++) {
        GraphElement element = ((GraphElementEditPart)selection.get(i))
                                                .getGraphElement();
        ModelElement model = element.getSemanticModel();
        traverseModel(model);  //Traversere modellen med "model" som startpunkt
        printElements(); //Skriver ut navnet på traverserte modellelementer
        treatElements(); //Sender traversert liste til videre behandling
        }
 }
private void traverseModel(ModelElement model){
       String name=model.getName();
       if (!isElementAdded(name)){   //Sjekker om elementer er besøkt før
           addToElements(name);      //Legger navnet i listen over besøkte elementer
           addObjs(model);           //Legger til det faktiske objektet
       }
       List out = model.getOutgoing();       //Objektets innkommende og utgående liste
       List in = model.getIncoming();
       for (int i = 0; i < out.size(); i++){ //Sjekker utgående liste

            if (out.get(i)!=null){
                Connection outCon = (Connection)out.get(i);
                ModelElement nextElem = (ModelElement)outCon.getTarget();
                if (!isElementAdded(nextElem.getName())){
                    traverseModel(nextElem); //Kaller seg selv med neste element
                }
            }
        }
        for (int i = 0; i < in.size(); i++){  //Sjekker innkommende liste
            if (in.get(i)!=null){
                Connection inCon = (Connection)in.get(i);
                ModelElement prevElem = (ModelElement)inCon.getSource();
                if (!isElementAdded(prevElem.getName())){
                    traverseModel(prevElem); //Kaller seg selv med neste element
                }
            }
        }
    }
```

Kodeliste 4 – Klassen RunAction

Parameteren som blir send til traverseModel() når denne blir kalt er det første valgte elementet i listen over modellobjekter. Det første som gjøres er å sjekke om navnet til elementet ligger i listen over besøkte elementer. Hvis det ikke er det legges det til både i listen over besøkte objektnavn og listen over faktiske objekter. Deretter hentes objektets lister over utgående og innkommende koblinger. Listene kjøres i en løkke som først sjekker utgående koblinger. For hver kobling sjekkes det om elementet i den andre enden er besøkt eller ikke. Hvis dette er tilfelle kalles traverseModel() igjen med det neste ikke-besøkte objektet. Inneholder ikke den utgående listen noen ikke-besøkte elementer gjøres det samme med listen over innkommende koblinger. Når alle elementer er besøkt avsluttes traverseringen.

Det er valget om man sjekker listen med utgående eller innkommende koblinger først som avgjør rekkefølgen i traverseringen. I og med at utgående koblinger hele tiden sjekkes først beveger traverseringen seg hele tiden dypere inn i modellen. Ikke før alle utgående koblinger er besøkt blir de innkommende koblingene besøkt. Denne metoden kalles ”dybde først” og kjennes igjen fra algoritmer og strategier for traversering av binære trær [W13]. Hadde innkommende koblinger blitt sjekket først ville det blitt en annen traverseringsrekkefølge, også kalt ”bredde først”. Figur 55 viser hvordan rekkefølgen i de presenterte traverseringene ville sett ut hvis en bredde først traversering hadde blitt brukt



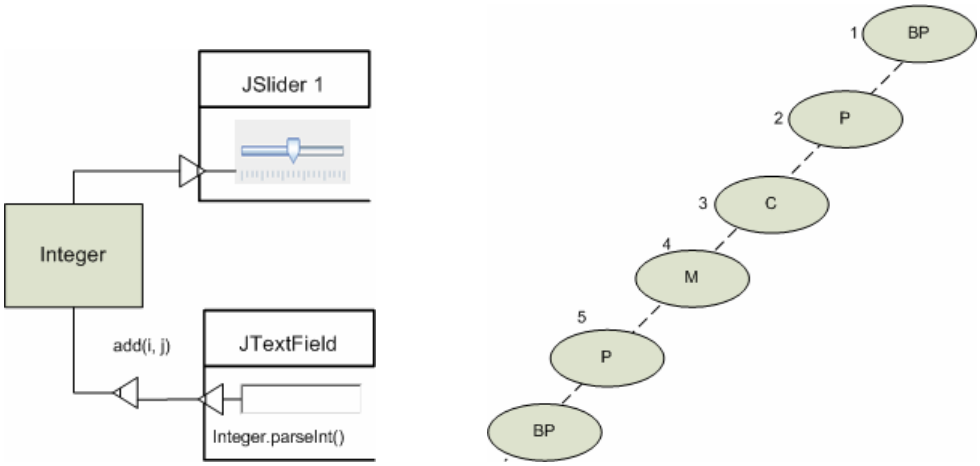
Figur 55 – ”Bredde først” traversering

I og med at alle besøkte elementer legges i en liste og sendes til videre behandling er ikke rekkefølgen på elementene i listen av betydning. Både bredde først og dybde først tilfredsstiller kravet til at alle elementer skal besøkes.

7.4.2 Verifisering av traversering og uthenting av data

Under utvikling og testing av traverseringskoden ble det laget metoder for å liste ut data om de klassene som ble traversert. Dette er ikke funksjonalitet som er brukt direkte i kodegenereringen da data ligger i objektmodellen i utgangpunktet. Det er heller ikke nødvendig å hente data på forhånd, men under arbeidet med utviklingen av traverseringskoden var det viktig å verifisere at alle objekter ble besøkt og alle data ble hentet ut. Noen av metodene for dette kunne også la seg gjenbruke i den faktiske

kodegenereringen. Figur 56 viser DiaMODL-modellen som er utgangspunktet for kodelistene vist i dette avsnittet. Figuren viser også objektmodellen, satt opp som et tre, og rekkefølgen på traverseringen er nummerert. Det øverste objektet (nr.1) er BeanPlacen, eller BeanPlace-representasjonen av interaktoren JTextField.



Figur 56 – DiaMODL-modell brukt i datautlisting

Data som i all hovedsak hentes ut er navnet på objektet, hvilken klasse som er den implementerende klassen, hvilket grafisk komponent den instansierer samt denne komponentens egenskaper. I tillegg så vises det hvilke andre objekter objektet er koblet sammen med. Hvis objektet er en beregning eller en port vil også funksjoner bli hentet ut.

Under ses data som hentes ut fra det første objektet i traverseringen. Det er et BeanPlaceobjekt med en utgående kobling til en port. Dette stemmer med modellen

```
Name of Beanplace :     Text 1
Implemented class :     class uiml.diamodl.impl.BeanPlaceImpl
SWT component: org.eclipse.SWT.widgets.Text
SWT properties:         [text!modify]
Outgoing Connections:
Outgoing target 1:    Gate1
No incoming connections!
```

Det neste elementet i listen, er en port, noe som stemmer med både modellen og den foregående interaktorens utgående kobling. Figur 43 viser at denne porten skal ha en funksjon, Integer.parseInt() for å gjøre om teksten som kommer fra interaktorens grafiske element til en annen datatype. Den faktiske funksjonskoden hentes ut sammen med datatype på funksjonens parameter, samt om det er en funksjon av typen MethodFunction eller ikke. Det siste av avgjørende for behandlingen av funksjonen i kodegeneratoren

[8.7.5]. Innkommende kobling er den foregående interaktoren og at den utgående koblinger er til beregningen.

```
Name of Component :     Gate1
Implemented class :     class uiml.diamodl.impl.GateImpl
Outgoing target 1:     Doubler
incoming source 1:     Text 1
Function is a MethodFunction
Function Source:       Integer.parseInt(String)
Function Resulttype:   int
Function Class:        class no.hal.diamodl.function.MethodFunction
```

Figuren viser at data som går ut fra porten går gjennom en beregning før den kommer til modellklassen. Utlistingen av data fra beregningen viser at innkommende kilde er porten, og utgående er modellen. Siden det er en beregning, skal den ha en funksjon. Dette er ikke en standardfunksjon men en som er skrevet i Pnuts og har navnet ”double”. Funksjonens parameter er verdien som kommer fra interaktoren.

```
Element is a Computation
Name of Component :     Doubler
Implemented class :     class uiml.diamodl.impl.ComputationImpl
Outgoing target 1:     Model
incoming source 1:     Gate1
Function is a selfwritten function
Function Source:       function double(i) {i*i}
Function Resulttype:   Object
Function Class:        class no.hal.uiml.diamodl.pnuts.PnutsFunctionFunction
```

Får beregning data fra to interaktorer vil funksjonen ha to inngående parametere. Det samme gjelder innkommende koblinger. Datautlistingen blir da som under:

```
Element is a Computation
...
incoming source 1:     Gate1
incoming source 1:     Gate2
Function is a selfwritten function
Function Source:       function add(i, j) {i*j}
...
```

Modellklassen i DiaMODL modellen har i utgangspunktet liten verdi for genereringen av kode. Dette vil bli drøftet senere, men foreløpig er modellobjektet en representasjon av en logisk dataenhet som er tilgjenglig for flere interaktorer. Under ser vi modellobjektets utgående og innkommende koblinger, samt at objektets datatype.

```
Element is a Model
Model valueType:       Integer
Name of Model :         Model
Outgoing target 1:     Gate2
incoming source 1:     Doubler
```

Den inngående porten til den lyttende interaktoren har ingen funksjoner og er ikke tatt med her. Interaktoren har ingen utgående koblinger. Det forteller kodegeneratoren at dette er en BeanPlace som bare lytter på data.

```
Name of Beanplace :     Slider 1
Implemented class :     class uiml.diamodl.impl.BeanPlaceImpl
SWT component: org.eclipse.SWT.widgets.Slider
SWT properties:         [selection]
Outgoing Connections:
No outgoing connections!
incoming source 1:     Gate2
```

Det skal også legges til at objektene som traverseres har en betydelig mengde parametere som ikke er tatt med her. I all hovedsak er dette fordi de ikke er av avgjørende betydning for kodegenereringen.

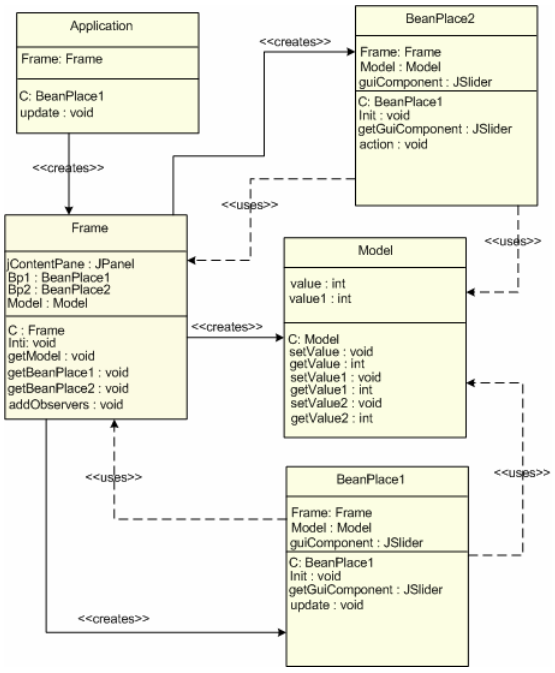
7.5 Løsningens programstruktur

Basert på objektmodellen, den presenterte traverseringskoden og DiaMODL-eksemplene vil dette kapittelet presentere hvordan genererte kode vil se for hvert av eksemplene. Denne koden vil være utgangspunkt for en utvikling av en generell metode senere i kapittelet. Den genererte koden vil inneholde klasser som relaterer direkte til objektene i objektmodellen, men også andre administrative klasser. Det vil først bli gjort rede for den generelle strukturen og de administrative klassene. Deretter vil det bli gjort rede for hvordan objektene og funksjonaliteten i objektmodellen relater til metoder og klasser i den genererte koden.

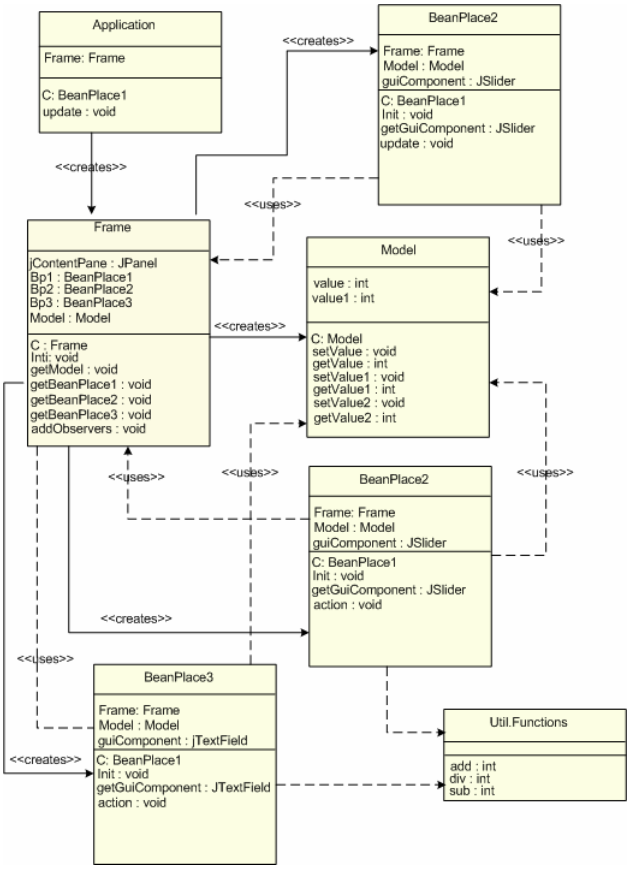
Et viktig løsningskrav er kravet om at den genererte koden skal være av en så generell at den er likt oppbygd for alle genererte eksempler. Med det menes det at det skal finnes et sett hovedkomponenter enhver løsning skal bestå av. Dette er viktig med tanke på at koden presentert i dette kapittelet skal kunne brukes i en generell metode. Klassediagrammet i figur 57 viser realiseringen av Eksempel1, og er en oversikt over de klassene som vil bli generert for en hver modellering. Applikasjonen startes av klassen Application. Denne er kun en oppstartsklasse. Klassen instansierer klassen Frame, som igjen sørger for å instansiere det grafiske rammeverket det kjørende programmet vil være en del av. Frame instansierer også de andre klassene i programmet, og som det vil bli vist håndterer den også avhengigheter mellom disse.

Modellklassen kan kjennes igjen fra eksemplene. Klassen fungerer som en lagringsplass for de andre genererte klassene. Senere vil det bli gjort rede for en vesentlig forskjell i funksjonaliteten mellom modellobjektet brukt i DiaMODL og modellklassen. De to resterende klassene, BeanPlace1 og BeanPlace2 er en representasjon av objektmodellens BeanPlaceobjekt, samtidig som den også kan tolkes som en representasjon av DiaMODL-modellens interaktor. Klassene sørger for instansiering av de grafiske komponentene samt realisering av porter og beregninger. I den genererte koden er hver instans av objektene representert som egne klasser. Klassen funksjonalitet vil bli nøye gjennomgått i et eget avsnitt.

Figur 58 viser klassediagrammet for kode generert for Eksempel2. De viktigste forskjellen er at eksemplet har tre interaktorklasser; To av disse bruker en beregning. Verken funksjonen i diagrammets ene port eller beregningen er synlige i klassediagrammet. Derimot er klassen util.Functions med. Som vist i kapittel 6, så består resultatet av en kodegenerering ikke bare av generert kode. Det er også mulig å skrive egen kode som den genererte koden kan bruke for å utvide funksjonaliteten.



Figur 57 – Klassediagram for generert kode basert på Eksempel1



Figur 58 – Klassediagram for generert kode basert på Eksempel2

I den genererte koden er det to klasser som alltid er med. De er ikke basert på objektmodellen og fungerer henholdsvis som en oppstartklasse og en administrativ klasse. Det er en klasse som blir generert lik for alle programmer. Den gjør ikke annet enn å bestemme programmets plassering på skjer og instansierer klassen Frame. Den har noe mer kode enn det som er vist her, men den er ikke relevant for det som blir gjennomgått. Kodeliste 5 viser deler av denne klassen.

```
public class Application {
 boolean packFrame = false;
 public Application() {
 Frame frame = new Frame();
…
 public static void main(String[] args) {
       new Application();
 }
```

Kodeliste 5 – Klassen Application

Klassen Frame er rammeverket til den genererte applikasjonen. For hver generering som gjøres, så blir komponentene som er med, lagt inn i denne klassen slik at de kan instansieres. Komponentene det er snakk om er da BeanPlace- eller modellklasser. Klassen styrer også kommunikasjon, avhengighet mellom klassene og dataflyt i

programmet. Kodeliste 6 viser oppbyggingen av klassen. Koden er fra genereringen av Eksempel1. De to BeanPlaceklassene er instansiert og referert til i hver sin variabel. Det samme er modellklassen.

```
public class Frame extends JFrame {
public Javax.swing.JPanel jContentPane = null;
private Model1 model1 = null;
private BeanPlace1 bp1 = null;
private BeanPlace2 bp2 = null;
private void init () {
this.setSize(400,200);
this.jContentPane = (JPanel) this.getContentPane();
jContentPane.setLayout(new GridLayout());
getModel1();
getBeanPlace1();
getBeanPlace2();
model1.addObserver(bp1);
…//Getters and Setters
```

Kodeliste 6 – Klassen Frame

Metoden Init() er av spesiell interesse. De tre første linjene i metoden er også lik for hver generering, men så kalles getModel() og getBeanPlace1() og getBeanPlace2(), noe som instansierer disse klassene. Antallet slike kall er avhengig av hvor mange interaktorer det er i modellen. Neste avsnitt vil redegjøre for ”Observer/observable” funksjonaliteten som finnes i Java. Den genererte koden bruker denne funksjonaliteten til å la klasser observere forandringer i modellklassen. Hvilke klasser som lytter bestemmes av den siste linjen i metoden init(): ” model1.addObserver(bp1);” her legges klassen bp1 til som observatør av modellen.

Klassen Frames generelle struktur er:

- Variabler som holder programmets klasser
- Initiering av det grafiske rammeverket.
- Init()-metode som instansierer disse klassene, og som setter hvilke klasser som skal observere modellen
- Aksessmetoder.

7.5.1 Implementering av koblinger

Modellklassen arver javaklassen Observable og representere data eller modellen i MVC-paradigmet. Arvingen gjør det mulig for en eller flere andre klasser å lytte på endinger i modellens parametere. Når den observerbare instansen endres kalles metoden notifyObservers() som et signal til at de lyttende klassene kan hente verdien fra modellen

[W9, oversatt]. De lyttende klassen må implementere grensesnittet (eng: interface) ”observer” som har metoden update().

Oppgavens problemstilling har et funksjonelt krav om at den genererte koden skal være oversiktlig, utvidbar og vedlikeholdbar. Det er også tilstrebet å holde forskjellige funksjonalitet atskilt. Interaktorene representerer grafiske komponenter som en bruker kan manipulere. Det innebærer en manipulasjon, som gjør at en komponents verdi forandres. Da flere interaktorer kan lytte på samme verdi bruker de i prinsippet samme modell. For at en eller flere komponenter skal kunne bruke samme verdi, uten for mye avhengigheter ble ”observer/observable” brukt [W9].

I objektmodellen er koblinger egne objekter som representerer en rettingsbestemt kobling mellom de andre objektene i modellen. I den genererte koden er det ingen klasser som representerer koblinger. Assosiasjoner mellom objektene kan ses på som koblinger, men dette må implementeres. I den genererte koden er koblingsobjekter representert på to måter. BeanPlacer, porter og beregninger er implementert i en og samme klasse, noe som fjerner behovet for assosiasjoner mellom disse. Mellom BeanPlacer og modellklasser er koblingene implementert ved hjelp av observer/observable funksjonaliteten. Lytting på en observerbar klasse kan ses på som en innkommende kobling. Tilsvarende er en utgående kobling representert ved at det skjer en oppdatering av den observerbare klassen.

7.5.2 Klassen BeanPlace

Klassen BeanPlace er den mest omfattende og mest sentrale klassen i den genererte koden. En BeanPlace er en representasjon av objektmodellens BeanPlace. Den holder en grafisk komponent og kommuniserer via porter. I tillegg så kan den også ses på som en hybrid mellom en BeanPlace og en interaktor. Bakgrunnen for dette er håndteringen av beregninger og porter. Selv om disse objektene finnes i objektmodellen, så er de ikke synlige i klassediagrammet til den genererte koden. Grunnen til det er at en port, enten den er utgående eller inngående er implementert som en metode i klassen BeanPlace. En metode for inngående klasser og en annen for utgående. Kapittel 7.2 viste hvordan en objektmodell kan flates ut. Det resulterte i at en port og en beregning fikk samme funksjonalitet; forandring av data som passerer gjennom. Dataene som endret verdi kommer fra BeanPlacens grafiske komponent, noe som gjør det naturlig at funksjonen i

porten eller beregningen som gjør dette blir lagt inn i portmetoden. Implementeringen av porter og beregninger i de genererte BeanPlaceklassene er også grunnen til at det for hvert BeanPlaceobjekt blir generert en ny klasse. Et alternativ kunne vært en BeanPlaceklasse som ble instansiert for hvert BeanPlaceobjekt. Dette er ikke naturlig av flere grunner; Først og fremst vil man måtte operere med tre forskjellige typer BeanPlaceklasser. En klasse med en inngående port, en utgående port eller både inn- og utgående. De genererte klassen vil også ha forskjellige grafiske komponenter samtidig som de kan implementere et ulikt antall beregninger. Til sammen utgjør dette bakgrunnen for at hvert BeanPlaceobjekt blir generert som en egen klasse. Forskjellen i BeanPlaceklassenes funksjonalitet gjør også at genereringskoden blir omfattende. Metodene som håndterer portene kan utvides og tillegges mer funksjonalitet hvis det trengs, og den grafiske komponenten kan være mer avansert enn hva som er tilfelle i eksemplene.

```
public class BeanPlace1 implements Observer {
...
private Functions funcs = new Functions();;
private JSlider guiComponent = null;
...
public BeanPlace1(Frame frame, Model1 model1){
this.frame = frame;
this.model1 = model1;
frame.jContentPane.add(getGuiComponent());
init();
}
```

Kodeliste 7 – Klassen BeanPlace

Kodeliste 7 viser hvordan en BeanPlaceklasse blir satt opp med en Java Swing-komponent og variabler som referanser til andre klasser. I dette tilfelle er komponenten en JSlider. Frameklassen instansierer et grafisk rammeverk samt utseende til det genererte programmet. På en måte er det ikke logisk at BeanPlacen skal håndtere en grafisk komponent. På en annen side så er det i tråd med DiaMODL-modellen. En interaktor instansierer en grafisk komponent. Denne settes funksjonelt sammen med andre komponenter. Frame fungerer som et rammeverk for dette. Vi ser i koden at den grafiske komponenten legges til rammeverket med metoden ”frame.JContentPane.add(*komponent*)”. Etter dette kalles metoden init(). Hvis klassen bare mottar data, så er init() tom. Sender den derimot fra seg data, så er det i denne klassen koden for å lytte på en komponents hendelser blir implementert. Dette er kode som er spesifikk for hver Swingkomponent og noen genereringsmotoren tar hensyn til. En komponents hendelse kaller metoden action(). Denne metoden er representasjonen av den utgående porten. På

samme måte kan metoden update() ses på som den innkommende porten. Metoden action() sørger for å hente verdien til den grafiske komponent og lagrer denne i modellklassen. Først lagrer den verdien i modellens variabel som er laget for slik at klassen selv kan bruke den. Hvis det ikke finnes beregninger eller porter lagres den samme verdien i modellens lyttbare variabel. Kodeliste 8 viser dette.

```
public void action(){
model1.setValue2(guiComponent.getValue());
model1.setValue(guiComponent.getValue);
```

Kodeliste 8 – BeanPlace forts.

En BeanPlace som tar i mot inngående data, eller lytter på modellen, reagerer på modellens notifyOberservers(), og metoden update() kjøres. Denne metoden, om det ikke eksisterer beregninger eller porter med funksjoner, setter modellens verdi på sin egen grafiske komponent. Kodeliste 9 viser en BeanPlace som lytter på modellen og den tilhørende updatemetoden. Metoden har flere betingelser som må være oppfylt for at oppdateringen skal skje. Dette er kode som håndterer funksjoner, åpning og lukking av porter og noe feilhåndtering.

```
private void init(){
guiComponent.addChangeListener(new javax.swing.event.ChangeListener(){
public void stateChanged(javax.swing.event.ChangeEvent e){
...
public void update(Observable obs, Object obj){
 if (obs == model1){
  if (!actionHappend){
    updateHappend = true;
    if (ingateopen){
      guiComponent.setValue(model1.getValue());
    }
  }
  updateHappend = false;
  }
}
… //Getters and setters
```

Kodeliste 9 – BeanPlace forts.

Klassens generelle struktur er:

- Variabler, hvis type blir satt under genereringen
- Initialisering av den grafiske komponenten
- Har interaktoren en utgående port, blir metoden init(), og action() generert.
    - Init() håndterer den grafiske komponentens hendelser
    - Action() sørger for å sette modellens og sin egen verdi.

- Har interaktoren en inngående port, genereres metoden update(). Klassen blir da registrert som en lytter på modellen. Metoden update() henter modellens verdi og setter den på klassens grafiske komponent.
- Aksessmetoder.

7.5.3 Porter og beregninger

Det har hittil blitt vist at porter og beregninger kan ses på som like i en utflatet objektmodell. Forrige avsnitt redegjorde for at objektene blir implementert som metoder i den genererte BeanPlaceklassen. Dette avsnittet vil vise konkrete eksempler på denne implementeringen. Koden generert for Eksmpel2 er utgangspunktet, og det vil bli vist metoden som representerer en inngående port, en utgående port, porter med funksjoner og bruken av beregninger. En BeanPlace sine to mulige porter er representert av metoder. Som nevnt er dette metoden action() og update(). Førstnevnte representerer utgående port, og sistnevnte inngående port. Kodeliste 10 viser dette. Metoden update() henter data fra modellen og setter denne på den grafiske komponenten. Tilsvarende ser vi at init() henter data fra den grafiske komponenten og kaller action(). Action setter verdien på modellen.

```
public void update(Observable obs, Object obj){
 if (obs == model1){
    if (ingateopen){
      guiComponent.setValue(model1.getValue());
}

private void init(){
   guiComponent.addChangeListener(new javax.swing.event.ChangeListener(){
    public void stateChanged(javax.swing.event.ChangeEvent e){
    if (outgateopen){
       action();}
   }

public void action(){
       model1.setValue2(guiComponent.getValue());
       model1.setValue(guiComponent.getValue);
}
```

Kodeliste 10 – Metoder som representerer porter i en BeanPlaceklasse

Hvis en beregning eller en port med en funksjon ligger mellom BeanPlacen og beregningen implementeres dette i update() og action() metodene. Funksjoner bra beregninger og funksjoner fra porter behandles likt, noe som er i samsvar med den utflatede objektmodellen. Hva funksjoner i seg selv angår kan det skilles mellom to typer, nemlig standard javafunksjoner og egenkonstruerte funksjoner. Standardfunksjoner er funksjoner, eller metoder som finnes i Java og kan brukes direkte. Egenlagede funksjoner

er metode utvikler selv skriver. Disse er implementert i den ikke-genererte klassen util.functions, og kalles fra den genererte koden.

Standard javafunksjoner

Eksemplet som er brukt her er Integer.parseInt(), som er en metode som konverter tall representert som tekst til datatypen Integer, Dette er nødvendig da den grafiske komponenten JTextField i Eksempel2 sin datatype er tekst. For å kunne sette denne verdien i modellen som skal ha en Integer, må tallet, for eksempel 23 konverteres fra å være teksten 23 til tallet 23 [W10]. Som vist støtter DiaMODL-editoren at man setter denne typen funksjoner i modellen. Kodeliste 11 viser hvordan funksjonen blir lagt rundt kallet som setter verdien på modellen. Funksjonen er den samme som er satt som parameter i den utgående porten og hentet under traverseringen.

```
public void action(){
model1.setValue1(Integer.parseInt(guiComponent.getText()));
...
}
```

Kodeliste 11 – Eksempel på generering av standard javafunksjon

Egenkonstruerte funksjoner

Den andre typen funksjoner DiaMODL-editoren støtter skriptspråket Pnuts [W11]. Dette kapittelet vil redegjøre for valget om å ikke inkludere Pnuts i den genererte koden.

Med egendefinerte funksjoner menes funksjoner utvikleren selv lager. Disse skrives inn i porter eller beregninger på samme måte som standard javafunksjoner, men er altså implementert i en egen klasse.

```
package utils;
...
public final class Functions {
    public int add(int i, int j){
        return i+j;
    }

    public int double(int i){
        return i+i;
    }
}
```

Kodeliste 12 – Eksempel på generering av egenkonstruerte funksjoner

Kodeliste 12 viser et eksempel på en egen laget funksjon kalt add(); Eksempel2 viser en beregning som har koblinger inn fra to interaktorer. Dette viser at data fra to interaktorer på en eller annen måte skal kombineres i en funksjon, for så å sendes til modellen.

Hvilken funksjon det er kan utvikler selv velge. For eksempel kan det være et valg av høyeste verdi eller en sammenslåing av verdiene som i dette tilfellet. Datatypene trenger ikke å være av samme type da dette kan håndteres i metoden utvikler skriver, men det er i eksemplet valgt å konvertere tekst i den ene interaktoren til tall slik at begge inngående parametere til funksjonen er like. Utfordringen var i utgangspunket hvordan koden som ligger i en beregning utenfor interaktoren skal håndteres. Siden det er bestemt at porter og funksjoner skal håndteres likt, blir også funksjoner i beregninger lagt i action()-metoden. Dette er naturlig det da det er data på vei ut av porten som er brukt i beregningen. Her kommer også modellklassens ekstra variabler inn. Action() metoden setter verdien på sin egen komponent i modellen før koden som setter verdien på modellen kjøres. Dette gjør det også mulig å hente verdien til en annen interaktor fra modellen, og det er denne verden som brukes i kallet til funksjonen som en parameter i de tilfelle hvor to interaktorer er involvert. Kodeliste 13 viser en implementasjon av dette. Først settes interaktorens egen verdi, noe som også blir gjort i de andre interaktorene. Deretter settes verdien på modellen gjennom et kall til den selvlagede funksjonen. Parametrene til funksjonen er interaktorens egen verdi samt verdien fra den andre interaktoren. Denne verdien hentes fra modellklassen.

```
public void action(){
       model.SetValue1(guiComponent.getValue());
model.setValue(funcs.add(guiComponent.getValue(), model.getValue2()));
}
```

Kodeliste 13 – Bruk av egenkonstruerte funksjoner

Hvis en ekstern funksjon bare har en parameter, bestemt av at den bare har en innkommende kobling fra en interaktor, håndteres dette på samme måte, men da er det kun interaktorens egen verdi som brukes. Kodeliste 14 viser dette.

```
public void action(){
model.SetValue1(guiComponent.getValue());
model.setValue(funcs.double(guiComponent.getValue()));
}
```

Kodeliste 14 – Funksjon med kun en parameter

Passerer data fra en interaktor gjennom både en port med funksjonalitet og en beregning kombineres dette som vist i kodeliste 15.

```
public void action(){
model.SetValue1(guiComponent.getValue());

model.setValue(funcs.double(Integer.parseInt(guiComponent.getValue())));
```

Kodeliste 15 – Kombinering av standard- og egenlagede funksjoner

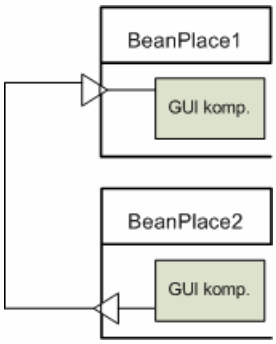
Implementasjonen av porter og beregninger fungerer fint for de spesielle tilfellene, og vil også være metoden som vil bli forsøkt brukt i den generelle metoden. En utfordring er nøsting av funksjoner. Det vil si når en beregnings resultat er parameter i en ny beregning. Fjerner man modellobjektet vil det også være en utfordring å bestemme om en funksjon skal implementeres i den inngående eller den utgående porten.

7.5.4 Klassen og begrepet "Modell"

I oppgavens eksempler har DiaMODL-modellene blitt vist med et modellobjekt brukt til å ta vare på data. Noen BeanPlaceklasser sender data til dette objektet mens andre henter. Modellbegrepet har blitt tatt med i den genererte koden. Dette avsnittet vil gjøre rede for bruken av modeller i den genererte koden, og forskjellen på den genererte modellen kontra bruken i DiaMODL.

Det vil også innledende bli drøftet hva som skjer hvis man fjerner modellobjektet fra DiaMODL-modellen. I en generalisert metode må dette være mulig. Den genererte koden vil fortsatt trenge en datarepresentasjon. Eksemplenes modellklasse er generert ut fra modellen-representasjonen i DiaMODL-modellen. For en generell metode kan det være nødvendig å basere modellklassen på andre kriterier.

I objektmodellen, eller en visuell DiaMODL-modell, er et modellobjekt brukt til å vise at flere interaktorer kan hente data fra et objekt satt til å holde en gitt datatype. Figur 59 viser en utgave av Eksempel1 uten modellobjektet. Funksjonaliteten er fortsatt den samme. Data sendes fra en interaktor til en annen. I den genererte koden har modellen to funksjoner. Den første er å publisere en verdi satt av en BeanPlace som sender fra seg data. Dette sammenfaller med modellobjektes funksjon. Modellens andre funksjon er at den blir brukt som lager for verdier til BeanPlacer som deltar i en beregning. Dette gjør at enhver BeanPlace sin verdi er tilgjengelig for andre BeanPlacer uten at de trenger å kalle BeanPlacen direkte.



Figur 59 – Eksempel1 uten modellobjekt

Fjerner man modellobjektet oppstår det to problemer som må håndteres i forbindelse med en kodegenerering. Dette er et viktig moment når det skal utarbeides regler for en generell metode. Den genererte koden vil fortsatt bruke en modellrepresentasjon da denne har en utvidet funksjonalitet. Men hva skal bestemme hvilke objekter som lytter på en modell, og hva er kriteriene for at en modellklasse skal opprettes? Den andre utfordringen grunner i bruken av beregninger. Hvis man i figuren setter en beregning på koblingen mellom de to interaktorene; Hva er det som bestemmer om funksjonen til beregningen skal ligge i den ene BeanPlacens inngående metode eller den andre BeanPlacens utgående? Med et modellobjekt hentet fra DiaMODL-modellen er ikke dette noe problem da den utgjør et naturlig skille.

7.5.5 SWT versus Swing

I [6.4.3] ble det redegjort for forskjellen på SWT og Swing samt hvordan DiaMODL bruker SWT-komponenter i modelleringen. Ved valg av metode for kodegenerering var det viktig å gjøre genererte koden så standardisert som mulig. Dette innebærer at de grafiske komponenten i de ferdige programmene må være fra Swing-biblioteket [W7]. Det finnes ingen metode for å konvertere SWT til Swing da disse er to forskjellige rammeverk. Samtidig er rammeverkene så like at det er mulig å se på parametrene til en SWT-komponent og bestemme hvilke tilsvarende parametere som ville være brukt på en Swing-komponent for at den skulle fungert likt. Dette er utnyttet i kodegenereringen. Kodeliste 9 viser klassen Mapper. Hovedfunksjonaliteten til klassen er å støtte en konvertering fra SWT til Swing. Det fungerer på det viset at hvis man vet hvilken SWT-komponent man har, så vet vi også hvilken SWT-komponent som skal brukes. Informasjon om SWT-komponenten sendes til mapperklassen, og man får informasjon om den tilsvarende swingklassen tilbake.

En Swing-komponents attributter er kjent, inkludert mulige hendelser, (eng: Events), lytter (eng: Listener) og datatype (som komponenten returnerer eller skal ha inn). En komponent kan ha flere typer hendelser men dette kan i de aller fleste tilfellene også leses ut fra SWT-komponent og oversettes. Den første metoden i klassen får inn navnet på en SWT-komponent og returnerer en liste over tilsvarende Swing-komponents egenskaper. Dette er en liste som kan utvides etter hvert som man trenger andre komponenter enn det som er vist her. Teksten i den returnerte listen brukes direkte i kodegenereringen.

```
public class Mapper {
        ...//Instansvariabler
        private int valCount = 1; //Brukt til nummerering av komponenter
        public Mapper(){ //Instansierer med potensielle komponenter
                slider[0] = "JSlider";
                slider[1] = ".addChangeListener(new javax.swing.event.ChangeListener()";
                slider[2] = "stateChanged(javax.swing.event.ChangeEvent e)";
                slider[3] = "int";
                slider[4] = "Value";
                text[0] = "JTextField";
                text[1] = ".addActionListener(new java.awt.event.ActionListener()";
                text[2] = "actionPerformed(java.awt.event.ActionEvent e)";
                text[3] = "String";
                text[4] = "Text";
        }
        //Sender tilbake Swing spesifikk funksjonalitet for tilsvarende SWT-komponent
        public String[] getGuiMapping(String SWTComponent) {
                String[] retval = new String[5];
                if(SWTComponent.equals("org.eclipse.SWT.widgets.Slider")) {
                        retval = slider;
                }
                else if(SWTComponent.equals("org.eclipse.SWT.widgets.Text")) {
                        retval = text;
                }return retval;
         }

        //Verdier til bruk I generering av funksjoner
         public String[] getAddDataType(String componentDataType){
        ...
        }
        //Henter neste mulige verdi til navngiving av komponenter
        public int getNextAddValue(){
            int retval = valCount; valCount++; return retval;
        }
}
```

Kodeliste 16 – Klassen Mapper

Kodelsite 16 viser mapperklassen. I praksis fungerer den som en tabell for oversettelse. I tillegg har den også en del andre metoder som ikke har direkte sammenheng med disse grafiske komponentene Dette er metoder som generatoren bruker på andre klasser som er avhengig av å vite hvordan den grafiske komponentens datatype skal behandles.

7.5.6 DiaModlGen StateChart støtte

I [5.3] beskrives et behov for å ha metoder som kan sette porter i en interaktor åpne eller lukket basert på tilstandsfunksjonalitet i DiaMODL-modellen. En slik funksjon er også et

viktig løsingskrav, og er implementert i de genererte BeanPlaceklassene. I alle genererte klasser genereres de boolske variablene ”ingateopen” og ”outgateopen”. Tilhørende disse metodene genereres aksessmetoder for å sette variablene til sanne eller usanne (eng: true or false). Kodeliste 17 viser hvordan disse variablene brukes. Update() representer porten inn til interaktoren. Er variabelen ”ingateopen” satt til usann, så kjøres ikke metoden som henter verdien fra modellen. Tilsvarende kjøres ikke action() metoden hvis en utgående port er satt til lukket.

```
private boolean outgateopen = true;
private booolean ingateopen = true;
…
public void update(Observable obs, Object obj){
if (obs == model1){
if (ingateopen){
guiComponent.setValue(model1.getValue());
}
...
private void init(){
   guiComponent.addChangeListener(new javax.swing.event.ChangeListener(){
    public void stateChanged(javax.swing.event.ChangeEvent e){
    if (outgateopen){
       action();}
   }
...
public void setOutGateOpen(boolean open){
 outgateopen = open;
}
```

Kodeliste 17 – Statechartstøtte

7.6 Genereringslogikk

Dette avsnittet vil presentere klassene som logisk bestemmer hva som skal genereres. Genereringen skjer basert på objektene funnet i objektmodellen, og realiseres ved hjelp av utskriftsklassen vist i forrige avsnitt. Koden som er vist er laget med tanke på at funksjonen til hvert element i en DiaMODL-modell skal kunne realiseres i den genererte koden

Kodeliste 18 viser klassen Writer. Klassen blir instansiert av genereringsklassen og sendt inn som parameter til klassehierarkiet som generer den faktiske koden. Den mottar et filnavn og en liste med alle tekstvariablene som har blitt generert. Denne listen traverseres gjennom og filene blir skrevet til fil linje for line.

```
public class Writer {
        private String fileExtension = ".java";
        public void print(ArrayList in, String fileName)throws IOException{
                this.fileName = fileName; this.text = in;
                File minFil = new File("../gen/"+fileName+fileExtension);
```

```
            BufferedWriter out = new BufferedWriter(new FileWriter(minFil));

        //Skriver til fil
        for(Iterator iter = text.iterator(); iter.hasNext();) {
            String e = (String)iter.next();
            if (e == "newLine"){
                out.newLine();
            }else {
                out.write(e);
            }
        }
        out.close();
```

Kodeliste 18 – Klassen Writer

GenWrapper er klassen som fungerer som et grensesnitt til klassene som tar seg av selve kodegenereringen. Grensesnittet utgjøres av klassens metoder som ved behov instansierer en eller flere av genereringsklassen etter hvert som de trengs. Mange av genereringsklassenes parametere settes ved instansiering men dette er ikke alltid nok for å kunne bestemme all funksjonalitet. De instansierte objektene returneres derfor til klassen CodeGenerator, hvor resten av variablene blir satt mens det blir traversert gjennom objektene hentet fra objektmodellen. CodeGenerator kaller grensesnittet hver gang det skal opprette en ny klasse. Kodeliste 19 viser deler av klassen GenWrapper.

```
       //Klassenavn for generering
       private final static String APP_CLASS_NAME          = "Application";
       private final static String FRAME_CLASS_NAME        = "Frame";
       ...
       private String currentBPClassName    = "";

//Oppretter klassen Starter
 public void addStarter(){
    StarterData starter = new StarterData(writer, APP_CLASS_NAME, FRAME_CLASS_NAME);
              classes.add(starter);
 }
//Oppretter klassen Model og returnere den
public ModelData addModel(String dataType){
ModelData model = new ModelData(writer, getModelClassName(), getModelName(),  dataType);
       classes.add(model);
       return model;
}
//Oppretter klassen Frame og returnere den
public FrameData addFrame(){
       FrameData frm = new FrameData(writer, FRAME_CLASS_NAME);
       classes.add(frm);
       return frm;
}
//Oppretter klassen BeanPlace, bruker data hentet fra Mapperklassen
public BeanData addBeanPlace(String modelDataType, String component, int classType){
       BeanData btn = new BeanData(writer, getBPClassName(), getBPName(), modelDataType,
       FRAME_CLASS_NAME+" "+FRAME_NAME...);
          btn.setClassType(classType);
          guiMapping = mapper.getGuiMapping(GUIcomponent);
          btn.setGUIComponentName(guiMapping[0]);
          btn.setGUIComponentListner(guiMapping[1]);
          btn.setGUIComponentEvent(guiMapping[2]);
          btn.setGUIComponentValue(guiMapping[3]);
          btn.setGUIComponentMethod(guiMapping[4]);

          btn.setComponentDataType(guiMapping[3]);
       classes.add(btn);
       return btn;
```

Kodeliste 19 – Klassen GenWrapper

Den siste, og viktigste klassen er klassen CodeGenerator. Den får inn en liste med traverserte objekter, og bruker den foregående klassen som et grensesnitt slik at parametere i de genererende klassene kan settes. Klassen har all genereringslogikk og er hjertet i applikasjonen. Kodeliste 20 viser de viktigste metodene i klassen. I tillegg har klassen en rekke støttemetoder som hovedmetodene bruker. Disse er utelatt av plasshensyn med unntak av to metoder som blir gjort rede for i neste avsnitt.

```
//Generer klasser og nullstiller etter endt generering
Private void printData();
    generateStarter();
    ModelData mymodel = generateModel();
    FrameData myframe = generateFrame(mymodel);
    generateBeanPlace(myframe, mymodel);
    gen.printClasses();
    clearClasses();
 }

 //Genereringsmetode for Modellklassen
 private ModelData generateModel(){
     String modelDatatype = "int";
     for(Iterator iter = models.iterator(); iter.hasNext();) {
         BeanPlace model = (BeanPlace)iter.next();
         modelDatatype = mapper.getAddDataType(model.getValueType().toString())[0];
     }
     ModelData mymodel = gen.addModel(modelDatatype);
     return mymodel;
 }

 //Genereringsmetode for Frameklassen
 private FrameData generateFrame(ModelData mymodel){
     FrameData myframe = gen.addFrame();
     myframe.addModelClass(mymodel.getFileClassName(), mymodel.getClassName());
     return myframe;
 }

//Genereringsmetode for BeanPlace
 private void generateBeanPlace(FrameData myframe, ModelData mymodel){
     for(Iterator iter = beanplaces.iterator(); iter.hasNext();) {
         BeanPlace bp = (BeanPlace)iter.next();
         BeanData mybean = gen.addBeanPlace(mymodel.getDataType(),
         bp.getValueType().toString(), bpClassType(bp));

 //Sjekker typen BeanPlace
         if (bpClassType(bp) == 1 || bpClassType(bp) == 2){
             int addValue = mapper.getNextAddValue();
             mybean.setUseAdditionalValue("Value"+addValue);
             result = mapper.getAddDataType(mybean.getComponentDataType());
             mymodel.addAdditionalValue(result[0], result[1]);
             String[] functionData = bpOutgoingFunction(bp);
                 if (!functionData[0].equals("")){
                     mybean.setUseFunc(true);

                     //Legger til eventuelle egne funksjoner
                     mybean.setFunction(functionData[0]);
                     if (functionData[1].equals("true")){
                         mybean.setFunctionParams(true);
                         if (isFunctionVisited) {
                             mybean.setFunctionValue("Value"+(addValue-1));
                         } else {
                          mybean.setFunctionValue("Value"+(addValue+1));
                          isFunctionVisited = true;}
                  } else {
                      mybean.setFunctionParams(false);}
                 }
```

```
                Legger til eventuelle standardfunksjoner
                String methodFunction = bpOutgoingMethodFunction(bp);
                if (!methodFunction.equals("")){
                    mybean.setUseMethodFunc(true);
                    mybean.setMethodFunction(stripMethodFunction(methodFunction));
                }
            }

        //Adding bean to Frameklassen
    myframe.addBeanPlace(mybean.getFileClassName(), mybean.getClassName(),
    mymodel.getFileClassName(), mymodel.getClassName());
            if (bpClassType(bp) == 0 || bpClassType(bp) == 2){
                myframe.addObservers(gen.getCurrentModelName(), mybean.getClassName());
            }
        }
    }
```

Kodeliste 20 – Klassen CodeGenerator

CodeGenerator mottar en liste med traverserte modellobjekter. Deretter deles de forskjellige typene klasser opp og legges i hver sin liste. På små modeller med en modellklasse er ikke dette nødvendig, men ved større og mer komplekse modeller, hvor en modell er delt i flere mindre delere kan en oppdeling av objektene gjøre det mulig å styre i hvilken rekkefølge genereringen skjer. Det er også verd å merke seg at dette gjør kodegenereringen avhengig av et modellobjekt i DiaMODL-modellen. Som vist er ikke dette objektet nødvendig for at en DiaMODL-modell skal være komplett. Innføringen av en generell metode vil endre denne delen av kodegenereringen. Det samme gjelder rekkefølgen objektene blir generert i. Avhengigheten mellom genereringsklassen følger avhengigheten til de endelig genererte klassene. Skal man generere en BeanPlaceklasse så må både en frameklasse og en modellklasse tilgjengelig. BeanPlaceklassen bruker metoder i disse klassene for å sette variabler som styre funksjonalitet. På samme måte må frameklassen vite om modellklassen. Dette styrer rekkefølgen på metodekall i klassen. Metoden printData()kaller først metoden for å generere en starterklasse. Denne er ikke dynamisk og trenger ingen parametere. Deretter kalles metoden generateModel(). Metoden instansierer en klasse som brukes til å generere en modellklasse og setter datatypen brukt i modellklassen. Ved det påfølgende kallet til generateFrame() sendes modellgeneratoren inn som parameter. GenerateFrame() oppretter en genereringsklasse for frameklassen og legger inn en referanse til modellklassen.

Den genererte BeanPlaceklassen refererer til og bruker både modellklassen og frameklassen. Dette kan kjennes igjen i metoden generateBeanPlace(). Den instansierer en genereringsklasse for en BeanPlace og bruker klassen Mapper til å sette riktige verdier for den grafiske komponenten. Deretter blir det avgjort hvilken type BeanPlace det er snakk om. Det opereres med tre typer: En som har kun har inngående port, en som har

utgående port og en klasse som har både inngående og utgående porter. Som vist har dette betydning for hvilke metoder som skal genereres. I tillegg kalles metoder som sjekker om BeanPlacen bruker funksjoner, enten i porten eller frittstående funksjoner som ligger utenfor. Når alle klasser er ferdig generert kalles metoden printClasses() som sørger for at skrivingen til fil blir gjort. For at det skal være mulig å generere modellen på nytt igjen uten å starte programmet på nytt nullstilles alle lister og generatorklasser.

7.6.1 Funksjoner i porter og beregninger

I kapittel 5.5 beskrives det hvordan DiaMODL implementerer funksjoner. Forrige avsnitt viste hvordan metoder for å sette disse funksjonene inn i kodegenereringen blir brukt. Dette avsnittet vil vise de faktiske metodene som håndterer funksjoner. Kodeliste 21 viser både metoden som kalles for å finne eventuelle standardfunksjoner på porter og metoden som finner egenlagede funksjoner. Den førstnevnte funksjonen vil være lettere å håndtere da man vet at den bare vil ha en funksjonsparameter. Metoden starter med å sjekke om den utgående porten har en standardfunksjon. Dette gjøres med metoden extractFunctionSource(). Hvis porten eller beregningen har en funksjon vil den faktiske funksjonen returneres som en tekststreng. Mangler dette vil det blir returnert en tom tekst og resultatet er at ingen funksjoner blir generert.

```
private String bpOutgoingMethodFunction(ModelElement bp){
        List outgoingList = bp.getOutgoing();
        Connection outgoing = (Connection)outgoingList.get(0);
        Gate gate = (Gate)outgoing.getTarget();
        String returnValue = extractFunctionSource(gate);
           if (returnValue.equals("Object")){
               return "";
           } else{
               return returnValue;
           }
    }

    private String[] bpOutgoingFunction(ModelElement bp){
        String[] returnValues = {"",  ""};
        List bpOut;
        List gateOut;
        boolean hasComputationTwoParams = true;

        bpOut = bp.getOutgoing();
        Connection bpCon = (Connection)bpOut.get(0);
        Gate mygate = (Gate)bpCon.getTarget();
        gateOut = mygate.getOutgoing();
        Connection gateCon = (Connection)gateOut.get(0);
        ModelElement myComputation = (ModelElement)gateCon.getTarget();

        if (myComputation instanceof Computation){
            Computation comp = (Computation)myComputation;
            hasComputationTwoParams = isTwoFunctionParameters(comp);
            returnValues[0] =  stripFunction(extractFunctionSource(comp));

            if (hasComputationTwoParams){
```

```
            returnValues[1] = "true";
        } else {
            returnValues[1] = "false";
        }
    }
    return returnValues;
```

Kodeliste 21 – Metoder for behandling av funksjoner

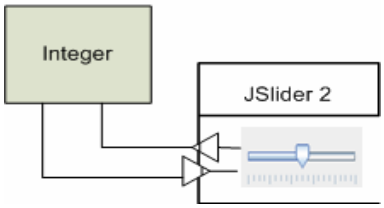
Hvis det derimot eksisterer en funksjon returnerer metoden for teksten som representerer selve funksjonen. Metoden som sjekker egenlagede funksjoner bruker i tillegg metoder som sjekker antallet parametere funksjonen skal ha. Skal funksjonen ha to eller flere parametere så vet man at de andre verdiene kommer fra andre BeanPlaceklasser. Her er det programmert inn en begrensning som direkte følge av at eksemplene er så enkle som de er. Det sjekkes kun på om det er en eller to BeanPlacer som er deltagende i beregningen.

7.6.2 Pnuts versus egenlagede funksjoner

I og med at dagens DiaMODL-editor ikke støtter bruken av egenlagede funksjoner i modellen var alternativet enten å implementere støtte for Pnuts i kodegenereringen eller å utvide DiaMODL modellen til å støtte egenlagede funksjoner. Det første alternativet ville skapt avhengigheter mellom den genererte koden og et tredjeparts bibliotek. Noe som ikke var ønskelig. En utvidelse av DiaMODL-editoren var heller ikke et reelt alternativ. Løsningen ble å tilpasse den pnutskoden man skrev inn i modellen på en måte som gjør det mulig å kjenne igjen et funksjonsnavn. En støttemetode for å hente ut dette funksjonsnavnet ble lagt til i genereringskoden.

Eksempel2 viser en beregning som legger sammen summen som kommer fra to interaktorer. I en DiaMODL-modell ville denne vært implementert som pnutsfunksjonen "function add(i, j) {i+j}". Støttemetoden i genereringen henter ut navnet på pnutsfunksjonen som i dette tilfellet er add(). Siden utvikler vet navnet på funksjonen som blir hentet ut kan han lage en egen metode i funksjonsklassen med samme navn, og på den måten omgå begrensningen.

7.6.3 Konkurerende oppdatering



Figur 60 – DiaMODL-modell med konkurerende oppdateringt

I den genererte koden er det frameklassen som bestemmer hvilke klasser som lytter på modellen. Klassene som blir registrert som lyttere må fylle kravet til en observerklasse ved å ha en oppdateringsmetode som kalles når modellen endres. Hvis en BeanPlaceklasse lytter på modellen og samtidig oppdaterer den vil det oppstå en ufordring i form av en uendelig løkke; Når klassen oppdaterer modellen vil den får beskjed om at modellen er oppdatert og henter den oppdaterte verdien fra modellen. Siden en ny verdi blir satt på den grafiske komponenten vil modellen igjen bli oppdatert. Figur 60 viser en DiaMODL-modell hvor dette er tilfelle. Oppdateringen vil pågå til verdien når en eventuell grense. For å unngå at dette skjer må BeanPlaceklassen implementere logikk som sjekker om den verdien som settes på den grafiske komponenten er et resultat av en brukerinteraksjon eller en modelloppdatering. Kodeliste 22 viser dette.

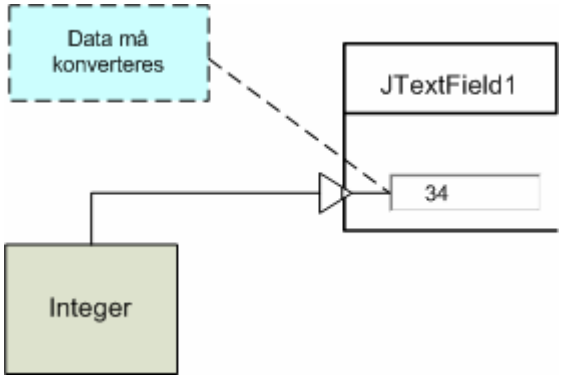
```
public class BeanPlace1 implements Observer {
...
private boolean actionHappend = false;

private boolean updateHappend = false;
...
public void action(){
model.setValue1(guiComponent.getValue());
if (!updateHappend){
actionHappend = true;
model.setValue(guiComponent.getValue());
}
actionHappend = false;
}
...
public void update(Observable obs, Object obj){
if (obs == model){
        if (ingateopen){
 if (!actionHappend){
          updateHappend = true;
           guiComponent.setValue(model.getValue());
 }
        }
updateHappend = false;
}
}
```

Kodeliste 22 – Sjekk på konkurrerende oppdatering

Variablene ”actionHappend" og ”updateHappend” er boolske variabler som i utgangspunkter er satt til å være usanne. Hvis modellen blir oppdatert kalles metoden update() i BeanPlaceklassen. Metoden setter updateHappend til å være sann; Det har skjedd en oppdatering og dette blir flagget. Siden den grafiske komponenten ble oppdatert, vil metoden action() prøve å oppdatere denne modellen med verdien til den oppdaterte grafiske komponenten. Den sjekker updateHappend for å se om det er en reell oppdatering eller ikke. Variablene som nå er sann viser at det ikke er tilfelle og action() fullføres ikke. Den samme funksjonaliteten blir også brukt til å hindre updatemetoden i å kjøre hvis en reell oppdatering skjer. Dette er kode som blir generert for alle BeanPlaceklasser, og er av funksjonell art.

7.6.4 Funksjoner på inngående porter

Foreløpig har det bare blitt vist kode som bruker funksjoner i forbindelse med utgående porter og beregninger. Dette har vært både egenkonstruerte og standardfunksjoner. Dette avsnittet vil vise den samme funksjonaliteten for inngående porter og beregninger.



Figur 61 – Konvertering av datatype i inngående port

Kodeliste 23 viser deler av koden til BeanPlaceklassen i figur 61. Klassens grafiske komponent er en JTextField. I dette tilfelle holder modellobjektet en verdi av typen integrer. Siden den grafiske komponenten skal vise frem datatypen tekst må tallet i modellen forandres fra tall til tekst før det blir sendt til visning. Dette typen konvertering er det laget støtte for i mapperklassen. Der blir det sjekket på differanser mellom datatypen til modell og BeanPlace, og riktig konverteringsmetode blir returnert.

```
public void update(Observable obs, Object obj){
if (obs == model){
if (!actionHappend){
```

```
updateHappend = true;
...
guiComponent.setText(""+model.getValue());
}
updateHappend = false;
}
```

Kodeliste 23 – Standard javafunksjon på inngående port

Kodeliste 24 viser hvordan dette blir hvis datatypene er bytte om. Her er holder modellen en tekst, mens interaktoren skal ha inn en tallverdi. Her brukes standardfunksjonen for å konvertere tekst til tall, på samme måte som funksjoner på utgående porter. Dette gjelder også for egenlaget funksjoner brukt i beregninger.

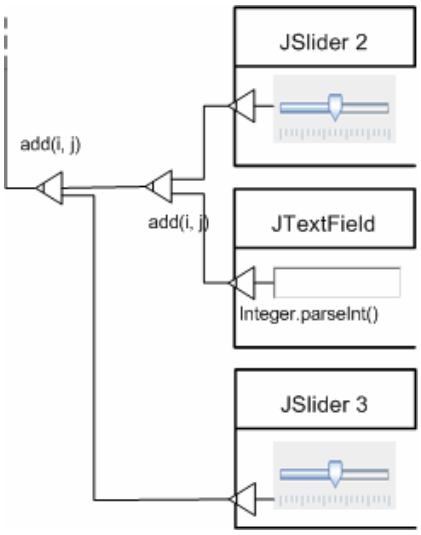
```
public void update(Observable obs, Object obj){
...
guiComponent.setValue(Integer.parseInt(model.getValue()));
...
```

Kodeliste 24 – Standard javafunksjon på inngående port forts.

7.6.5 Nøsting av beregninger

Med nøsting mens at en funksjon bruker verdien fra en annen funksjon som inngående parameter. Dette avsnittet vil vise at det er flere alternative løsninger på hvordan nøsting av funksjoner vil kunne implementeres. Figur 62 viser den aktuelle problemstillingen. Verdien som sendes videre er summen av utgående data fra tre interaktorer.

Modellen viser at modellklassens verdi skal være en kombinasjon av verdien i BeanPlacen ”JSlider3” sammen med summen av verdien til ”JSlider2” og ”JTextField”. Hvis vi tar utgangspunkt i JSlider2 og JTextField, altså de to BeanPlacen som bruker den innerste funksjonen, så vil det naturlige være å la de gjøre ting på samme måte som før. Det vil si at siden de er uvitende om den ytterste funksjonen skal de sette verdien på modellen til summen av seg selv og den andre medvirkende BeanPlacen. Dette vil ikke stemme med DiaMODL-modellen. Verdien av de to innerste interaktorene kan ikke bli satt på modellen da den verdien bare en er parameter i den ytterste funksjonen. Det man vet er at det første en BeanPlace gjør er å sette sin egen verdi i en variabel laget for akkurat det i modellen. Det gjør at vi til enhver tid kan få tak i alle BeanPlacer sin verdi, noe som gir to mulig måter nøstingen kan løses på.



Figur 62 – Nøsting av beregninger

Den første måten er å bruke en modellklasse for hver av beregningene. Det vil si at den innerste beregningen setter en verdi i en egen modellklasse. Det vil gjøre at koden for de to innerste interaktorene vil være lik koden for bruk av en enkel beregning. Dette vil derimot ikke være tilfelle for koden som bruker den ytterste beregningen, da denne må ha funksjonalitet å for å hente verdi fra en modell som er innført på grunn av nøstingen i seg selv. På bakgrunn av eksemplene som er brukt er ikke slik funksjonalitet innført men vil være et viktig moment i forbindelse med en generell genereringsmetodikk.

Den andre mulige metoden gjør at genereringslogikken vil måtte gjøre en utvidet sjekk, men koden i alle tre interaktorene vil bli lik. Koden som er laget for eksemplene sjekker om en interaktor har forbindelse med en beregning, og henter data for den andre interaktoren som tar del i denne beregningen. En nøsting vil gjøre at det spesifikt må sjekkes på om det er en ny beregning utenfor beregningen. Dette vil allikevel være en naturlig utvidelse. Den genererte koden som implementerer dette vil sette en verdi på modellen. Dette gjelder alle interaktorene. Forskjellen fra en enkel beregning er at hver enkelt interaktor vil måtte kaller to funksjoner i stedet for en, og nøster disse kallene på samme måte som eksisterende beregningene og porter er nøstet.

Dette kapittelet har vist hvordan hvert enkelt modellelement i en utflatet objektmodell har en spesifikk funksjon. Denne funksjonaliteten er realisert i generert kode, og det er vist hvordan den genererende koden gjør denne jobben. Begrensningen ligger i at den presenterte kodegenereringen kun er basert på to eksempler. Neste kapittel vil redegjør

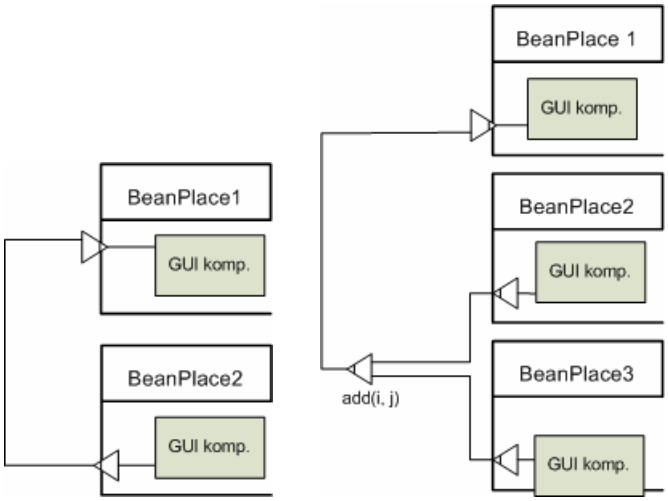
for regler som må implementeres i genereringskoden for at metoden skal kunne ses på som generell.

8 Generell metode for kodegenerering

I kapittel 7 ble det på bakgrunn av to DiaMODL-eksempler vist hvordan det kan genereres kjørbar kode. Koden realiserer funksjonaliteten i eksemplene, og er strukturert etter et mønster som er ment å være felles for all generert kode. Metoden som er vist er ment å være første ledd på veien mot en generell metode. Dette innebærer at den nåværende utgaven av genereringskoden har begrenset funksjonalitet. Begrensningene er et resultat av mangler som gjør at genereringskoden enda ikke kan ses på som generell. Identifiseres disse har man en oversikt over hva som mangler for at dette skal være tilfelle. Dette kapittelet vil gjøre rede for hva som må endres og legges til for at en generell kodegenerering skal kunne implementeres. Metoden som vil bli brukt er en gjennomgang og analyse av DiaMODL-modeller som er mer kompleks eller har annen funksjonalitet enn modellene som hittil har blitt brukt. Gjennomgangen gjør det mulig å gjøre rede for hva som ikke fungerer, hvorfor det ikke fungerer og hva som må endres for at det skal fungere.

8.1 Identifisering av utfordringer og mangler

Dette avsnittet vil redegjøre for to potensielt store utfordringer i den eksisterende genereringskoden. Som det vil bli vist er disse basert på bruken av begrepet modell.

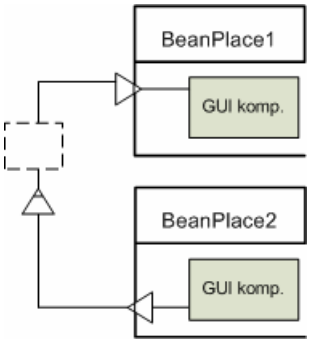


Figur 63 – Oppgavens DiaMODL-eksempler uten modellrepresentasjon

I DiaMODL-modellene er begrepet modell brukt om et BeanPlaceobjekt som lager en verdi. Den genererte kodens modellklasse realiserer denne datalagringsfunksjonaliteten.

Samtidig brukes modellklassen som et grunnlag for implementasjonen av objektmodellens koblinger. Dette gjøres ved å la klasser oppdatere eller lytte på verdiene i modellklassen.

Den genererte koden bruker eksistensen av modellrepresentasjonen som utløsende faktor for at det skal bli generert en modellklasse. Dette er problematisk da ”modellen” i en DiaMODL-modell ikke er en egen komponent. Designer kan velge å bruke modellbegrepet på samme måte som det er vist i kapittel 4 og 7, men kun da som en representasjon av data. Figur 63 illustrerer dette ved å vise DiaMODL-modeller av eksemplene brukt i kapittel 7 uten modellrepresentasjonen. Modellenes logiske funksjonalitet er ikke forandret, men på en annen side er det heller ingen ting som forteller genereringskoden at det skal genereres en modellklasse. Videre kan en DiaMODL-modell ha flere interaktorer som sender fra seg forskjellige data. Dette fører til at en generell kodegenerering må kunne bestemme hvilke, og hvor mange modellklasser som skal genereres selv om det ikke eksisterer en modellrepresentasjon.



Figur 64 – DiaMod-modell uten modellkomponent, men med beregning

I kapittel 7 ble nøsting av beregninger og implementering av disse drøftet. Samtidig ble det redegjort for implementasjon av funksjoner på inngående porter. Begge deler er funksjonalitet som må støttes i en generell kodegenerering, men er også innledningen til den andre utfordringen som eksisterer i forbindelse med en generalisering. Figur 64 viser en DiaMODL-modell med en beregning. Ser man bort fra den stiplede firkanten har modellen ingen modellrepresentasjon. Firkanten viser hvor en slik modell kunne vært plassert, men det er ingen ting i veien for at den kunne vært satt mellom beregningen og BeanPlace2 i stedet for mellom beregningen og BeanPlace1. Dette problemet ble adressert i forrige avsnitt, men utfordringen her er ikke bare hvor modellen logisk skal

plasseres. Det må også bestemmes i hvilken BeanPlaceklasse beregningen skal implementeres. Med modellrepresentasjonen på plass ville genereringskoden kunne avgjøre dette. Hadde beregningen ligget mellom BeanPlace2 og firkanten ville data passert gjennom beregningen før verdien ble satt i modellen. BeanPlacens metode for representasjon av utgående port ville da være den naturlige plassen for en implementasjon av funksjonen. Det motsatte blir tilfelle hvis firkanten plasseres mellom BeanPlace2 og beregningen. Det ville ført til at data hadde blitt satt i modellen før den passerte gjennom beregningen. Funksjonsimplementasjonen ville da ha sin naturlige plassering i BeanPlace1.

En generell genereringsmetode må implementere en algoritme som på bakgrunn av modellers og beregningers plassering løser begge utfordringene.

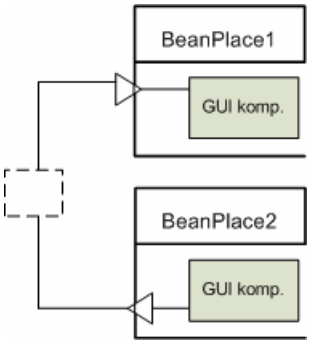
8.2 Drøfting og analyse DiaMODL-modeller

Dette avsnittet vil ved en gjennomgang, drøfting og analyse av flere og mer komplekse DiaMODL-modeller utdype utfordringene presentert i forrige avsnitt. Til slutt vil det bli presentert konkrete regler som en generell genereringsalgoritme må implementere for at utfordringene skal kunne løses.

Figur 65 viser DiaMODL-modellen brukt i kapittel 7s Eksempel1 uten en modellrepresentasjon. Siden den genererte koden har en modellklasse som en del av konstruksjonen, må genereringen av denne være basert på et annet grunnlagt enn eksistensen av tilsvarende modell i DiaMODL-modellen. Siden modellklassen er laget for å representere data vil den en mest naturlige løsning vil være å generere klassen basert på hvilke interaktorer (BeanPlaceobjekter) som sender fra seg data. På grunn av den genererte kodens oppbygging må data som sendes fra en interaktor eksponeres og være tilgjengelige for andre BeanPlaceklasser. Dette skjer ved observering av modellklassen [7.5.4]. Den stiplede firkanten viser hvor modellen opprinnelig var plassert[17]. Samtidig viser den også hvor en generert modellklasse logisk må være plassert for at koblinger skal bli implementert riktig. Koden generert på bakgrunn av figuren skal være lik koden

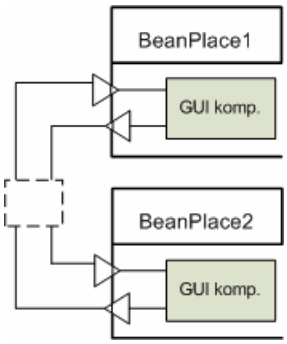
[17] Den stiplede firkanten vi også bli brukt i videre figurer for å vise hvor modellklassen logisk kan plasseres.

generert for Eksempel1. Dette er naturlig da det ikke er noen funksjonelle forskjeller i figuren med eller uten modellrepresentasjonen.



Figur 65 – Basis DiaMODL Modell

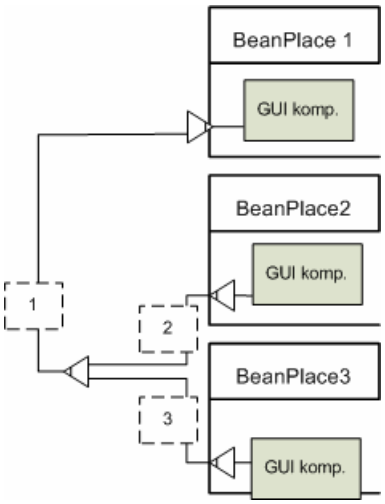
Figur 66 viser en annen variant av figur 65. Forskjellen er at begge interaktorene sender data til hverandre. Dette leder til følgende problemstilling: Hvis sending av data skal være grunnlaget for at modellklassen genereres, skal det da i dette tilfelle genereres en eller to modellklasser? Genereringskoden må implementere logikk som på bakgrunn av andre opplysninger bestemmer dette.



Figur 66 – Gjensidig sending av data

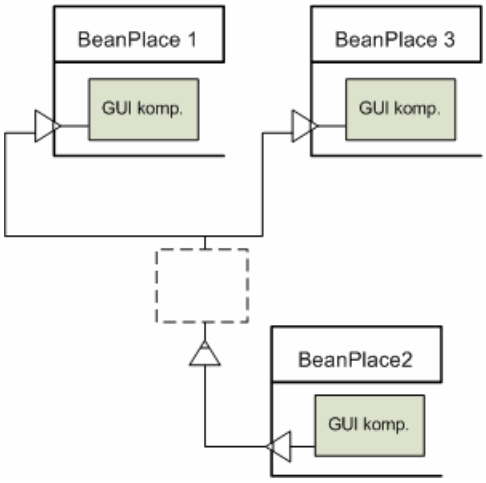
Figur 67 kan kjennes igjen fra Eksempel2 i kapittel 7. I tillegg til å illustrere problemet med hvor modellen skal plassers viser også figuren problemet med plassering og implementasjon av beregninger. Som vist er dette to varianter av samme problem. I den opprinnelig modelleringen var modellrepresentasjonen plassert mellom beregningen og BeanPlace1. Den stiplede firkanten (1) viser dette. Samtidig er det tegnet enda to firkanter (2 og 3). Disse sitter på hver sin utgående kobling fra henholdsvis BeanPlace2 og 3. Dette er konsistent med antagelsen om at data som kommer fra en interaktor må eksponeres i en modellklasse. Plasseres modellen etter beregningen (1) er det resultatet av denne som lagres. Implementasjonen av beregningen vil være i BeanPlacene

BeanPlace1 og BeanPlace2. Dette er måten det den eksisterende kodegenereringen gjør det på, men det på grunn av eksistensen av en modellrepresentasjon. Den andre muligheten er at hver av de to interaktorene som sender data har sin egen modellklasse (2 og 3). Beregningen må da være implementert i BeanPlace1 og denne må registreres som lytter på begge modellklassene. Når en oppdatering skjer i den ene interaktoren hentes data fra begge og brukes som parameter i beregningens funksjon. Begge implementasjoner er fullt mulig, og bestemmes av valg av algoritme og genereringsregler.



Figur 67 – Mulige logisk plassering av modellklasser

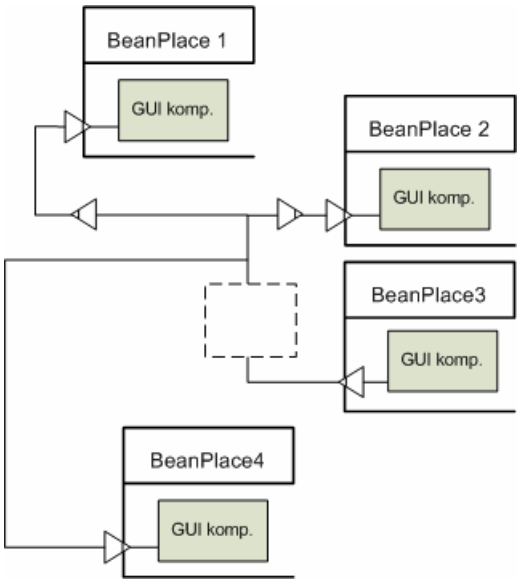
Figur 68 viser en DiaMODL-modell som definere starten på en regel for hvor modellklassen logisk skal plasseres og hvilke BeanPlacer som skal implementere beregninger. Figuren har en interaktor som sender fra seg data. Resultatet av beregningen brukes av to andre interaktorer. Det vil si at data som går ut fra beregningen deles av to andre interaktorer uten å på noen annen måte forandres. Det gjør at det vil være logisk å legge resultatet av beregningen i en modell som de to andre interaktorene lytter på. Beregningens funksjon kan implementeres av interaktoren som sender data.



Figur 68 – Deling av data som kommer fra en beregning.

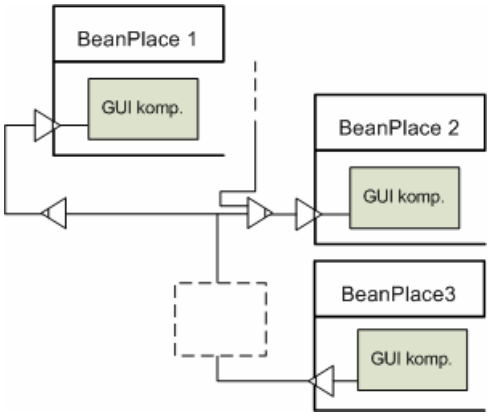
Figur 69 viser en DiaMODL-modell hvor beregningene er plassert er motsatt av hva som var tilfelle i figur 68. Modellen har fire interaktorer. BeanPlace4 får data uforandret fra BeanPlace3. Dette vil si at den må hente data fra en modell som ikke har en verdi forandret i en beregning. Data som kommer bra BeanPlace3 brukes også i to forskjellige beregninger. Dette gjør at data som blir send ikke kan oppfattes som felles for flere kilder[18]. BeanPlace1 og 2 bruker hver sin beregning, og må derfor i den bruke sin egen instans av verdien hentet fra modellen. For å realisere denne funksjonaliteten må modellen logisk ligge før beregningen, og beregningen må implementeres i metoden for inngående porter i BeanPlace1 og BeanPlace2

[18] Felles data: Data som uforandret brukes av en eller flere mottaker.



Figur 69 – Modellklassen logisk plasseres før beregningene.

Figur 70 er en variant av figur 69. BeanPlace4 er tatt bort da den ikke er viktig for hva som skal illustreres. Hadde den vært med, og hadde modellen logisk vært plassert etter beregning, måtte en egen modellklasse blitt generert slik at BeanPlace 4 kunne hente data uforandret. Dette er ikke tilfelle, men er verd å legge merke til da slike tilfeller vil kunne oppstå. BeanPlace2 får inn en verdi som kommer fra en beregning som har to innkommende koblinger. Dette forsterker argumentasjonen for den foreslåtte modellplasseringen er riktig. Dette er data som ikke er deles med andre interaktorer så er det naturlig at implementasjonen av beregningen legges til den inngående porten.

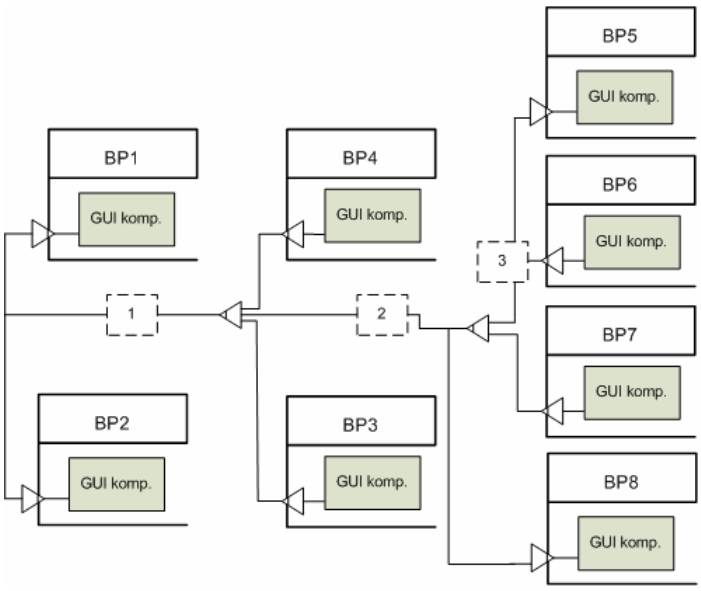


Figur 70 – Variasjon av figur 69

De to foregående figurene skisserer regler for en logisk løsning på problemet med plassering av modellklassen og implementasjonen av beregninger. At dette er den riktige fremgangsmåten kan ikke verifiseres uten det prøves på en større modell. Den faktiske

implementasjonen må også gjøres rede for. Gjennomgangen av figur 71 oppsummerer det som har blitt presentert så langt.

Figuren viser at Bp1 og Bp2 kun får inn data. Bp3 og Bp4 sender fra seg data brukt i en beregning. Data som passerer gjennom beregningen brukes uforandret av Bp1 og Bp2 og kan derfor betegnes som felles data. Følges fremgangsmåten presentert for figur 69 og 70 skal Bp3 og Bp4 oppdatere en modellklasse (1) som Bp1 og Bp2 lytter på. Verdien denne modellklassen lagrer er verdien som kommer fra beregningen. Implementasjonen av beregningens funksjon vil være i metoden som representerer Bp3 og Bp4's utgående porter. Det som kompliserer det resonnementet er at beregningen har en trede inngående kobling. Kallet til funksjonen vil derfor ha tre parametere. Koblingen representerer data som kommer fra en annen beregning. Siden en beregnings utgående verdi brukes som parameter i en ny beregning er det en nøsting [7.6.8].



Figur 71 – DiaMODL-modell som grunnlag for generelle genereringsregler.

Bp8 lytter på data på som kommer fra denne beregningen uten at det gjøres andre endringer. Data inn til beregningen kommer fra Bp6 og Bp7, og sett fra Bp6 og Bp7's synsvinkel sender de data til en beregning som igjen sender fra seg felles data. Dette kjennes igjen fra figur 68 og sannsynliggjør at beregningens funksjon bør implementeres i Bp6 og Bp7. Modellen (2) kan av den grunn logisk plasseres utenfor, eller etter, beregningen. Dette stemmer også med hvordan Bp8 skal kunne hente sin verdi. Klassene som implementer den første beregningen bruker modellklassen lengst til venstre (1) for å

lagre sine egne verdier. Samtidig henter de data fra modellklassen lengst til høyre når beregningen skal implementeres.

Den siste BeanPlaceklassen er Bp5. Den lytter på data som kommer fra Bp6. Disse dataene er uforandret. Siden Bp6 også sender fra seg data til en beregning, må Bp5 hente data fra en egen modell (3). Hvis betraktningene over oppsummeres og konkretiseres er resultatet at man sitter igjen med to regler som i implementert tilstand kan representere en generell metode for kodegenerering av alle DiaMODL-modeller. Disse er:

- Hvis data fra en beregning ikke forandres før den kommer til en porter data felles. Modellklasse kan logisk plasseres etter beregningen. Beregningens funksjon implementeres i BeanPlacen(e) som sender fra seg data.

- Det motsatte er tilfelle hvis data fra en BeanPlace eller beregning brukes i en eller flere nye beregninger. Data er da ikke felles, og det må opprettes en modell som logisk ligger før beregningen. BeanPlaceklassene som mottar data må implementere beregningens funksjon.

Frameklassen styrer hvilke klasser som registres som lyttere på de forskjellige modellene og realiserer dermed koblingene mellom objektene i objektmodellen. Koden for frameklassen i figur 71 er vist i Kodeliste 25

```
...
private Model3 model1 = null;
private Model3 model2 = null;
private Model3 model3 = null;

private BeanPlace1 bp1 = null;
private BeanPlace2 bp2 = null;
private BeanPlace3 bp3 = null;
private BeanPlace4 bp4 = null;
private BeanPlace5 bp5 = null;
private BeanPlace6 bp6 = null;
private BeanPlace7 bp7 = null;
private BeanPlace8 bp8 = null;

model1.addObserver(bp1);
model1.addObserver(bp2);
//model2.addObserver(bp3);
//model3.addObserver(bp4);
model2.addObserver(bp8);
model3.addObserver(bp5);

…//Getters and Setters
```

Kodeliste 25 – Frameklassen generert for modellen i figur 74

I koden er det kommentert ut at Bp3 og Bp4 registres som lytter på modell2. Dette er for å illustrere to mulige hendelsesforløp. Hvis klassene ikke registreres som lyttere på

modell2 vil verdien bli sendt til Bp1 og Bp2 kun når enten Bp3 eller Bp4's verdi forandres. Det går også an å tolke på en annen måte; Modell1's verdi er summen av tre koblinger, hvorav en kommer fra modell2. Forandres verdien til modell2, så skal dette propageres videre igjennom beregningen til modell1. Bp1 og Bp2 implementerer beregningen som setter verdien i modell1, og må av den grunn gjøres oppmerksom på forandringen i modell2.

De påfølgende kodelistene viser metodene som implementerer inn- og utgående porter til de forskjellige BeanPlaceklassene. Kode for sjekk på andre hendelser [7.6.6] og stenging av porter [7.5.6] er utelatt. Det samme er metodene som representerer porter som interaktoren ikke har. For å illustrere beregningers funksjonalitet antas det at beregningene har en funksjon som legger samme verdiene som komme inn (”add()”).

```
public void update(Observable obs, Object obj){
 if (obs == model1){
    if (ingateopen){
      guiComponent.setValue(model1.getValue());
}
```

Kodeliste 26 – Kode for Bp1 og Bp2

```
private void init(){
...

public void action(){
       model1.setValueBp3(guiComponent.getValue());
       model1.setValue(add(guiComponent.getValue(), model1.getValueBp4(), ¬
model2.getValue()));
}
```

Kodeliste 27 – Kode for Bp3 versjon1. Kode for Bp4 er tilsvarende

```
public void update(Observable obs, Object obj){
 if (obs == model1){
    if (ingateopen){
      guiComponent.setValue(model1.getValue());
      action();
}
private void init(){
...

public void action(){
       model1.setValueBp3(guiComponent.getValue());
       model1.setValue(add(guiComponent.getValue(), model1.getValueBp4(), ¬
model2.getValue()));
}
```

Kodeliste 28 – Kode for Bp3 versjon2. Kode for Bp4 er tilsvarende

Kodeliste 28 viser hvordan koden blir hvis Bp3 og Bp4 settes til å lytte på modell2 slik at verdien av denne kan propageres videre. Om dette skal implementeres er et valg som må gjøres i den generelle algoritmen.

```
public void update(Observable obs, Object obj){
 if (obs == model1){
    if (ingateopen){
      guiComponent.setValue(model5.getValue());
}
```

Kodeliste 29 – Kode for Bp5

```
private void init(){
...

public void action(){
       model2.setValueBp6(guiComponent.getValue());
       model2.setValue(add(guiComponent.getValue, model2.getValueBp7()));
}
```

Kodeliste 30 – Kode for Bp6. Tilsvarende for Bp7

```
public void update(Observable obs, Object obj){
 if (obs == model1){
    if (ingateopen){
      guiComponent.setValue(model2.getValue());
}
```

Kodeliste 31 – Kode for Bp8

Det som enda ikke har blitt gjort rede for et funksjonaliteten i figur 66. Den genererende koden må avgjøre om det skal genereres en eller to modeller. Det er i utgangspunktet to mulige løsninger på dette. Siden det ikke eksisterer noen beregninger kan man konkludere med at data er felles. Det lyttes altså på same verdi. Dette tilsier at det skal genereres en enkel modellklasse. Implementasjonen av en slik logikk kan utelates for å unngå kompleksitet i genereringskoden. Det vil da bli generert to modellklasser. Hver BeanPlace vil da bli registrert som lytter på hver sin modellklasse.

Ut fra drøftingen og gjennomgangen av modellene i dette kapittelet kan man konkludere det med at de presenterte reglene vil være mulig å bruke i en generell genereringsalgoritme. Neste avsnitt vil kort presentere de endringer som må gjøres i genereringskoden for at den presenterte funksjonaliteten skal kunne implementeres.

8.3 Endringer i genereringskoden

Som det har blitt vist, så er ikke traverseringsalgoritmen avhenging av en modellrepresentasjon for å traversere gjennom alle objektene i objektmodellen. Måten den undersøker koblinger mellom objektene og objektmodellens oppbygning gjør at den trenger å endres. Klassene i pakken generators trenger heller ikke å endres som direkte følge av en generell metode. Et unntak er utvidelse av koden som generere metodene som representerer inn- og utgående porter. Disse vil i et generelt tilfelle kunne få noe utvidet funksjonalitet. Genereringsklassene er tilpasset en bestemt klassestruktur. Eventuelle endringer i disse klassene vil komme som følge av at variasjoner av eksisterende kode skal kunne genereres. I all hovedsak vil endringer bli dekket av nye kall til grensesnittklassen.

Genereringslogikken er lagt til klassen Codegenerator. Det er algoritmene i denne som må endres hvis de skisserte reglene skal implementeres. Som vist så opprettes det først en oppstartsklasse. Dette gjelder for generering av alle DiaMODL-modeller. Deretter opprettes modellklassen. Referansen til denne blir brukt av både frameklassen og BeanPlaceklassene. Genereringen av BeanPlaceklassen skjer ved at objektmodellens BeanPlaceobjekter blir gått gjennom en for en. Avhenging av hvilke opplysninger som er tilgjengelige, blir parametere som BeanPlaceklassen bruker også satt i modell og frameklassen. Til slutt blir det undersøkt om data som sendes ut eller mottas av klassen blir brukt i en beregning. Hvis det er tilfelle blir beregningens funksjon generert som en del av metoden som setter den offentlige verdien i modellklassen. Eksistensen av en modellrepresentasjon gjør det unødvendig å sjekke på om en beregning skal tilhøre ut- eller inngående port. Når denne fjernes må modellklassen i stedet genereres på bakgrunn av de reglene som ble presentert i forrige avsnitt. Siden den logiske plasseringen av modellklassen styres av DiaMODL-modellens bruk av beregninger må genereringskoden bestemme hvor modellen skal plasseres før BeanPlaceklassen faktisk blir generert.

En mulig genereringssekvens er at oppstartklassen og frameklassen opprettes på samme måte som nå, bortsett fra at frameklassen ikke får vite om modellklassen. Referansen til denne, eller disse, må settes inn i forbindelse med genereringen av BeanPlaceklassene.
Etter frameklassen er opprettet må det undersøkes om BeanPlaceklassen sender fra seg eller mottar data. Begge deler kan være tilfelle. Sender klassen fra seg data må det

undersøkes om målet for dataene er en beregning eller en annen port. Eksistere en beregning må det implementeres en algoritme som sjekker om data sendt gjennom beregningen deles eller brukes i en ny beregning. Algoritmen må også kartlegge om beregning får data fra andre kilder enn fra den BeanPlaceklassen som blir undersøkt. Ut fra beregningens natur må det genereres en modellklasse. Klassens logiske plassering bestemmes ved å sette hvilke klasser som lytter på hvilken modell. Hvilke klasser som lytter på eller oppdaterer modellen bestemmer også hvilke ekstra parametere den skal ha for lagring av data. Har BeanPlacen en inngående port, må det sjekkes om data som kommer inn kommer via en beregning eller direkte fra en annen port. Hvis en beregning eksisterer må det sjekkes om data inn til beregningen også brukes i en annen beregning. Dette vil være førende for om beregningens funksjon skal legges til klassens inngående portmetode.

I utgangspunktet er det i objektmodellen ikke noe skille mellom en port og en beregning. Kapittel 7.5.3 drøfter kort at et slikt skille allikevel ville kunne være praktisk. Et eksempel på det er nettopp i algoritmen presentert ovenfor. Strukturen i objektmodellen, og DiaMODL-modellen i seg selv, viser at på utsiden av en BeanPlace vil det alltid være en port. En port har en kobling enten til en beregning eller en annen port. Finner man en port i den andre enden av en ports utgående kobling, så vet man at data ikke skal gjennom en beregning. Dette kan utnyttets når modellklassens logiske plassering skal bestemmes.

En annen utfordring som må en generell metode bør dekke er hvis er hvis designer faktisk velger å bruke en modellrepresentasjon i DiaMODL-modellen. Dette kan håndteres på to måter. Den første er at genereringskoden overser modellrepresentasjonen og generer modellklasser på bakgrunn av de opplysningene den selv finner. Dette vil føre til at genereringskoden blir mindre kompleks da det ikke er nødvendig å ta hensyn til en modellrepresentasjon som muligens har en annen plassering den genereringslogikken støtter. Samtidig så kan det skape problemer at den overses. Oppførselen til det genererte programmet vil kunne oppleves annerledes enn det designer så for seg når han eller hun plasserte modellen. Skal det tas hensyn til manuelt plasserte modeller, så må genereringskoden også sjekke hvilke data modellrepresentasjonen representer, og på bakgrunn av det unngå at det blir generert modellklasser som representerer samme data.

De presenterte endringene medfører ikke store strukturelle forandringer i genereringskoden. Det er selve genereringslogikken må endres. Naturlige punkter for videre arbeid med dette, samt flere spesifikke utfordringer vil bli presentert i kapittel 10.

9 Drøfting av resultat og arbeidsmetode

Et resultat dreier seg om hva som er oppnådd når det blir sammenlignet med et mål. Målene for denne oppgaven er definert som en problemstilling med tilhørende løsningskrav og avgrensninger [5]. Dette kapittelet vil oppsummere hva som har blitt gjort, og på bakgrunn av det kort drøfte resultatene. Er det som er gjort, og det som er oppnådd, i henhold til problemstilling og løsningskrav? Var fokus riktig og ble det valg passende eksempel? Hvis dette ikke er tilfelle, hva kunne vært gjort annerledes?

Det vil først bli redegjort for arbeidet med løsningskravene i problemstillingen. Deretter vil arbeidsmetode, prosess og resultat bli drøftet på et mer generelt nivå. Til slutt vil det bli vist eksempel på at løsningen også har en funksjonell kvalitet.

9.1 Verifisering av løsningskrav

Realisering av løsningskrav er det som fører til resultat. Dette avsnittet vil på bakgrunn av løsningskravene definert i problemstillingen oppsummere og drøfte hva som har blitt oppnådd. Den overordnede problemstillingen blir drøftet til slutt, så det innledende problemet er:

> *"Den genererte koden skal så langt det er mulig implementere oppførselen spesifisert i modellen. Dette betyr at det for hvert modell-element må genereres kode som implementerer tilsvarende oppførsel."*

Koden som generer kjørbar javakode realiserer oppførselen til to DiaMODL-modeller brukt som eksempel. I det første eksemplet skulle verdien av et GUI-element bestemme verdien på et tilsvarende element i en annen interaktor. Den genererte koden kompileres uten feil, og programmet som kjøres viser to JSlider-komponenter. Når den ene slideren blir manipulert endres verdien tilsvarende i den andre slideren. For det andre, mer kompliserte, eksemplet skjer det samme. DiaMODL-modellen viser at det skal brukes en beregning som legger sammen verdier som kommer fra to interaktorer. Den kalkulerte verdien blir vist av den tredje interaktor. Det genererte programmet realiserer dette.

> *"Det må utvikles og presenteres en algoritme som sikrer at alle modell-elementer blir representert i den genererte koden".*

På bakgrunn av objektmodellens sammensetting og eksistensen av retningsbestemte koblinger ble det presentert en rekursiv algoritme som sikrer at alle modell-elementer blir besøkt. For å ha en plass å teste denne algoritmen ble det også utviklet et rammeverkt i Eclipse. Dette rammeverket ble senere brukt til implementasjonen av DiaModlGen.

Løsningskravet over konkretiseres i flere punkter:

> *"En BeanPlace (Interaktor) må oversettes til en klasse som kan presentere GUI-element."*

I en utflatet objektmodell er interaktoren fjernet. I stedet er interaktorens rolle delvis implementert i klassen BeanPlace, som også implementerer porter og beregninger. Interaktorens roller er å avgrense et område for interaksjon. Dette kan overses i en utflatet objektmodell. Den andre rollen er, via et BeanPlaceobjekt, å instansiere et GUI-element som bruker kan manipulere. Den genererte BeanPlaceklassen instansierer et spesifisert grafisk element, og tilbyr underliggende funksjonalitet til dette. Den underliggende funksjonaliteten er håndtering av hendelser i den grafiske komponenten, samt metoder som muliggjør kommunikasjon.

> *"En beregning eller port må oversettes til noe som beregner en ny verdi fra en eller flere andre verdier."*

Porter og beregninger eksisterer som egne objekter i objektmodellen. Siden interaktorene kan fjernes er det ingenting som i utgangpunktet skiller porter og beregninger. Den genererte koden håndterer allikevel beregninger og porter på en noe atskilt måte. Dette er fordelaktig hvis det senere skal genereres kode fra en objektmodell som ikke er flatet ut. Hverken porter eller beregninger er i generert form egne entiteter. BeanPlaceklassen har en metode ("action()")som håndterer sending av verdier som kommer fra GUI-komponenten. Tilsvarende metode for å sette verdier ("update()"). Disse metodene representerer inn og utgående data, og realiserer på den måten interaktorens porter.

En port eller en beregnings hovedfunksjonalitet er å kunne beregne en ny verdi basert på andre verdier. Dette håndteres i den genererte koden ved å implementere porters og beregningers funksjoner i funksjonskallet som setter verdien på modellklassen. Det gjør at data passerer gjennom et metodekall i stedet for et objekt, men den logiske funksjonen

blir den samme. Funksjonskallets parameter representerer antall koblinger inn til porten eller beregningen.

> *"En kobling må oversettes til kode som lytter på en endring i kilden og videreformidler denne til et mål."*

I objektmodellen er koblinger egne objekter. Dette er ikke tilfelle i den genererte koden. Koblinger realiseres ved at data lagres i en modellklasse. BeanPlaceklasser setter verdier på denne modellen ved å bruke metoden beskrevet i forrige punkt. På sammen måte hentes data inn ved å lytte på endringer i modellens verdi. Hvilke BeanPlaceklasser som oppdaterer hvilke modeller, og hvilke klasser som lytter på den sammen modellen bestemmes i frameklassen, og realiserer på den måten koblinger mellom objekter.

> *"En interaktor/tilstand grupperer elementer som aktiveres/deaktiveres som en enhet. I praksis er dette implementasjonen av DiaMODL-editorens støtte for Statechart XML [5.4]"*

I en utflatet objektmodell er ikke representasjonen av tilstander eller en interaktor tatt høyde for. En støtte for deaktivering/aktivering er allikevel implementert i form av boolske variabler i BeanPlaceklassene. Verdiene av variablene sjekkes før data transporteres ut eller inn av klassen. Denne funksjonaliteten kan ved en videreutvikling brukes til å realisere aktivering og deaktivering av en interaktor, noe som vil være aktuelt i kodegenerering basert på en komplett objektmodell.

Punktene over viser at løsningskravet om at alle DiaMODL-elementer skal kunne oversettes til en funksjon i den genererte koden er oppfylt. Sammen med en strukturering av koden blir mulig å gjenskape funksjonaliteten i DiaMODL-modellene.

Den overordnede problemstillinger for oppgaven var:

> *"Er det mulig å utarbeide en metode som beskriver hvordan en DiaMODL-modell kan oversettes til kjørbar kode?"*

Som vist, så er det ved hjelp av et utviklingsrammeverk implementert genereringskode som gjør det mulig å realisere oppførselen i et sett DiaMODL-modeller. Hvert DiaMODL-element er oversatt til funksjoner i en generert koden, som igjen er organisert

i en klassestruktur. Resultatet av implementeringen ble et viktig grunnlag for de to neste løsningskravene:

> *"...går man et skritt videre er det naturlig å se om en slik metode vil kunne realiseres på et generelt nok vis til at kode kan genereres for alle DiaMODL-modeller?"*
>
> *"Basert på utfordringer og mangler funnet i arbeidet må det utledes regler for en generell genereringsmetode, samt hva som må endres for at denne skal kunne implementeres."*

Den presenterte klassestrukturen gjør det mulig å generere kode fra DiaMODL-modeller uten noen form for begrensning. Dette er nødvendig i en generell metode. Traverseringsalgoritmen besøker alle objektene i en DiaMODL-objektmodell, noe som også nødvendig. Koden som generer kode ble utviklet på bakgrunn av to eksempler. Selv om de valgte eksemplene dekker alle elementene en DiaMODL-modell kan ha, er de allikevel begrenset i størrelse og kompleksitet. Resultatet ble begrensinger i genereringskoden, noe som gjør at den ikke kan ses på som generell. Implisitt støttes ikke generering av alle mulige DiaMODL-modeller. Disse begrensingene lar seg identifisere, noe det er gjort rede for. Resultatet er et sett med regler som må implementeres. Når dette er gjort vil det være mulig å definere metode som støtter et ukjent antall interaktorer og koblinger, samt fraværet av en eller flere modell-representasjoner. Det er også vist at genereringsrammeverket er tilstrekklig nok for at den generelle algoritmen skal kunne implementeres. Det generelle nivået på metoden lar seg ikke bevise, men den kan etter en implementasjon verifiseres empirisk. Dette støttes av at det under arbeidet av metoden ble testet på flere DiaMODL-modeller med konstruert kompleksitet.

9.2 Funksjonelle resultat og utvidelsesmuligheter

De funksjonelle kravene til implementeringen er i all hovedsak basert på normer for god programmering og har sånn sett lite å si for resultatet av problemstillingen. På en annen side så er muligheten for et videre arbeid, og videre implementering av DiaModlGen i DiaMODL vært et viktig fokus. Siden den genererte koden skal være et konkret utgangspunkt for utvikler er muligheten for enkel manuell utvidelse og endring av koden viktig. Den genererte koden skal være et startpunkt for et videre implementasjonsarbeid, og det er da viktig koden er lettfattelig og enkel og utvide. Begrepet utvidelse er todelt:

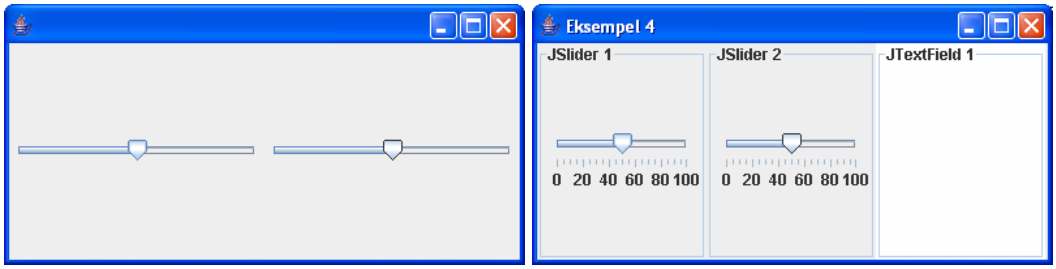
Endring og utvidelse av den genererte koden

Under er det vist hvordan manuell endring kan endre utseende og funksjonalitet på GUI-komponentene. Dette er viktig, da hver GUI-komponent kan ha parametere som ikke lar seg hente ut fra DiaMODL-modellen. En manuell endring vil være nødvendig for å oppnå ønsket funksjonalitet. Metoden init() i alle BeanPlaceklasser blir kalt fra klassens konstruktørmetode (eng: constructor) og brukes til i konfigurere klassens komponenter. Kodeliste 32 viser hvordan utvikler kan legge til komponentspesifikk endringer.

```
private void init(){
//Koden under er lagt til manuelt
                guiComponent.setMinimum(0);
                guiComponent.setMaximum(100);
                guiComponent.setMajorTickSpacing(20);
                guiComponent.setMinorTickSpacing(5);
                guiComponent.setPaintTicks(true);
                guiComponent.setPaintLabels(true);
...
```

Kodeliste 32 – Manuell endring av generert kode

Figur 72 viser skjermbildet hentet fra den genererte koden før og etter manuell endring.



Figur 72 – Skjermbilde med og uten manuell modifikasjon

En annen utgave av manuell endring er at utvikler kobler den genererte koden sammen med eksisterende program eller kode. Her er mulighetene uendelige. Et eksempel er at en generert grafisk komponent blir brukt til å vise data hentet fra en database. Dette må gjøres manuelt, og utvikler styrer hvordan dette skal skje.

Endring og utvidelse av genereringskoden

Hovedfokus i et videre arbeid vil være å implementere en generell kodegenerering. Samtidig vil det bli vist at en utvidelse av den eksiterende funksjonaliteten også er viktig. Dette vil bli drøftet i neste kapittel, men vil kort bli vist her da muligheten for dette er et funksjonelt resultat oppstått på bakgrunn av valgt struktur for den genererte koden.

Eksemplene brukt i oppgaven er lite varierte når det gjelder utvalg av grafiske komponenter. Kun et utvalg av er støttet og brukt. Dette avsnittet vil vise at strukturen på den genererte koden er så generell at det på en enkel måte er mulig å utvide støtten for grafiske komponenter. JList er en Java SWT-komponent som viser frem et eller flere elementer organisert i en liste. Listens parametere bestemmer om det er mulig å velge et eller flere elementer samtidig. Dette fører til to mulige scenario. Uansett, så er det første som må gjøre å legge inn støtte for JList i mapperklassen, likt JSlider og JTextField. Når genereringskoden finner en SWT-liste i DiaMODL-modellen er den i stand til, via mapperklassen, å generere kode for tilsvarende Swing-liste.

Når det blir valgt et element sender listen fra seg et objekt av typen ”Object”. Dette propageres videre og lagres i en modellklasse. Interaktoren som skal vise frem valget har et tekstfelt som grafisk komponent. Datatypen tekstfeltet skal ha inn er tekst (javaobjektet String). Siden String og Object er to forskjellige typer må det konverteres fra det ene til det andre. I dette tilfelle er det Object som må gjøres om til String. Dette er en funksjonalitet i Java[19] og gjøres som i kodeliste 33. Typen det skal konverteres til settes i parentes før kallet til det opprinnelige objektet. Metoden som er vist representerer inngående port. Om forskjell i objekttype skal detekteres av genereringskoden eller om det manuelt må skrives inn i porten som en funksjon er et valg som må gjøres ved implementering av genereringsalgoritmen. Begge deler er mulig, men DiaMODL-editoren støtter på nåværende tidspunkt ikke annet enn standard javafunksjoner.

```
update(){
        guiComponent.setValue((String)model1.getObject());
}
```

Kodeliste 33 – Konvertering av objekter

Det andre mulige scenarioet ved bruk av en liste er det velges flere elementer fra listen samtidig. JList vil da sende fra seg en liste (eng: array) med de valgte elementene. Siden flere elementer er valg, vil det måtte vises i en tilsvarende liste i interaktoren som tar i mot data. Dette er fordelaktig da det ikke må konverteres fra en datatype til en annen, men modellklassen må i samarbeid med mapperklassen støtte lagring av lister. Metoder

[19] Dette er funksjonalitet som også finnes i andre objektorienterte språk.

for henting og setting av verdier på og fra listene legges i mapperklassen. Skal noe annet enn å vise frem listen gjøres kan metoder for dette skrives manuelt og kalles fra klassen util.Funcs. Metodekallet legges da enten i den inngående porten eller en egen beregning.

9.3 Kritikk og vurdering av arbeidsmetode

Dette avsnittet vil drøfte den generell arbeidsmetoden, de målene som ble satt, samt kvaliteten og nivået på den tekniske løsning. Med arbeidsmetode menes valg og strategi før, under og etter defineringen av problemstillingen.

Arbeid med en problemstilling er en iterativ prosess. Det gjelder også for denne oppgaven, hvor arbeidet har vært iterativ på flere områder. Den første iterasjonsrunden dreide seg om hvilket fagområde som skulle adresseres. Deretter ble det gjort et arbeid over flere iterasjoner som gikk på sette seg inn i DiaMODL og hvilke deler av det som kunne fungerer som grunnlag for en problemstilling. Problemstillingen var den delen som var gjennom flest runder med iterasjon. Ikke bare for å spesifisere den, men også kombinert med testing og utprøving av forskjellige implementasjonsmetoder og avgrensninger. Selv etter problemstillinger var definert var det uten en faktisk implementasjon vanskelig å se i hvor stor grad problemene og løsningskravene måtte detaljeres for å representere et konkret nok problemstilling. Dette viser at mange iterasjoner ikke trenger å være utelukkende positivt. Ved manglende forståelse av det steget man befinner seg på, vil det gjøre det verre å gå videre til neste iterasjon. Momenter vil bli oversett og glemt, og iterasjonen kan føre til at man mister fokus. I verste fall baseres en utvikling på feil grunnlag. Dette har til tider vært tilfelle under arbeidet med å spesifisere oppgavens problemstilling. En konkret slik hendelse er oppdelingen i kodegenerering for utvalgte eksemper og utarbeidelsen av regler for en generell metode. Med bedre oversikt før implementering startet ville denne ekstra iterasjonen vært unødvendig. Fokus på en generell metode fra starten av ville også ført til en mer helhetlig implementasjon enn hva som kan bli tilfelle når denne utvikles i ettertid.

Den lengste iterasjonsrunden var i forbindelse med implementering av genereringskoden. Programmering og utvikling er, og skal være, en iterativ prosess, men antall iterasjon kan reduseres ved å få forhånd ha god nok kunnskap om det som skal utvikles. I ettertid kan det drøftes om det var tilfelle med denne oppgaven. Det har blitt vist kodegenerering av

DiaMODL-modeller er mulig. Det har også blitt presentert regler for en generell algoritme. Sett i etterpåklokskapens lys, ville et bedre valg av eksempler som implementeringsgrunnlag vært fordelaktig. Valget av såpass enkle og lite kompliserte eksempler førte til at når fokus først ble satt på implementasjonen av disse, ble det lett å overse andre viktige momenter. Hadde ufordringen eksemplene ikke dekket blitt oppdaget og tatt høyde for ville veien mot en generell genereringsalgoritme kunne vært kortere. Muligheten for en faktisk verifisering ville også vært bedre. Når det er sagt, skal det sies at den rammen DiaModlGen utgjør er laget for å kunne utvides. Det arbeidet som er gjort er veloverveid og funnet brukbart på mange områder.

Kritikk av oppgavens avgrensing er også på sin plass. Resultatet av avgrensingen er at det på steder hvor det kunne blitt utviklet mer sofistikerte løsninger, så har ikke dette blitt gjort fordi det ikke var nødvendig for å oppnå ønskede funksjonalitet. Bakgrunnen er i all hovedsak tidsforbruk, men også for å sikre at fokus ble holdt på det faktiske problemet. I ettertid er det mulig å se at avgrensingene kanskje burde vært konkretisert i større grad, slik at fokuset på en generell løsning ville fått større fokus på et tidligere tidspunkt.

10 Videre arbeid og utfordringer

DiaMODL i seg selv er en forholdsvis ny notasjon. Det medfører at grunnlaget for den praktiske bruken enda ikke er ferdig utforsket. Implementasjonen av DiaMODL-editoren er enda nyere, og enda i stadig utvikling både hva bruksområder og ny funksjonalitet angår. Det medfører at både i løpet av den tiden det har tatt å skrive denne oppgaven, og i tiden fremover, så har det vært og kommer det til å være en utvikling både i det teoretiske og det praktiske grunnlaget oppgaven baserer seg på. Det arbeidet denne oppgaven til nå har presentert er ment å dekke eksisterende konsept i DiaMODL. Samtidig har det blitt forsøkt å ta høyde for kommende utvidelser. På grunn av utviklingen kan den presenterte applikasjonen aldri se på som ferdig. Innenfor rammene problemstillingen presenter, så er det flere tema som har falt utenfor. Hovedårsaken til det er todelt. For det første så har det i enkelte tilfeller ikke eksitert et ferdig nok grunnlag til at et problem kan eller bør løses. For det andre så har mengden utfordringer vært så omfattende at noen har blitt sett på som viktigere. Et sett med konseptuelt mindre viktige aspekter befinner seg i et grenseområdet, men har falt utenfor problemstillingens avgrensning. Det samme gjelder for en del identifiserte tekniske utfordringer. Dette avsnittet vil redegjøre for flere av disse.

Arbeidet med DiaModlGen har to hovedfokus. Først og fremst kommer det teoretiske målet, som er å vise at DiaMODL-editoren og spesielt DiaModlGen kan tilføre en merverdi til en utviklingsprosess. Det andre målet har vært å vise at verktøystøtte for DiaMODL kan utvides i en retting som gjør det lettere å bruke sammen med eksiterende utviklingsparadigmer. Begge deler gjør at bruksområde til DiaMODL, og ikke minst bruksverdien vil bli utvidet og øke. På bakgrunn av dette er betydningen av videre arbeid ment å være ”videre utvikling og spesifisering av DiaModlGens teoretiske og tekniske bruksområde”.

Et teoretisk rettet videre arbeid vil innebære å gjøre rede for bruksområder, bruksmønster og anvendbarhet av både DiaMODL og DiaModlGen i en utviklingssituasjon. Det mest spennende feltet her vil være å se på mulighetene for å bruke verkøyet som et hjelpemiddel i forskjellige trinn i eksisterende utviklingsprosesser. Mer bestemt kan man

ved å spesifisere at man for eksempel i RUP, skal bruke DiaMODL i enkelte trinn av design og implementasjonsfasen i stedet for eksisterende verktøy som UML. Dette vil være med på å sette fokus på hva som må til for at DiaMODL og DiaModlGen skal videreutvikles i en mest mulig effektiv retting, noe som vil føre til at flere vil se en reell nytteverdi.

Et teknisk videre arbeid er i all hovedsak en videreutvikling av DiaModlGen, og naturlig å dele opp i to faser. Fasens fokus går på tvers av teknisk og teoretisk utvikling, men er i utgangspunktet forankret på den tekniske siden da teori er til liten nytte uten et teknisk grunnlag. Den første fasen innebærer en kvalitetssikring av eksisterende funksjonalitet samtidig som det som kan ses på som kjernefunksjonalitet blir videreutviklet. Eksempelvis vil dette være en utvidet støtte for større og mer komplekse DiaMODL-modeller. Den andre fasen bør være mer rettet mot tilpassing av DiaModlGen mer forpraktisk bruk samt utvidelse av anvendelsesområde.

I begge tekniske faser vil presentasjon av DiaMODL-editoren og DiaModlGen for potensielle bruker vil være avgjørende. Tilbakemeldinger vil fungere som en kvalitetssikring og produktet, samt gi føreringer på hva som er viktig å fokusere på. Samtidig vil man få en bonuseffekt i form av at produktet blir mer kjent. Et paradoks her at jo mer modent et produkt er før det presenteres, jo mer vil brukere se merverdien av det. Bakdelen er at man kan komme i den situasjonen at feil avdekkes for sent i utviklingsprosessen. Av den grunn vil brukertesting med et sett tilpassede scenarios[20] være å foretrekke i den første fasen av den videre utviklingen, mens man venter med en presentasjon og caser i et produksjonsmiljø til fase to.

10.1 Videre arbeid

Et videre arbeid med fase èn omhandler mulige rettinger DiaModlGen kan og bør utvikles seg i for å tilføre en merverdi for bruker, samt øke sannsynligheten for at det vil være mulig å ta det i bruk. Mer bruk vil igjen føre til mer og bedre utvikling. Funksjonsutvidelser kan ses på som generelle rettinger for et videre arbeid og har som mål og øke DiaModlGens brukbarhet og merverdi.

[20] Tilpasset i form av spesialiserte tester som direkte berører det som skal videreutvikles.

10.1.1 Implementering av generell algoritme

Som det kommer frem av kapittel 8 så er det ikke ønskelig med en videre utvikling før en generell genereringsalgoritme er implementert. Implementeringen av denne, basert på de viste reglene bør ha absolutt hovedfokus i et videre arbeid. Flere funksjonelle utfordringer kan implementeres uavhengig av genereringsalgoritmen, men det anbefales at denne kommer på plass først slik at det ikke dannes uønske avhengigheter. Implementasjon av genereringsalgoritmen vil også være med på å avdekke flere funksjonelle behov enn det som er beskrevet her.

10.1.2 Kobling mot eksisterende system

DiaMODL modellerer brukergrensesnitt med tilhørende logikk. Det programmet som blir generert i DiaModlGen er en realisering av dette. Verken DiaMODL eller DiaModlGen har direkte støtte for å koble grensesnittkoden mot andre program eller programkomponenter. Modellklassen representer på mange måter modellen i MCV, som der representerer data på en mer omfattende måte. Data er det som skal presenteres, og kan således bety et helt system, ikke bare en klasse. På sammen måte ville det vært verdifullt å kunne bruke den genererte modellklassen som et grensesnitt mot underliggende logikk som for eksempel et foretningslag eller en database. Dette er mulig med den genererte koden men må da gjøres manuelt. Det må utarbeides en metode som avgjør om en slik funksjonalitet skal kunne beskrives i DiaMODL-modellen eller om det er verdier som kan settes før generering av kode.

10.1.3 Generering for web

Web, eller internett, spiller en stadig større rolle. Mange programmer som før ble laget på tradisjonell måte utvikles i dag i stedet som websider. DiaMODL, eller DiaMODL-editoren, har ingen begrensning i så måte. Den kan like gjerne brukes til å modellere eller dokumentere webgrensesnitt som grensesnittet til et frittstående program. Utfordringer ligger i kodegenerering av dette. Hvordan DiaMODL-elementenes funksjon skal realiseres når det er snakk om objektorientert kode er det redegjort for, men det eksisterer ingen tilsvarende standard for web. En webside presenteres med i en nettleser og er i utgangspunktet statisk. Det finnes flere teknologier for å generere sidene dynamisk, samt fylle siden med data fra for eksempel en database. Hvordan en slik logikk skal bygges opp via en kodegenerering er avhengig av hvilken teknologi man velger å bruke, men det

mulig å se for seg html-sider som via kall (med valgt dynamisk språk) til et underliggende system realisere dette. Det kan gjøres på følgende måte[21]: Siden Java allerede er brukt vil det være naturlig å generere DiaMODL-elementene til tilsvarende komponenter i html. Disse kan kalle en generert Java Servlet som fungerer som kontroller. Servleten vil måtte styre både logikk og kommunikasjon mellom komponentene. Samtidig vil den måtte generere html-koden som skal vises for bruker når en beregning er gjort. En slik generering vil være utfordrende, og kan i liten grad baseres på funksjonaliteten som finnes i DiaModlGen i dag.

10.2 Funksjonelle utfordringer

I kapittel [7.6] ble det redegjort for spesialtilfeller funnet under utviklingen av DiaModlGen. I tillegg til de tilfellene som allerede er løst eksisterer det et sett med tekniske utfordringer. Med det menes utfordringer som når de er løst vil tilføre DiaModlGen mer fleksibilitet og sikker funksjonalitet. Utfordringene kan igjen deles opp i tekniske mangler og mer generelle utfordringer. Med mangler menes ikke mangler i form av at DiaModlGen ikke vil fungere uten dem, men tekniske løsninger som ikke er viktige for kjernefunksjonaliteten selv om de burde vært med. Utfordringene er teknisk funksjonalitet som helt eller delvis ikke er løst eller fokusert på da de falt utenfor problemstillingens kravspesifikasjon.

10.2.1 Feilsjekking

Koden presentert i kapittel 7 er fokusert på problematikken rund en kodegenerering. En klar mangel er feilsjekking i koden, noe som ikke har noe å si for funksjonaliteten. Det vil allikevel være viktig å kunne fange opp både feil introdusert av bruker under modellering og interne feil i programmet. Et eksempel er sjekk på tomme variabler.

```
Public void generateFrame(Model model) throws ModelElementException{
If (model != null){
…
} else throw ModelElementException("Empty object "+model.getClass());
```

Kodeliste 34 – Feilsjekking med unntaksmeldinger

[21] Det er ikke oppgavens intensjon å legge føringer på valg av teknologi, men det er på sin plass å presentere en mulig metode

Noe av variablene brukt under generering vil være tomme til data er hentet fra gjennomgangen av modellen. Er ikke disse satt vil i beste fall koden som blir generert ikke kompilere, og i verste fall vil man få feil under genereringen. Feil som tomme variabler vil i all hovedsak oppstå hvis parametere ikke er satt i modellen. En annen mulighet er at det innføres modeller som genereringskoden ikke enda har støtte for. Den vil da kunne feile i den form at den ikke vet hva som skal gjøres med det data som hentes. I begge tilfeller bør det da kunne gis tilbakemeldinger på til bruker, så denne vet hva som mangler.

En annen form for feilsjekk tilhører hører inn under god programeringsskikk. Det kan være at det sjekkes at lister ikke som traverseres ikke er tomme eller at man prøver å traversere over en listes lengde. Kodeliste 34 viser en sjekk på at et objekt man har hentet faktisk eksisterer. Slike sjekker vil minske sannsyneligheten for at feil oppstår under kjøring av programmet. Utfordringen ligger i hvor man skal legge sjekkene og hvordan eventuelle feil skal håndteres. Et naturlig valg vil være å lage et unntaksrammeverk (eng: exceptions), med et sett unntak som beskriver mulig problemer på best mulig måte. Javas egne unntak blir kastet når en feil oppstår. Disse bør fanges opp der det trengs og gjøres om til egenkonstruerte unntak som bedre beskriver feilene som er spesifikke for DiaModlGen. Riktig bruk av dette vil føre til at man kan programmets variabler og be bruker kjøre på nytt uten å måtte starte programmet på nytt.

En tredje mulighet som kan hjelpe utvikler er logging av hendelser i det kjørende programmet. Logging kan konfigureres til å gi meldinger ved spesielle hendelser, som for eksempel å skrive til en fil hver gang en metode blir kalt. Dette gir utviklere en mulighet til å se gjennom loggen for å se hvor eventuelle feil ligger. Logging er ofte til mest hjelp ved leting etter feil på et høyere nivå en direkte feilsjekking.

Et fjerde og siste alternativ er kvalitetssikring gjennom enhetstesting. Det er en metode som heller ikke går direkte på feilsjekking, men vil i tillegg til debugging være en metode som hjelper utvikler å finne logiske feil i en utviklingsfase. I utviklingsparadigmer som ”Ekstrem programmering” [2.6] er enhetstesting et av de viktigste momentene i utviklingsprosessen. Enhetstesting [eng: Unit tests] fungerer ved at man ved hjelp av et rammeverk programmerer metoder som tester programmets metoder ved å kalle dem og

sammenligne det resultatet som kommer tilbake med det utvikler mener resultatet skal være [Cheon, 2002]. Enhetstesting kan gjøres på alt fra enkeltmetoder til større programdelers funksjonalitet. Denne type testing på logisk funksjonalitet bør være et tillegg til feilsjekker i koden, men vil kunne gi utvikler mulighet til å verifisere at metoder gjør det de skal uten å måtte starte en ny instans av editoren. Enhetstester som dekker hver sin metode eller hver sin funksjonelle del av et program vil kunne hjelpe til med lokalisering av logiske feil. I tillegg til resultat fra metoder kan man også teste på innholdet i eventuelle unntaksmeldinger, noe som gjør det mulig å kombinere enhetstesting med et unntakshierarki.

10.2.2 Java unntakshåndtering (eng: exceptions)

Forrige avsnitt presentere unntakshåndtering, eller feilsjekking, som en del av DiaModlGen-rammeverket. En annen utfordring er å generere kode som i seg selv avhenger av å håndtere unntak. For eksempel kan dette være et kall til en standard javafunksjon som ved feil, for eksempel i en parameter, fører til et unntak. For at et kall til denne metoden skal fungere må selve metodekalle være omkranset av kode som håndterer unntakshåndteringen. I Java gjøres dette med en "try/catch"[22] blokk [W6]. Figur 29 i kapittel 4 viser en DiaMODL-modell av en nettleser realisert ved bruk av standard javakomponenter. Interaktoren som representerer nettleservinduet holder en javakomponent som viser frem en nettside bestem av hvilke nettadresser som blir sendt inn som parameter til denne komponenten. Kallet til funksjonen for å sette en ny adresse kaster et unntak (eng: throws exception) hvis det er en feil i den innsende nettadressen. Det medfører at hvis man generer kode basert på denne modellen, så vil nettleserkomponenten være avhengig kode som håndterer unntak rundt selve kallet for å sette adressen. Dette kan ikke den genererende koden vite, noe som kan håndteres på flere måter. Et eksempel er å automatisk legge inn en "try/catch" før alle metodekall, men dette fører til overflødig kode og gjør koden lite lesbar. Man vil også med den metoden måtte basere seg på standard unntaksmeldinger da forskjellige komponenter kaster forskjellige meldinger. Det sistnevnte er en sentral funksjonalitet i hendelseshåndtering. Hvis man da måtte overse at det finnes flere typer meldinger vil dette være et tap i forhold til hva Java tilbyr.

[22] Lar seg ikke direkte oversette, men på norsk er betydningen ”Prøv en eller annen handling, og hvis den feiler så fang opp feilmeldingen”.

En annen metode er å bruke mapperklassen og lage en oversikt over hvilke komponenter som trenger unntakshåndtering. Denne metoden vil fungere, men vil når man innfører nye komponenter medføre unødvendig ekstraarbeid og fare for feil.

En tredje metode er bruk av "Java reflection"[23] [W6] til å undersøke om klassen til de grafiske komponentene kaster et unntak, og eventuelt hvilken type unntak. Disse opplysningene kan brukes til å generere kode som passer til det kallet som skal gjøres. Denne metoden er den som vil kreve mest på forhånd å implementere. Men vil imidlertid får igjen for dette arbeidet, da lite ekstra trenger å bli gjort ved innføring av nye komponenter. Det er en klart mest generiske og elegante måten, men arbeidet med å gjøre det må vurderes opp mot hvilke behov man har. Viser empiri at behovet for å håndtere unntak er relativ lite kan en av de to første løsningen være den mest effektive.

10.2.3 Layout og innstillinger

Muligheten til å kunne styre utseende og ikke-standard parametere til genererte GUI-elementer er en av de mest elementære. Et program generert ut fra en modell vil fint fungere som en test på logikken som styrer kommunikasjon mellom komponenter, men vil samtidig være mangelfull med tenker på hva plassering og utseende har å si for interaksjon med bruker.

For at kodegenerering av DiaMODL-modeller skal kunne brukes til annet enn testing av logikk og interaksjon trengs en mer generell løsning både når det gjelder komponenters utseende og plassering. Det helhetlige programmets utseende må kunne styres i større grad samtidig som logikken beholdes. Dette vil i seg selv ikke nødvendigvis redusere det manuelle arbeidet i etterkant, men vil føre til at det arbeidet som må gjøres blir mer fokusert på spesialisering av funksjon og utseende som ikke kan styres i forkant.

Settes grafiske javakomponenter inn i en ramme (eng: frame) med standard styring av komponentplassering blir det etter hver tilfeldig hvor de havner i forhold til hverandre.

[23] Undersøkelse av javaobjekter i kjøretid. Returnerer opplysinger om det inspiserte objektet.

Spesielt i modeller med et større antall komponenter vil plasseringen fort bli lite oversiktelig eller ulogisk hvis man ikke på en eller annen måte kan styre utsende og plassering under eller etter genereringen. I mindre modeller, som eksemplene som denne oppgaven viser, er sjansen større for at det genererte utseende blir bra nok til å kunne brukes for testing. Allikevel vil det antakelig være ønskelig å kunne styre plasseringen i enkelte tilfeller. Dette kan være grunnet testing av utseende mot faktiske brukere, eller at det komponenter må plasseres på en spesiell måte for å få en ønsket visuell effekt.

Når det gjelder hver enkelt komponents utseende tilbyr Swing et sett med standardverdier som kunne vært brukt i større grad. En slik løsning vil fortsatt være statisk og like lite heldig som et manuelt arbeid i etterkant.

Java har klasser som tilbyr forskjellig plassering av komponenter [Swing i W6]. Dette kan være relativt i forhold til andre komponenter, eller mønster som delvis kan styres manuelt. I et videre mulig arbeid vil det være mulig å implementere bruk av disse klassene. Fire mulige metoder kan skisseres.

I modelleringsverkøyet kan man setter parametere på de forskjellige komponentene. Man kan se for seg en utvidelse av antall parametere som styring komponenters utseende, standardverdier og plassering. Utseende og standardverdier er det som er mest aktuelt da plassering kan være utfordrende å beskrive. Den enkleste metoden vil være å sette en nummerering på hver komponent som kan styre en form for rekkefølge. Utfordringen med en slik utvidelse vil være at den må implementeres i DiaMODL-editoren i stedet for DiaModlGen. Dette fører til en uønsket tett kobling.

En annen metode, antakelig den minst funksjonelle, er allerede nevnt, nemlig å sette flere standardverdier i mapperklassen. Dette må kombineres med logikk som styre utseende ved hjelp av parametere når genereringen blir kjørt. Denne metoden kan og bør brukes kun i mindre grad. Eventuelt kan den kombineres med andre metoder.

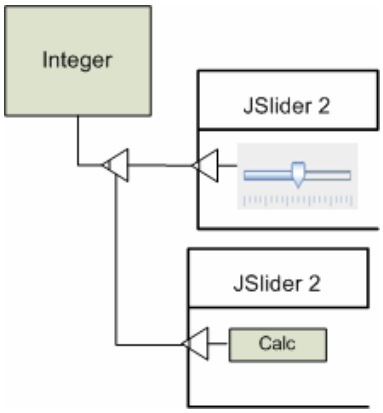
Den tredje metoden er å utvide den genererende koden til å ta hensyn til maler i form av design beskrev i egne filer, for eksempel XML. En slik fil vil kunne brukes på to måter. Det første angår utseende, da man kan referer til modellobjekter i filen. Parametere kan

bli satt og lest inn under generering. Det mest anvendelige med en slik løsning vil være å kunne tilby et sett med layouter som man basert på antall komponenter velger. Går man enda lenger.

Det siste løsningen lar seg kombinere med metoden over, og består av en meny i DiaMODL-editoren. Menyen gir mulighet til å velge både parametere og maler for utseende. Dette er funksjonalitet som finnes i de fleste programmer i dag. En slik løsning, eventuelt sammen med maler, vil være den mest generiske, men samtidig medføre mest jobb.

10.2.4 Åpne / lukke knapp

En form for modellering som ofte er i bruk både i programmer og i DiaMODL-modeller er knapper som styrer funksjonalitet. Med funksjonalitet menes i dette tilfellet tilstander eller hendelser. En knapp er en boolsk komponent. Den kan enten har verdien trykket inn eller ikke trykket inn (av eller på). I tillegg kan den initiere en hendelse. Hendelsen er noe som skjer umiddelbart når knappen blir aktivert. Både den boolske funksjonaliteten og aktiveringen medfører to utfordringer som må håndteres på hver sin måte.



Figur 73 – "Åpne / lukke" knapp

Figur 73 viser en knapp som er satt på koblingen mellom en interaktor og en modellrepresentasjon. Basert på måtene en knapp fungerer på kan denne modellen vise to ting. Enten at knappen brukes til å eller stenge for koblingen mellom interaktoren og modellen. Det vil si at knappen i en tilstand (for eksempel trykket inn) så er koblingen åpne, og verdien fra interaktoren vil settes i modellen hver gang det sendes en ny verdi fra interaktoren. Står knappen i steng stilling, vil ikke verdien settes i. Den andre

funksjonaliteten medfører at verdien interaktoren sender fra seg vil settes i modellen i det øyeblikket knappen trykkes. Det vil si at man først endrer den grafiske komponentens verdi for deretter å trykke på knappen.

Dette er to forskjellige utfordringer som må løses på hver sin måte. En metode for lukking og stenging av porter er allerede presentert og vil kunne brukes for å imitere funksjonaliteten til en av-eller-på knapp. Utfordringen ligger i at en knapp i selv er en grafisk komponent i en interaktor. En inngående port med en updatemetode i denne interaktoren vil måtte sette status i stedet for verdi. Dette lar seg løse ved å sjekke på hvilken type komponent som genereres. I figuren åpner og lukker knappen for kommunikasjon ut fra en annen interaktor. Det vil i praksis si at actionmetoden i knapp-interaktoren vil måtte sette variabelen som bestemmer om en porter er åpen eller ikke i den andre interaktoren.

En større utfordring er å realisere en knapp som initierer en hendelse. Interaktoren som implementer knappen vil ikke noen inngående port da knappen bare har en status. En mulig sekvens er at noe ikke skjer før knappen trykkes. Det vil si at den utgående metoden i knapp-interaktoren kaller actionmetoden i interaktoren som sender fra seg data. Dette er en mulig måte, men er ikke heldig da den fører til spesialtilfeller og uønskede avhengigheter mellom komponentene.

11 Konklusjon

I kapittel 9 ble det presentert resultater basert på et sett løsningskrav. Det var et løsningskrav det ikke ble redegjort for, nemlig kravet om at det med oppgavens arbeid skal kunne identifiseres en merverdi. Dette ble ikke bare spesifisert i problemstillingen, men har vært et gjennomgående krav både i innledning, teori og faktisk arbeid.

En implementasjon basert på hypoteser utledet i en problemstilling er ikke et komplett arbeid. Det må utledes en generell metode basert på funn gjort i implementasjonsfasen. Dette har blitt vist i denne oppgaven, men selv den en generell metode er lite verdifull hvis den ikke medfører økt bruksverdi. Den må også være av interesse for målgruppen som adresseres. Gjøres det en jobb ingen er interessert i vil jobben være bortkastet. Det er på bakgrunn av dette at en merverdi må kunne identifiseres. For oppgavens del er merverdien todelt. Først og fremst gjenspeiles den i valg av teoretisk grunnlag. DiaMODL og DiaModlGen er ment å passe inn i, og tilføre noe til, metodene og systemutviklingsparadigmene teorien redegjør for. Punktene under oppsummerer dette:

Roller

Rollene det fokuseres på i oppgaven er i all hovedsak designer (spesielt interaksjonsdesigner) og utvikler. DiaMODL og er i hovedsak et verktøy for førstnevnte rolle. En nedkorting av avstanden mellom rollene vi føre til økt kvalitet og kommunikasjon i et utviklingsscenario. Denne funksjonaliteten og merverdien tilføres av DiaModlGen

Prototyping

Prototyping hører til feltet interaksjonsdesign, men kan også brukes om aktiviteter innen systemutvikling. DiaMODL gjør det mulig å prototype brukergrensesnitt og interaksjon. DiaModlGen gjør det mulig å videreføre design til funksjonelle prototyper. Disse kan utvikles videre, og fungere som selvstendige prototyper.

Brukersentert utvikling

DiaMODL gjør det mulig å inkludere bruker på en bedre på måte på et tidligere tidspunkt. Muligheten til å generere kode, og vise en kjørbar utgave av programmet gjør

denne prosessen komplett. Bruker får mulighet til å raskere se en kjørbar utgave av programmet.

UML

UML er standard dokumentasjons- og spesifiseringsverktøy i en utviklingsprosess. DiaMODL baseres på UML, men utvider funksjonaliteten i form av støtte for interaksjonsmodellering. DiaModlGen relaterer ikke direkte til UML, men er et verktøy som gjør det mulig å bruke DiaMODL og UML om hverandre.

Wisdom

Wisdom var tidlig i arbeidet med oppgaven en mal for hvordan en realisering av DiaMODL-modeller kunne bygges opp. Tankegangen i Wisdom virket ideell, men det er vist at DiaMODL og DiaModlGen til sammen utgjør noe Wisdom mangler, nemlig støtte for modellering av interaksjon og realisering av disse.

Beskrivelsen av DiaMODL utgjør oppgavens største teoridel. Den er også den klart viktigste da DiaModlGen bruker DiaMODL som utgangspunkt. Det er i forholdet mellom DiaMODL og DiaModlGen den viktigste formen for merverdi er synelig; Brukerne av DiaMODL vil være de som får mest ut av innføringen av DiaModlGen. Funksjonaliteten og mulighetene for integrasjon blir drastisk utvidet. På en annen side så må merverdien spesifiseres ut fra resultatene som er faktisk er oppnådd og redegjort for i forrige kapittel. Først og fremst var hensikten å utvide DiaMODLs funksjonalitet, noe som er oppnådd. Det ble presentert en funksjonell kodegenerering, og koden lot seg også utlede i regler som en generell genereringsmetode må følge. Det kan på bakgrunn av det konkluderes med at oppgavens løsingskrav er tilfredsstilt. Integrasjonen med DiaMODL-editoren kunne vært tettere, og mer funksjonalitet kunne vært implementert, men dette er funksjonelle utvidelsers som ikke forandrer innholdet av problemstillingen og tilhørende løsning. Bedre valg av eksempel ville gjort det mulig å utlede en generalisering på et tidligere tidspunkt, men det er ikke sikkert generaliseringen ville blitt noe annerledes.

Oppsummert vil det si at det er mulig å realisere oppførselen i en DiaMODL-modell ved å generere kode. På den måten tilføres merverdi til både systemutviklingsprosess og brukere av DiaMODL, noe som var oppgavens mål. En videreutvikling og faktisk bruk

av DiaModlGen vil understreke dette, og forhåpentligvis vil DiaModlGen stimulere til økt bruk av DiaMODL, noe som i seg selv er en merverdi.

12 Litteratur

Artikkler og Bøker

(Trætteberg 1, 2002) - Trætteberg, H: "Model-based User Interface Design", NTNU, 2002.

(Trætteberg 2, 2002) - Trætteberg, H: "Dialog modelling with interactors and UML Statecharts - A
hybrid approach", NTNU, 2002

(Trætteberg 1, 2003) - Trætteberg - "Integrating Dialog Modelling and Application Development", NTNU, 2003

(Chandler, 1998) - Nonaka, I., Takeushi, H. "*A theory of the firm's knowledge-creation dynamics*. In "The dynamic firm. The role of technology, strategy, organization and regions." Chandler jr, A.D, Hagstrøm, P., Søvell, Ø. (eds). Oxford University Press, 1998.

(Nunes, 2000) - Nunes, N.J. og Cunha, J. F.:
"Wisdom - A UML based architechture for interaktive systems", s.191-205, i "Proceedings of the 7th International Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS2000)", Patrno, F., Palanque, P. (Eds.), Springer-Verlag , New York,
2000

(Nunes, 2001) - Nunes, N.J. - Object Modelling for User-Centered Development and User Interface Design - The Wisom Approach, 2001

(Coyle, 1994) - Coyle, F.P. (Dept. of Comput. Sci. & Eng., Southern Methodist Univ. "Smalltalk's Model-View-Controller architecture, Dallas, TX, USA)" Proceedings of OBJECT EXPO '94, 6-10 June 1994 , New York, NY, USA

(Leff, 2001) R11: Leff, A. (IBM Thomas J. Watson Res. Center, Hawthorne, NY, USA); Rayfield, J.T. "Web-application development using the Model/View/Controller design pattern"Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001, p 118-27

(Carnell, 2003) - Carnell, Linwood, Zawadzki
"Professional Struts Applications: Building Web Sites with Struts, Ojb, Lucene and Velocity" APress: 2003.

(Burbeck, 1992) - Burbeck,S (Ph.D): Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC), 1992.

(Gulliksen, 2001) - Gulliksen, Göransson og Lif: A User Centred Approach to Object Oriented UI Design. Centre for User Oriented IT Design, Royal Institute of Technology, Sweden, 2001

(Braaten, 2006) - Braaten, Anders: ProtoMODL - Interaksjonsmodellering i brukergrensesnitt,
IDI, NTNU 2006 (Draft)

(Campos, 2004) - Campos P. F og Nunes N. J.: "A UML-Based Tool for Designing User Interfaces", i "UML Modeling Languages and Applications: UML 2004 Satellite Activities Lisbon, Portugal, October 11-15, 2004 Revised Selected Papers", Springer-Verlag, 2004.

(Campos, 2003) - Campos, P.: "User-Centered Software Development supported by Analysis, Design and Modeling Tools", PhD Thesis Proposal, University of Madeira, 2003

(Toxboe, 2005) - Toxboe, A.: Introducing User-Centered Design to eXtreme Programming
Copenhagen Business School, Department of informatics, May, 2005

(Gutierrez, 1989) - Gutierrez, O: Prototyping techniques for different problem contexs. University of Massachusetts at Boston College of Management, 1989

(Hartvigsen, 1998) - Hartvigsen, G: Forskerhåndboken
Høyskoleforlaget, 1998

(Kreger, 2001) - Kreger, H: Web Services Conceptual Architecture (WSCA 1.0)
IBM Software Group 2001

(W3C, 2006) - State Chart XML (SCXML): State Machine Notation for Control Abstraction
W3C Working Draft 2006

(Cheon, 2002) - Y Cheon, GT Leavens: A Simple and Practical Approach to Unit Testing: The JML and JUnit Way ECOOP, 2002 – Springer

(Bambara, 2001) - Bambara, Allen: The J2EE Bible,
Unleashed, 2001 – Sams

(Beck, 1999) - Beck, Kent, Extreme Programming Explained, Addison Wesley, 1999

Websider

W1: Java Blueprints Model-View-Controller (Mangler)
http://java.sun.com/blueprints/patterns/MVC-detailed.html

W2: Eclipse Foundation
http://www.eclipse.org

W3: Wells D.: "Extreme Programming: A gentle introduction",
http://www.extremeprogramming.org/, 1999

W4: Agile Modeling – UML klassediagram
http://www.agilemodeling.com/artifacts/classDiagram.htm

W5: Object Management Group
www.omg.org

W6: Sun Java Tutorial
http://java.sun.com/docs/books/tutorial/reallybigindex.html

W7: Java Swing
java.sun.com/docs/books/tutorial/uiswing/

W8: OMG UML Resource Page
http://www.uml.org/

W9: Observer Observable pattern
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observable.html

W10: Java Integer Class
http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Integer.html

W11: Java Script Language Pnuts
https://pnuts.dev.java.net/

W13: Ulf Ottersrud, HiO – Binærer Trær
http://www.iu.hio.no/~ulfu/appolonius/e-bok/kap5/kap5.html

W14: Marinilli, M.: Swing and SWT: A Tale of Two Java GUI Libraries
http://www.developer.com/java/other/article.php/10936_2179061_1

W15: Trond Løvereide – Systemuvikling i UML
http://www.ia.hiof.no/~trondl/UML/pages/modul3.html

W16: Apache Commons SCXML
http://jakarta.apache.org/commons/sandbox/scxml/guide.html

W17: Metamodell
http://en.wikipedia.org/wiki/Meta_model

W18: Haas: Ten Guidelines for User-Centered Web Design
Usability Interface, Vol 5, No. 1, July 1998
http://www.stcsig.org/usability/topics/articles/ucd%20_web_devel.html

W19: Interaction modeling - based on the text ``Use Case Driven Object Modeling with UML, A Practical Approach'' by Rosenberg and Scott
http://www.devdaily.com/java/java_oo/node28.shtml

13 Vedlegg

Vedlegg på CD :

1. DiaModlGen kildekode

2. Komplett pakke med Eclipse, DiaMODL-editoren og DiaModlGen.