UNIVERSITETET I OSLO

Institutt for informatikk

Entydig modellering med connector og association forent i samme språk

Masteroppgave

(60 studiepoeng)

Tormod Vaksvik Håvaldsrud

01. Mai 2008

Sammendrag

I dagens samfunn ønsker vi å innføre programvaresystemer for å effektivisere og sikre kvaliteten på tjenester. Programvaresystemene blir større og større, og det stilles store krav til stabilitet og brukervennlighet. Dette gjør utviklingsprosessen så komplisert at mange systemer blir for dyre til at de blir realisert. Introduksjon av associationer, connectorer og parter i modelleringsspråk eller programmeringsspråk vil gi utviklere et enda bedre verktøy i utvikling av programvaresystemer. Dette kan gi økt kvalitet og produktivitet, som igjen vil redusere kostnadene.

Disse begrepene har opphav innen modellering, og vi har ikke så lange tradisjoner på å tolke de entydig som vi har med logiske konstruksjoner som if og while.

I denne oppgaven har vi laget et modelleringspråk hvor vi bruker associationer, connectorer og parter for å modellere programvaresystemer. Modellene blir entydige og kan oversettes direkte til kjørbar kode.

Vi har kommet frem til en måte å forene associationer, connectorer og parter slik at de utfyller hverandre og fremstår på en helhetlig måte. Vi har også kommet med en beskrivelse av hvordan associationer kan spesialiseres og vi har innført nestede associationer for å kunne modellere spesielle egenskaper ved systemet.

Takk til

For det første vil jeg takke veilederen min, Birger Møller-Pedersen, for all hjelp og støtte han har gitt meg. Han har gitt meg god hjelp i skriveprosessen og kommet med konstruktiv kritikk til oppgaven. Vi har hatt mange interessante samtaler om modellering, og han har gitt meg muligheten til å dra nytte av hans inngående kunnskap og lange erfaring.

Jeg vil takke kona mi, Mariann Vaksvik Håvaldsrud, for tålmodighet og interesse for mitt arbeid med masteroppgaven. Hun har vært en god støtte gjennom intense arbeidsperioder og har gitt ortografisk og språklig hjelp til oppgaven.

Jeg vil også takke Olav Skjelkvåle Ligaarden for sjenerøsitet når det gjelder å lese og gi konstruktive innspill til oppgaven under skriving. En stor takk går også til familie for langvarig støtte til skolearbeid. Dette har hatt stor betydning for at jeg har kommet dit jeg er i dag.

Innhold

1.0 Innledning

I dagens samfunn tar vi i bruk programvaresystemer til mange formål som vi tidligere løste med annen teknologi. Programvaresystemene blir større og bruken blir hyppigere. Dette setter store krav til systemene vi utvikler i dag. De skal være brukervennlige, pålitelige, forutsigbare, omfattende, modulære, utvidbare og billige. Det er nesten ikke grenser for hvilke krav som settes til et programvaresystem i dag. Disse kravene kommer i tillegg til kravet om funksjonalitet som ofte er en stor utfordring for programvareutviklerne.

Årsaken til feil i systemer er ofte at endringer i koden fører til uforutsette resultater i systemene. Når de store linjene og konstruksjonene blir lite tydelig blant store mengder detaljer er det ikke lett å holde oversikt til enhver tid. Dokumentasjon over systemet er sjelden oppdatert under utvikling. Grunnen er at systemet stadig endrer seg og at det koster å ha noen til å gjøre den kontinuerlige oppdateringen.

Unified Modelling Language (UML)[6] er et modelleringsspråk som ofte brukes for å planlegge programvaresystemer. I UML er vi vant til å modellere programvaresystemer med klasser og associationer. Vi erfarer at klasser er gode til å beskrive egenskaper til objekter i programmet, og dette har ført til at vi har fått klasser i programmeringsspråk. Associationer har blitt brukt suksessfullt i mange år. Årsaken til det er at de beskriver hvordan klasser samarbeider og forholder seg til hverandre på en naturlig måte. Dette taler for at vi bør ta i bruk associationer i programmeringsspråk, for dermed å kunne beskrive programvaresystemer på en enda bedre måte.

Detaljer som programvaresystemet er bygget av bør beskrives på et riktig abstraksjonsnivå, slik at detaljer ikke dekker over de store linjene. Om vi skal beskrive hvordan et kraftverk virker er det lite hensiktsmessig å bare bruke planker og spiker, mens det er nettopp planker og spiker som er naturlig å bruke i en byggebeskrivelse til en fuglekasse. Vi må kunne beskrive systemet med elementer som har de egenskapene som vi ønsker å bygge systemet med.

Slik har Rumbaugh[2] uttalt seg om dette: ”It is possible to program relations using object-oriented constructs, but only by writing a particular implementation in which the programmer is forced to specify details irrelevant to the logic of an application, it is not possible to separate the abstraction from the implementation with the same clarity as found in the relational data model.” I sin avhandling argumenterer Rumbaugh for å introdusere nettopp relasjon som en primær byggestein (first class construct) i objektorientert programmeringsspråk på lik linje med konseptene klasse og spesialisering. Mange har identifisert dette problemet med vanlige objektorienterte programmeringsspråk. Bierman og Wren[3] sier det på denne måten: ”However, whilst object-oriented languages easily represent real-world entities (e.g students, lectures, buildings), the programmer is poorly served when trying to represent the many natural relationships between those entities (e.g ‘attends lecture’, ‘is taught in’).”

En introduksjon av associationer i objektorienterte programmeringsspråk kan fylle mye av det gapet vi har i dag mellom modellering og programmering av programvaresystemer. I stedet for å skrive gjentagende kode, vil vi kunne gjenbruke testet og feilfri kode. Dette vil gi utviklerne mer tid til å lage funksjonalitet i stedet for å programmere deler av den samme logikken om og om igjen som innimellom fører til feil.

I en tradisjonell iterativ utviklingsprosess blir først kravene for systemet spesifisert. Etter dette begynner valg av arkitektur og utforming av skisser over systemets deler, og ansvarsområder blir fordelt. Resultatet av denne delen av prosessen er ofte en mengde UML-diagrammer og dokumenter som beskriver systemets oppbygning og funksjonalitet slik at systemet skal ivareta de spesifiserte kravene. Det neste som står for tur er selve byggingen av systemet. Dette gjøres som oftest ved å lese dokumentasjonen og UML-diagrammer for så å programmere funksjonaliteten i henhold til beskrivelsene. Når systemet er ferdig må det dokumenteres og de initielle UML-diagrammene må justeres og detaljeres etter hvordan systemet er implementert. Dokumentasjonen og brukerveiledinger er ofte en mer folkelig og visuell beskrivelse av krav utformet tidligere i prosessen.

Som vi ser blir UML-diagrammer utformet i forkant for å fortelle hvordan systemet skal være, og i etterkant blir det utformet mange UML-diagrammer for å beskrive hvordan systemet i virkeligheten ble. Mye av dette arbeidet er nødvendig for å vinkle informasjonen riktig, men vi har alle opplevd at brukermanualer og sekvensdiagrammer forklarer hvordan forrige versjon var og ikke den versjonen vi har nå. Det er et stort problem at UML-diagrammer ikke er oppdaterte under vedlikehold og utvidelse av systemer.

Mange feil oppstår fordi overføring av informasjon fra ett nivå til et annet er manuell. Dette gjelder ajourføring av arkitektur, design, og dokumentasjon når systemet endrer seg. Denne manuelle overføringen av informasjon introduserer unødvendige menneskelige feil og er årsaken til mange misforståelser og kostnader.

Mange av disse problemene kan løses på en bedre måte om vi benytter oss av modellbasert systemutvikling (MDA). Når arkitekturdiagrammene og realiseringen av selve systemet er koblet, blir det mulig å automatisk sjekke at integriteten er ivaretatt og at modellen er konsis. Sporbarhet gjør det mulig å finne ut at deler av systemet er endret, og at dette bør føre til oppdateringer i dokumentasjonen for berørte områder. Blant annet kan brudd på arkitektur detekteres, samtidig som modellering gir mulighet til å beskrive systemer på et mer riktig abstraksjonsnivå.

Thomas Meservy og Kurt D. Fenstermacher omtaler MDA på denne måten [1]: “The Model Driven Architecture initiative shifts the focus of software development from writing code to building models.”

Når fokuset skal flyttes fra å utvikle kode til å bygge modeller, er det naturlig at både språkene, metodene og verktøyene må tilpasses nye behov. I denne sammenhengen blir det naturlig at modellene blir mer omfattende og vil inneholde detaljer som til nå kun har vært i koden.

Med disse tankene i bakhodet startet vi med denne oppgaven.

Oppgaven er strukturert slik at vi først definerer problemstillinger i kapittel 2.0. Problemstillingene blir synliggjort med en del eksempler som aktualiserer problemstillingene. Etter problembeskrivelsen beskriver vi metoden. I kapittel 3.0 presenterer vi flere eksempler som skal være et verktøy for å teste ut modelleringsspråket som blir beskrevet i detalj i kapittel 4.0. Etter beskrivelsen av modelleringsspråket får vi tre kapitler som gjennomgår løsninger eller modeller av eksemplene for å vise at modelleringsspråket er i stand til å favne utfordringene som eksemplene byr på. Til slutt har vi en konklusjon i kapittel 8.0 som sammenfatter hva vi har kommet frem til og retter blikket mot gjenstående arbeid.

2.0 Problemstillinger og metode

Som nevnt har flere foreslått å introdusere associationer i objektorienterte programmeringsspråk. Et annet alternativ som vil fungere like bra er å lage et entydig modelleringsspråk med associationer til bruk i modellbasert systemutvikling. En entydig modell kan i etterkant transformeres til et kjørbart program eller kjøres direkte i en runtime-motor. Denne tankegangen ligger tett opptil tanken om et universelt og generelt modelleringsspråk.

I denne universelle og generelle tilnærmingen søker man å finne en god og presis beskrivelse av de grunnleggende byggesteinene vi bruker for å beskrive programvaresystemer. Med en slik tilnærming kan vi dekke mange domener med ett modelleringsspråk som kan brukes til å lage nesten hva som helst. Ulempen er at det ofte vil oppstå dissens om hva som er grunnleggende byggesteiner og hva disse betyr. Dette fører til at det er både kostbart og tidkrevende å komme frem til et slikt modelleringsspråk, men når man er i mål kan det til gjengjeld brukes i mange sammenhenger og er av stor verdi. UML er et kjent og viktig bidrag i arbeidet med å finne et slikt generelt modelleringsspråk. Denne oppgaven har tatt utgangspunkt i UML og er ment som et bidrag til prosessen for å lage et slikt språk. Selv om UML er stort og mye brukt ser vi at språket har som mål å være fleksibelt og at det har fokus på å kunne modellere hva som helst, fremfor å lage en entydig tolkning av modellene.

Modellbasert systemutvikling har som mål at modellene kan kjøres eller at de kan transformeres til kjørbar kode. Dette krever at modellene er entydige. Programmeringsspråk har vært nødt til å kreve en entydig tolkning av hver minste detalj for å kunne lage kjørbar objektkode. Når modellene nå skal leses av datamaskiner, kreves det en like stor entydighet innen modellering som det vi er vant med innen programmering. Derfor må vi finne en entydig tolkning av UML-elementene for at vi skal kunne bruke UML til modellbasert systemutvikling.

UML definerer flere forskjellige relasjoner mellom klasser; ”association”, ”aggregation” og ”composition”. Disse er sjelden kost blant dagens objektorienterte programmeringsspråk, og de implementeres ofte i programvaresystemer med en form for referanse. Når vi skal automatisere denne implementasjonen er det viktig å ha full forståelse av hva de forskjellige relasjonene er. For det er først når vi bestemmer oss for en tolkning og forstår denne, at vi virkelig kan utnytte dens styrker og muligheter i modeller som skal leses av maskiner.

I UML ble composite-strukturer introdusert i versjon 2.0, og dette ga oss enda en mulighet for å koble sammen objekter. En composite-struktur er en beskrivelse av hvordan en klasse er bygget opp. En klasse kan bestå av flere parter og disse partene kan kommunisere via connectorer. Partene er samlinger av objekter, og connectorene inneholder informasjon om hvilke objekter i en part som er knyttet til objekter i en annen part.

Selv om prinsippet med connectorer er ganske annerledes enn prinsippet med associationer, så kobler begge sammen objekter slik at objektene kan kommunisere og samarbeide. I denne oppgaven vil vi se på forskjeller

og likheter mellom disse prinsippene og forsøke å finne en felles plattform for dem.

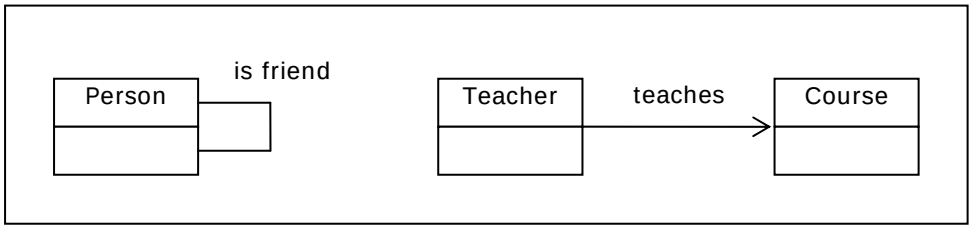
Selv om association ikke er noe nytt, så finnes det fortsatt flere tolkninger av den. Vi tenker her spesielt på spesialisering av associationer. I denne oppgaven er vi interessert i å finne en entydig tolkning av spesialisering av associationer og hvilke konsekvenser dette har for modellering og implementasjon.

For enkelhets skyld har vi i denne oppgaven bare omtalt associationer og connectorer med to ender. Dette er den desidert vanligste bruken av disse begrepene.

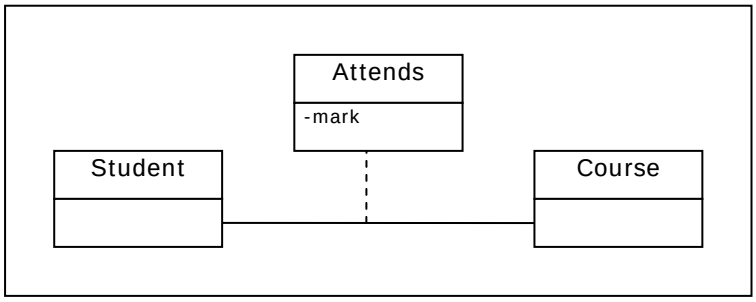
2.1 Association

En association er en beskrivelse av et forhold mellom to klasser. Som vi ser av Figur 2.1 og Figur 2.2 snakker vi om helt vanlige relasjoner som omgir oss til daglig. Begrepet venn beskriver noe om forholdet mellom to mennesker, og det er slike relasjoner vi bruker associationer for å beskrive. En klasse beskriver substantivers fellestrekk, som at alle baller har en diameter, en tyngde og en farge. Associationer beskriver relasjoner mellom substantiver, som for eksempel vennskap, lidenskap og eierskap. De aller fleste relasjoner kan beskrives fra det ene substantivet til det andre enten via et verb eller en verbfrase. Som en persons eierskap over en hund, kan beskrives med verbet “eier”. Vi sier: “Personen eier hunden.”

Booch, Jacobson og Rumbaugh[6] skriver: “In the same sense that classes correspond to nouns, associations corresponds to verbs.” Ambler [4] mener at associationer har navn som tilsvarer verb eller verbfraser. Vi ser et eksempel på dette i Figur 2.1 med associationen ”is friend”.

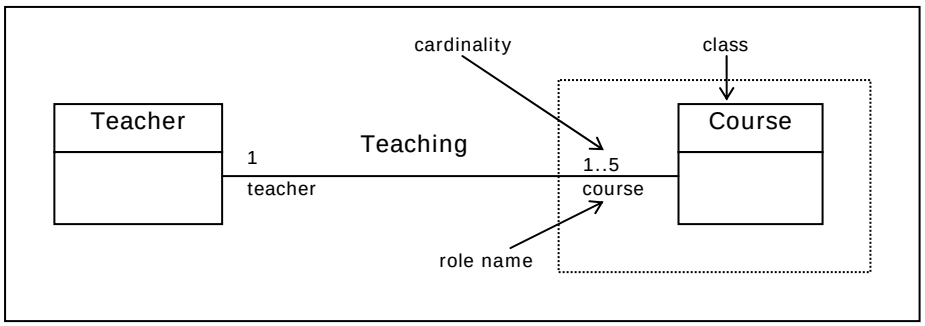


Figur 2.1 Eksempler på associationer



Figur 2.2 Eksempel på association med associations-klasse

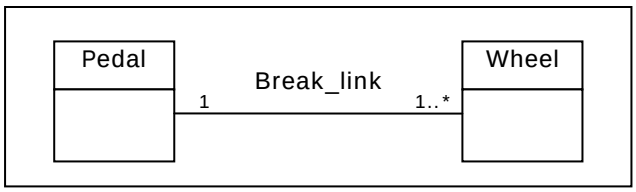
Instanser av associationer heter linker og minner mye om objekter. I tillegg til informasjonen om hvilke objekter som har et bestemt forhold til hverandre, kan en link inneholde informasjon om selve forholdet. Denne informasjonen blir definert ved en associations-klasse. Da vil linken være en instans av denne associations-klassen. Som vist i Figur 2.2 lagrer linker av associations-klassen Attends informasjonen om hvilken karakter studenten har i dette faget.



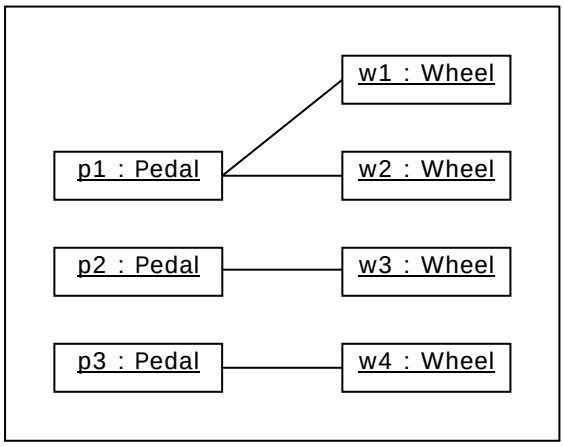
Figur 2.3 En association med navn, rolle og kardinalitet

Figur 2.3 viser et eksempel på at en association kan navngis nøytralt i forhold til rollene samtidig som den beskriver forholdet klassene imellom. Når vi ser på navngivningen i Figur 2.1, ser vi at associationen ”teaches” ikke er nøytral men at den er navngitt med hensyn på leseretning. På engelsk får vi setningen ”Teacher teaches course” når vi leser fra venstre mot høyre. Pilen på associationen ”teaches” i Figur 2.1 sier hvilken vei det er ment at associationen skal være navigerbar. Det vil si at et Teacher objekt skal vite om ”Course” objektene men ikke motsatt. I denne oppgaven vil vi prøve å ha nøytrale navn og heller bruke rollenavn i logikk. Vi ser også for oss en nøytral implementasjon som tilbyr informasjon om linkene. Det vil si at begge sider har tilgang til informasjon om den andre siden ved å spørre associationen. Associationen blir definert som en klasse og alle som klassen er synlig for vil ha tilgang til linkene. I eksemplet i Figur 2.3 vil en link bli inneholdt av en association som heter ”Teaching”, og linken vil inneholde to referanser som henholdsvis heter ”teacher” og ”course” som vil ha referanser til et objekt hver.

Associationen har to ender som representerer hver sin rolle. Den stiplete firkanten indikerer den ene av de to rollene associationen Teaching har. Denne rollen har tre egenskaper ”role name”, ”cardinality” og ”class”. ”role name” forteller hvilket navn denne rollen har og ”class” forteller typen til instansene i denne rollen. ”cardinality” beskriver kort sagt kardinaliteten til mengden av instansene i denne rollen og er et intervall i den naturlige tallmengden. Kardinaliteten gjelder for en gitt instans i den andre rollen til associationen. Et gitt Teacher objekt vil kunne være linket med 1 til 5 Course objekter, mens et gitt Course objekt vil bare kunne være linket til ett Teacher objekt. Det vil si at en lærer må undervise i minst ett kurs og kan maksimalt undervise i 5 kurs. Derimot blir et kurs undervist av én og bare én lærer.

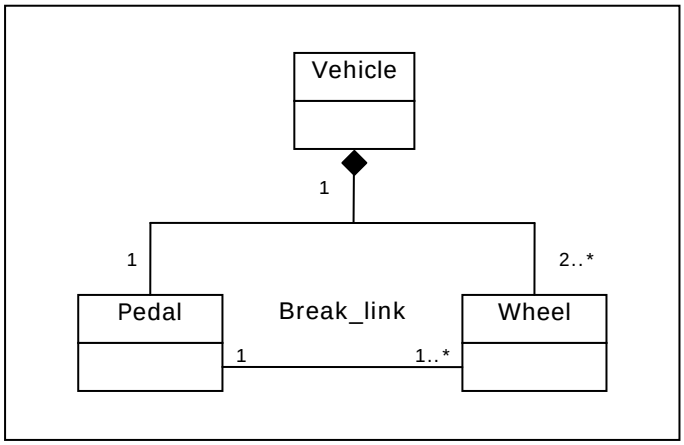


Figur 2.4 Eksempel på en association mellom klassene Pedal og Wheel



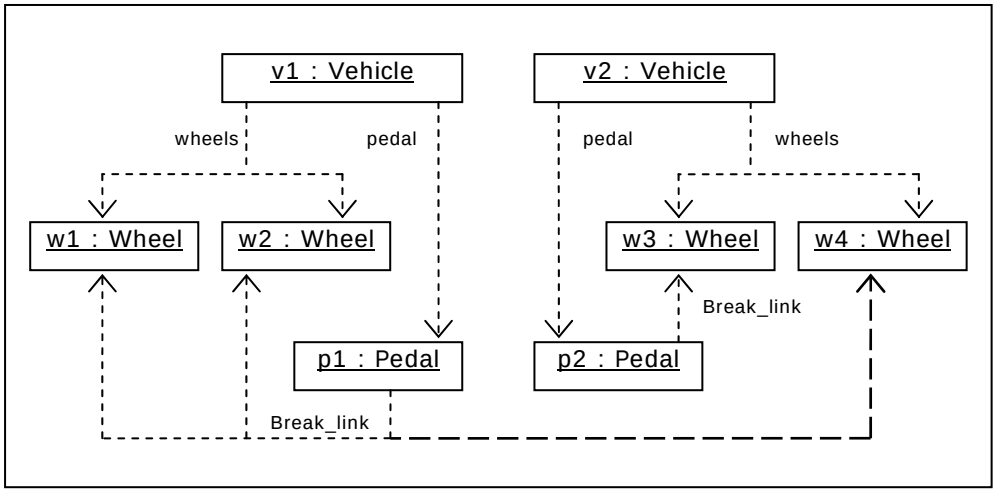
Figur 2.5 En bipartitt relasjon av linker

Figur 2.5 viser en tenkt situasjon med linker av associationen skissert i Figur 2.4. Strekene illustrerer linker. Hver av linkene linker sammen ett objekt av klassen Pedal og ett objekt av klassen Wheel. Vi ser at linkene danner en bipartitt relasjon som følger restriksjonene definert av associationen. Pedal-sidens noder har grad innenfor intervallet [1..*] og alle nodene på Wheel-siden har grad 1. Det er derimot ikke definert om denne relasjonen skal være enkel eller ikke, det vil si om det kan eksistere flere linker mellom f.eks. objekt p1 og w1. I UML kan vi åpne for flere linker mellom to gitte instanser med å si at en association ikke er unik.



Figur 2.6 Klassediagram over composition

I Figur 2.6 ser vi at Vehicle har ett Pedal objekt og to eller flere Wheel objekter. Associationen Break_link sier at hvert Pedal objekt kan være koblet til mange objekter av klassen Wheel. Når vi mennesker tenker på dette tar vi det som en selvfølge at Pedal objektet skal være koblet til Wheel objektene som eies av samme Vehicle objekt, men det sier ikke dette diagrammet noe om. Vi kan få en situasjon som vist i Figur 2.7, hvor p1 er koblet til w4 som ikke tilhører v1 men v2.

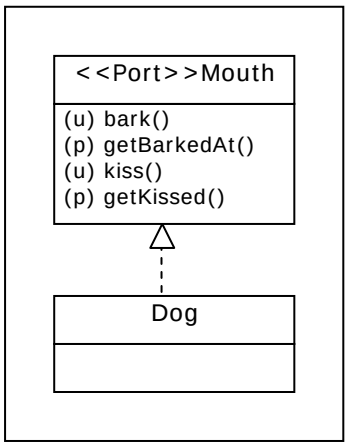


Figur 2.7 Objektdiagram over uønsket situasjon

Vi ønsker selvsagt ikke at bruk av en pedal i et kjøretøy fører til at et annet kjøretøy bremser. For å beskrive denne restriksjonen i et klassediagram må vi ta i bruk Object Constraint Language (OCL)[7] eller lignende. For å slippe bruk av denne type verktøy kan vi beskrive denne relasjonen med et Vehicle objekt som kontekst. Det er dette som gjøres i composite-strukturer. Da er denne relasjonen beskrevet med en connector.

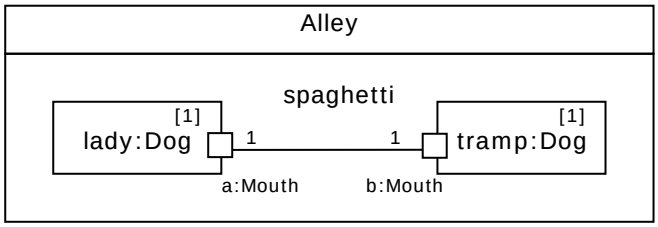
2.2 Composite struktur

Composite-struktur ble innført i UML 2.0. En composite-struktur er en beskrivelse med et objekt som omgivelse, og gjør det mulig å definere hvordan objekter forholder seg til hverandre og kommuniserer. Eller sagt på en litt annen måte så beskriver en composite-struktur hvordan objekter oppfører seg når de befinner seg inne i et annet objekt. En composite-struktur beskriver hvilke parter en klasse har og hvordan disse kommuniserer og samarbeider via connectorer.



Figur 2.8 Eksempel på en port

Figur 2.8 viser et klassediagram som viser at klassen Dog har porten Mouth.



Figur 2.9 Eksempel på en composite-struktur

Figur 2.9 viser et eksempel på en composite-struktur. Figuren forteller at klassen Alley har de to partene lady og tramp. Begge partene er av typen Dog og består av ett objekt hver. Siden partene er av typen Dog, må objektene være av typen Dog. Partene er koblet sammen med en connector som heter spaghetti. De små firkantene som heter a og b er portapplikasjoner av typen Mouth. Dette betyr at begge partene tar i bruk porten Mouth som er definert i Figur 2.8. Når partene bruker porten Mouth, kreves det at objektene også må ha denne porten. I dette tilfellet har alle objekter av klassen Dog denne porten.

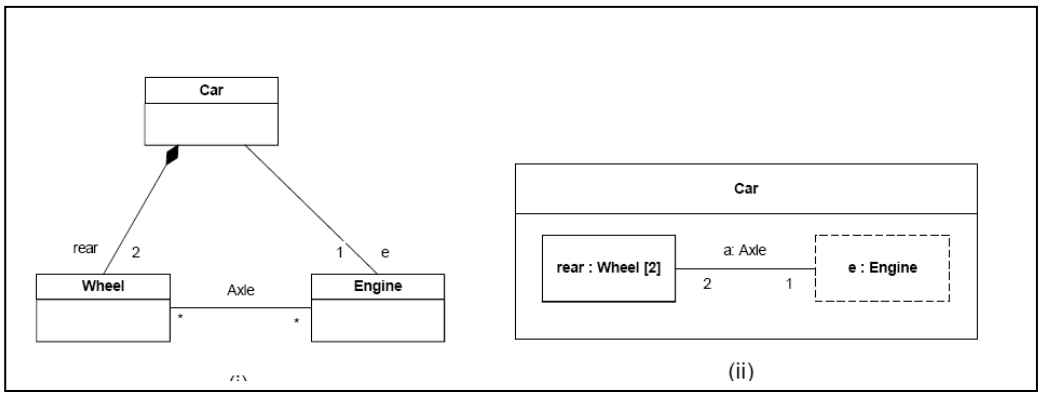
2.2.1 Part og Portface

Parter i UML opptrer i composite-strukturer og samarbeider for å nå et felles mål. Partene i en composite-struktur er koblet sammen av

connectorer, og de bruker disse for å kommunisere enten ved å sende signaler eller ved å påkalle metoder.

En part er en samling instanser med fellestrekk som opptrer på samme vis eller fyller samme rolle i denne strukturen. En part kan være av ”composition” eller ”reference” type.

Vi ser et par eksempler på bruk av parter i Figur 2.10 og Figur 2.11. Disse eksemplene er hentet fra UML-spesifikasjonen.



Figur 2.10 Eksempel på forhold mellom klassediagram og composite-struktur

En part av type ”composition” er i UML beskrevet som en part bestående av instanser opprettet internt, og eies av eieren av strukturen. Vi ser at Car eier 2 instanser av Wheel som igjen er ordnet slik at de utgjør parten rear. Parten e er av typen ”reference” og er da en part med referanse til en instans av Engine, som vist i Figur 2.10. En vanlig forståelse av UMLs parter er at objektene forblir i en part hele tiden, men i modeller der parten representerer hvilke roller et objekt fyller eller hvor et objekt befinner seg, vil det være naturlig at et objekt flyttes fra en part til en annen. For eksempel hvilken avdeling en pasient er innlagt på eller hvilken bøtte en ball ligger i.

Figur 2.11 Eksempel på multiplisitet

Partens multiplisitet beskrives som et intervall blant de naturlige tallene. Som vist i Figur 2.11 med to eksempler 4 og [1..2]. Generelt kan vi si at multiplisiteten beskrives med en nedre og en øvre grense for hvor mange instanser parten kan bestå av.

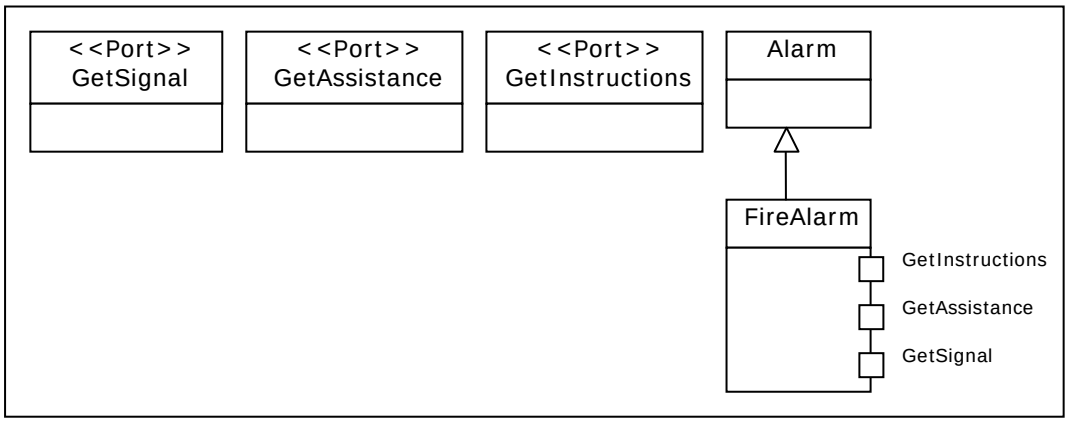
En port er en viktig bestanddel når vi skal finne ut hva en part i virkeligheten er. Når vi sier at en part er en samling instanser med felles egenskaper er det fordi vi ikke alltid beskriver en part med en type. Vi kan

beskrive en parts egenskaper med portApplicationer. Det vil si at vi beskriver hvilke porter en instans må ha for å kunne være en del av en part. Så vi kan si at i denne sammenhengen er portene viktigere enn selve typen til instansen. Dette kan minne om strukturlikhet og ikke navnlikhet.

En port beskriver hvilke interfacer porten tilbyr og hvilke interfacer som den forventer tilgang til. Bruk av interfacer i den ene eller andre retning foregår via connectorene som kobler partene sammen til en composite-struktur.

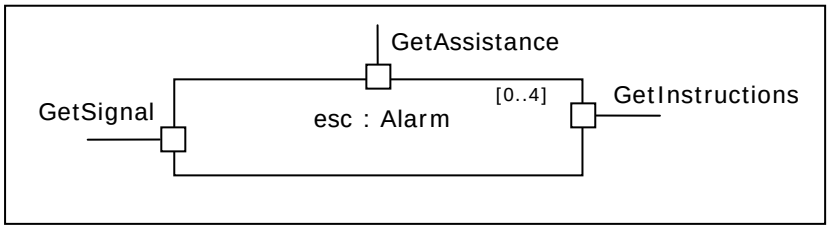
Ser vi på Figur 2.8 ser vi definisjonen av en port med navnet Mouth. Denne definisjonen sier at porten tilbyr metodene getBarkedAt og getKissed, samtidig som den bruker metodene kiss og bark. Det at den bruker en metode betyr at den forventer å få tilgang på denne metoden via en connector. Det vil si at i andre enden av en tilknyttet connector er det en part eller en port som tilbyr denne metoden.

En part vil kunne ha mange porter som til sammen fører til at parten spiller en relativt kompleks rolle i en composite-struktur. Det vil være praktisk å kunne beskrive en slik samling porter med et navn, akkurat som vi gjør med interface hvor vi gir en samling operasjoner et navn. Når parten blir beskrevet med en gitt samling porter, må alle instanser som skal være en del av denne parten støtte denne navngitte samling av porter. Om en part har ti porter må vi sjekke om en instans har alle disse ti portene før vi vet om den kan være en del av parten. Hvis vi derimot har en navngitt samling av de ti portene trenger vi bare å sjekke om instansen er av denne ene typen. Nå skal vi se et lite eksempel:



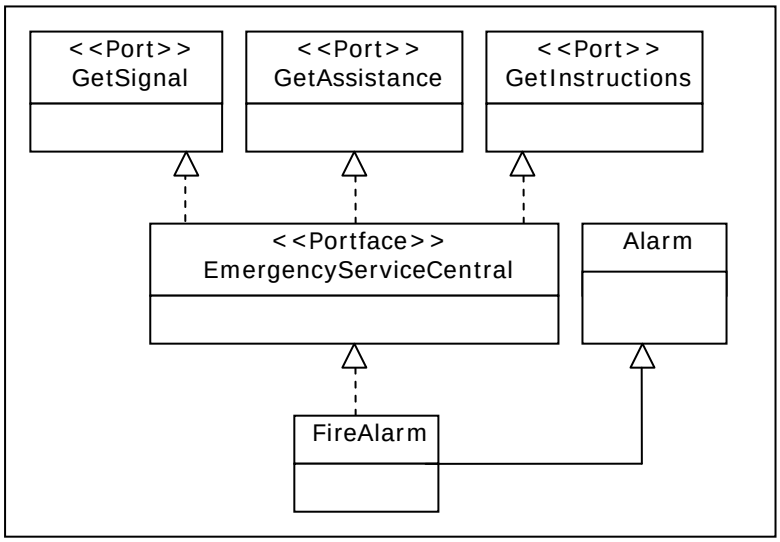
Figur 2.12 Skisse av et klassediagram

Figur 2.12 viser en skisse over tre porter og to klasser. Alarm er en generell beskrivelse av hvordan en alarm virker internt og er så generell at den ikke er tenkt til bruk i en composite-struktur. Klassen FireAlarm er en alarm som skal inn i en composite-struktur og definerer dermed de tre portene; GetInstruction, GetAssistance og GetSignal. Klassen FireAlarm spesialiserer klassen Alarm.



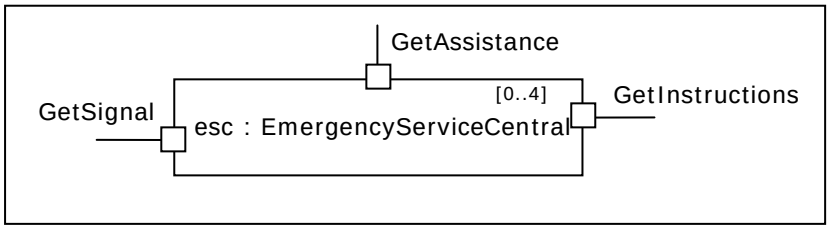
Figur 2.13 Utdrag av composite-struktur

Figur 2.13 viser et lite utdrag av en composite-struktur hvor vi ser at portene er brukt. Vi har ikke tegnet endene av connectorene, for det er ikke viktig i dette eksemplet. Vi ser at parten er typet med Alarm for å være sikker på at vi har Alarm objekter. Når vi nå får et FireAlarm objekt vet vi at det er av typen Alarm, men vi vet ikke at objektet vi får inn har portene den skal ha uten å sjekke det en for en. Portene kan brukes til andre formål også, så det er ikke nok å sjekke på noen, vi må sjekke alle. For å forbedre denne situasjonen kan vi gjøre slik:



Figur 2.14 Skisse av klassediagram med Portface

Figur 2.14 viser de samme portene og de samme klassene som Figur 2.12. Det som er forskjellen er at vi har definert en samling av portene som beskriver den rollen en alarmsentral har i en composite-struktur. Klassen FireAlarm realiserer nå portfacet EmergencyServiceCentral.



Figur 2.15 Utdrag av composite-struktur med bruk av portface som type

I Figur 2.15 ser vi at vi har typet parten med EmergencyServiceCentral som forsikrer oss om at alle objektene har akkurat de tre portene vi bruker i denne composite-strukturen. Som vi ser kan det være interessant å beskrive denne samlingen av porter som en type. Vi kan kalle dette konseptet for Portface.

I matematikk er mengder slik at et element e som eksisterer i en mengde A, kan også opptre i mengden B, og da blir B definert som enten å være en delmengde eller at den har et snitt felles med A.

Eksistensen til et objekt i informatikkens verden er begrenset av fysikk. Et objekts eksistens følger vanlig klassisk fysikk, for enhver informasjon i et objekt finnes det bare ett minneområde som er ansvarlig for å lagre denne informasjonen. Vi kan ha kopier, men de er dog kopier. Objektet finnes bare ett sted.

Når vi da snakker om mender av objekter i informatikk, har vi to måter å definere hva en mengde er. Den ene tolkningen er at mengden inneholder selve objektet og da vil vi kun ha disjunkte mengder. Den andre tolkningen er at vi har mengder med referanser til objektene. Vi kan si at selve objektene inngår i en grunnmengde hvor alle objektene ligger, nemlig selve minnet. Det er den siste tolkningen vi har valgt å bruke i denne oppgaven. Dette gir oss mulighet til å anse mengden av objekter av en subtype som en delmengde av objektene av supertypen, og det gir mulighet for å beskrive forhold mellom mengder av linker i et spesialiseringshierarki av associationer.

UML-diagrammer er av mange sett på som et vindu inn i modellen og viser derfor ikke hele sannheten. Derfor vil et kravet om disjunkte mengder, føre til mange potensielle problemer.

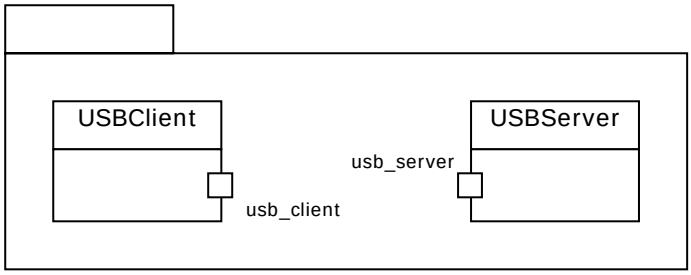
Et objekt kan ha en kompleks oppførsel og kan inngå i forskjellige roller sett fra forskjellige synsvinkler. Kravet om disjunkte mengder vil føre til at all modellering angående ett objekt må være ett sted eller i samme diagram. Dette bryter sterkt med UML's tankegang om at et diagram er modellen sett fra én side og trenger ikke å fortelle helheten. En person vil i ett tilfelle være med i kommunestyret samtidig som han er teknisk ansvarlig på et prosjekt på arbeidsplassen sin. Disse to rollene vil det ikke alltid være naturlig å beskrive i samme composite-struktur. Om alle mengdene definert av parter må være disjunkte vil det ikke la seg gjøre å beskrive både medlemmene i kommunestyret og prosjektet på to steder, uten å flytte personen rundt. I så fall kan ikke personen få SMS på

mobilen fra kommunestyret når han er på prosjektmøte eller utfører jobb som teknisk ansvarlig på prosjektet. I modellen kan han ikke fylle de to rollene samtidig, selv om han kan det i virkeligheten.

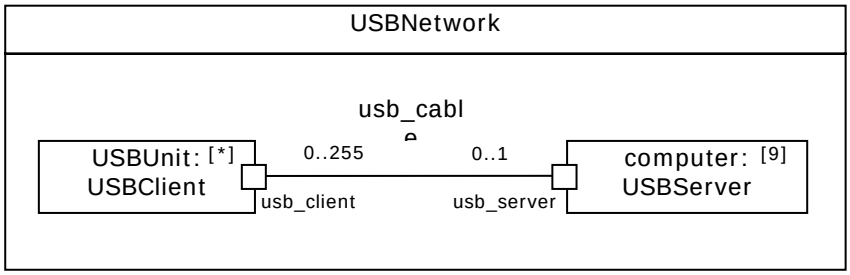
Dette er argumentene jeg har tatt hensyn til når jeg ikke anser partmengdene som disjunkte.

2.2.2 Connector

En klasse kan bestå av flere parter. Disse partene er typede og er en samling objekter av denne typen. To parter kan kobles sammen ved hjelp av en connector som gir objektene i de to partene mulighet til å kommunisere ved å sende meldinger via connectoren. En connector kan koble to parter direkte som vist i Figur 2.18, eller via to portApplicationer som vist i Figur 2.17. En portApplication er en bruk av en port. Portenes oppgave er å nærmere spesifisere hvilke meldinger som støttes og brukes av de forskjellige partene. På denne måten beskriver de hvilke meldinger som sendes og mottas over de forskjellige connectorene.



Figur 2.16 Klassediagram for USBClient og USBServer



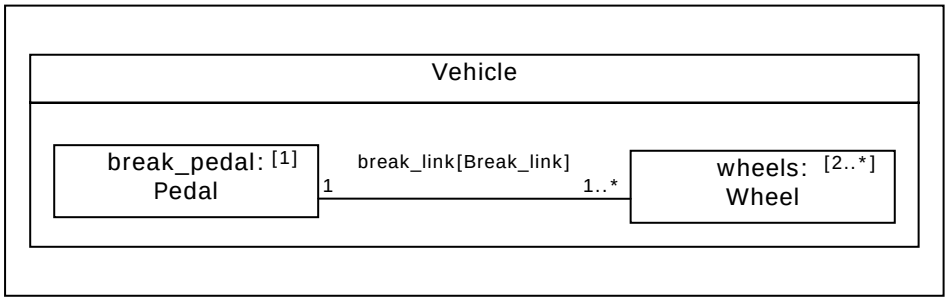
Figur 2.17 Connector mellom to parts

Eksemplet i Figur 2.17 viser hvordan vi kan modellere kommunikasjonen mellom en USB-klient og en USB-server. Denne modellen sier ikke noe om protokollen med hensyn på rekkefølge på meldingene, men den kan gi en detaljert beskrivelse av hvilke meldinger som støttes av partene i kommunikasjonen. Dette vil bli beskrevet i definisjonen av portene usb_client og usb_server. Vi kan tenke oss en printer med USB-port som skal koble seg til en usb_server. USB-kommunikasjonen vil da skje i form av meldinger over connectoren usb_cable. Siden USB-porten er navngitt kan printeren også modelleres med en parallellport uten at dette forstyrrer kommunikasjonen.

På samme måte som summen av alle linkene av en association holder rede på hvilke instanser som er koblet sammen, holder connectoren rede

på hvilke instanser som er koblet til hverandre over denne connectoren. En connector har også en kardinalitet som minner mye om den kardinaliteten en association har. Om vi kikker litt nøyere på Figur 2.17 ser vi at connectoren ”usb_cable” har to kardinaliteter. Den til venstre som er 0..255, som betyr at en gitt instans i parten computer kan være koblet til opp til 255 instanser i USBUnit. Til høyre ser vi at kardinaliteten er 0..1 og det betyr at en instans i parten USBUnit kan enten være tilkoblet til én instans i computer eller ikke være koblet til noen i denne parten. For de som kjenner USB-standarden er dette egenskaper ved USB-protokollen. Den bruker en byte [0..255] til å adressere de forskjellige enhetene som er koblet til USB-nettverket, og en enhet har kun én server av gangen.

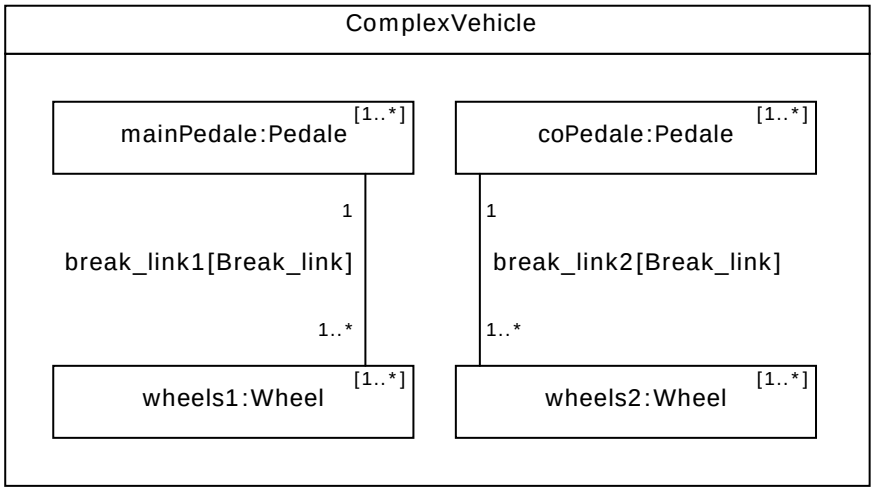
I Figur 2.17 ser vi at det ikke er en direkte kobling mellom kardinaliteten til connectoren og partene. Når det står at connectorens kardinalitet på venstre side er 0..255 gjelder dette for hver av de 9 USB-serverene i parten computer.



Figur 2.18 Connector mellom to parter

Eksemplet i Figur 2.18 viser hvordan en connector kan være basert på (”based on”) en association. Det vil si at denne connectoren er basert på associationen ”Break_link” som er beskrevet i Figur 2.6 side 9. Connectoren ”break_link” beskriver assosiationen ”Break_link” mer i detalj ved at denne connectoren har et objekt av Vehicle som kontekst. Dette fører til at et Pedal objekt må kobles med ett eller flere Wheel objekter som eies av samme Vehicle objekt som seg selv. Her ser vi at en bremsepedal (break_pedal) kan sende impulser til hjulene (wheels) om at de skal bremse. Med denne strukturen er vi sikre på at pedalen bremser den bilen som pedalen eies av, og vi unngår en situasjon som i Figur 2.7.

Når vi ser nøyere etter er kravene til kardinalitet for connectoren ganske gitte. Fordi connectoren baserer seg på en association må kardinalitetene til connectoren være innenfor kardinaliteten til associationen. Forøvrig sørger kardinaliteten på partene for at et Vehicle objekt minst har ett Pedal objekt og to Wheel objekter.



Figur 2.19 Et kjøretøy med to separate bremsesystemer

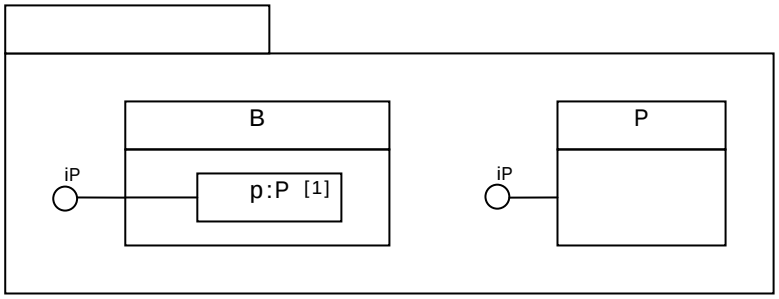
I Figur 2.19 ser vi oppbygningen av et annet kjøretøy som har to separate bremsesystemer. Vi har fortsatt beholdt kravet om at kjøretøyet må ha minst to hjul og at hver pedal må være koblet til minst ett hjul, men vi har nå fått mulighet til å bremse med minst to hjul i motsetning til ett. Siden både break_link1 og break_link2 bygger på Break_link må det være minst en kobling per connector, det er dette som forårsaker kravet om at minst to hjul må være tilkoblet brems. Om begge partene coPedale og wheels2 hadde en kardinalitet med nedre grense 0, ville vi brutt kravet om minst to hjul. Dette kunne vi fikset med å øke nedre grense på wheels1 til 2, men da ville vi mistet mulighet for å bare ha to hjul som er koblet til hver sin pedal, slik vi har på en motorsykkel. Da ville muligheten til å bruke den andre pedalen først kommet ved det tredje hjulet. Vi ser at med composite strukturer kan vi gi informasjon som klargjør detaljer, men er litt mer stivbeint enn det vi er vant med fra associationer.

En connector holder vanligvis rede på koblinger mellom instansene i partene den kobler sammen, men må den det i alle tilfeller? Finnes det ikke situasjoner hvor det er åpenlyst hvilke instanser som er mottakeren på den andre siden? Hvis alle på den andre siden skal gis beskjed må vi enten gi beskjed om at hver enkelt på den ene siden skal ha en kobling til hver enkelt på den andre siden og bruke disse koblingene, eller så kan vi ha en broadcast-connector som gir beskjed til alle på den andre siden uansett hvor mange det er.

Nå har vi omtalt en connector generelt, og eksemplene har gitt eksempler på det som UML kaller sammensetnings- (”assembly”) connector, fordi den kobler sammen parts til en composite-struktur. En annen type connector vi har er en ”delegate”-connector. Denne delegerer ansvar fra et nivå i strukturen til et indre nivå.

En delegate connector kan ses på som en forlengelse av en annen connector. Den blir brukt der en ytre klasse skal tilby noe den selv ikke implementerer, men som en part tilbyr. Så delegate connectoren delegerer ansvaret for interfacet eller tjenesten til en annen som tilbyr

det. Den kan på mange måter sammenlignes med å viderekoble en telefonlinje til en annen telefon.



Figur 2.20 Eksempel på delegator-connector

Vi ser at klassen P implementerer interfacet iP, og at klassen B skal tilby dette interfacet. Her brukes en delegate connector for å delegere interfacet iP implementert av klassen P slik at klassen B også kan tilby dette interfacet uten å implementere det selv. Vil multiplisitet på denne connectoren være fornufting, eller vil multiplisiteten være gitt av connectorens natur?

2.2.3 Forskjellen på association og connector

Historisk sett har association og connector opphav i to forskjellige modelleringstradisjoner. Etter vår kunnskap er de ikke beskrevet i samme modelleringsspråk før i UML 2. Samtidig som de begge beskriver et forhold mellom objekter, er omgivelsene for selve beskrivelsen forskjellig, og følgelig er de ikke lette å sammenligne. Connectoren er relativt ny i UML og vi ønsker å studere den nærmere for å få en mer presis forståelse og en entydig tolkning av den i modeller. Samtidig er det ønskelig å forene association og connector slik at de kan utfylle hverandre.

Oppgavens formål er å finne en entydig semantisk tolkning av både association og connector slik at de kan opptre sammen i et modelleringspråk på en helhetlig og utfyllende måte.

En association beskriver hvordan hvert objekt av en klasse forholder seg til andre objekter av andre klasser. En connector beskriver hvordan mengder av objekter forholder seg til andre mengder av objekter, og hvor mengdene beskriver objekter som er i en bestemt situasjon og med felles egenskaper. En association sier dermed noe om alle objekter av en klasse, mens connectoren beskriver noe om objekter i en bestemt situasjon.

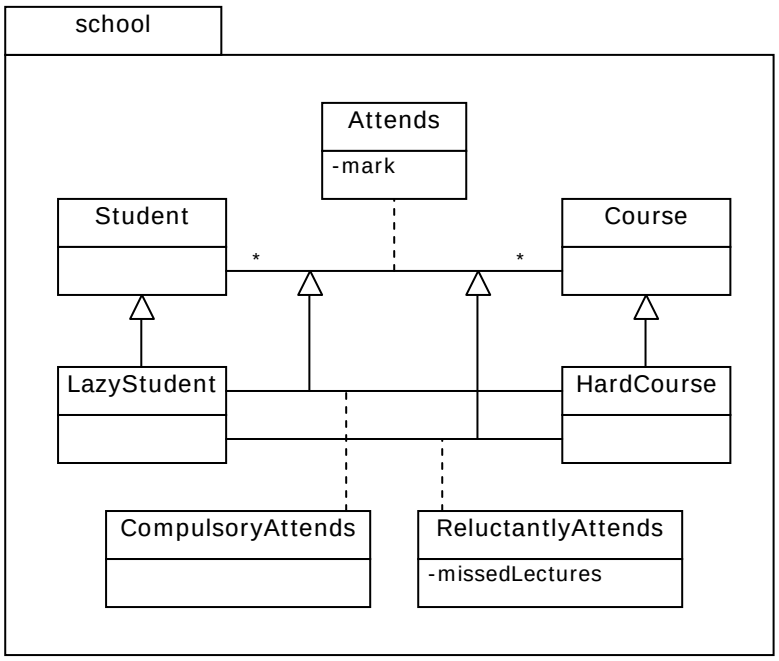
En association er naturlig å omtale som en type, mens connector er mer en beskrivelse av hvordan ting skal fungere.

2.3 Spesialisering av association

Spesialisering av associationer er et omdiskutert emne. UML har spesialisering av associationer, men definerer ikke selve tolkningen av hva dette betyr. Bierman og Wren[3] tar opp temaet og identifiserer problemene de mener vi vil få om denne spesialiseringen fungerer på samme måte som for klasser. Klasser spesialiseres slik at en subklasse får

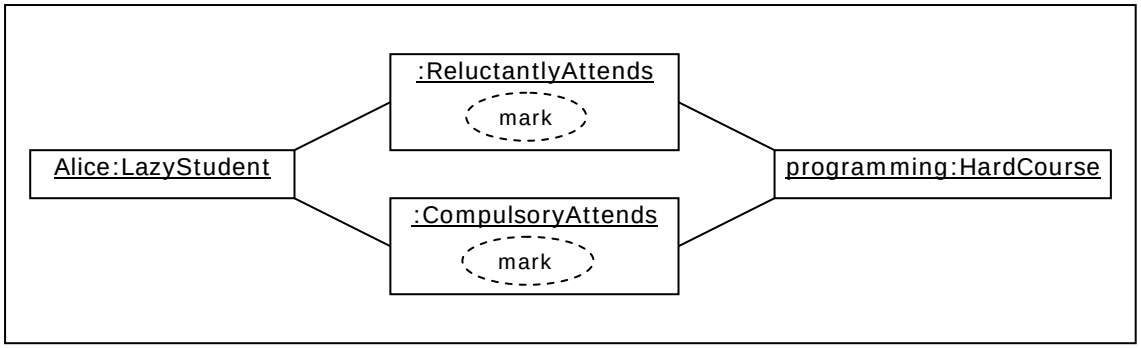
en kopi av alle attributtene superklassen har. Problemet de omtaler blir synlig i eksemplet vist i Figur 2.21 og Figur 2.22. Figur 2.21 beskriver klassediagrammet og Figur 2.22 viser hvordan attributten mark blir duplisert om associationen spesialiseres på samme måte som for klasser.

I klassediagrammet i Figur 2.21 har vi to spesialiseringer av en association Attends; CompulsaryAttends og ReluctantlyAttends. Samtidig inneholder Attends attributten mark.



Figur 2.21 Illustrasjon av modellen 'school'

I dette eksemplet har vi to forskjellige måter å delta i et kurs (Course) på, og de er ikke gjensidig utelukkende. Det vil si at vi kan få en instans av både CompulsaryAttends og ReluctantlyAttends mellom studenten Alice og kurset programmering som vist i Figur 2.22.

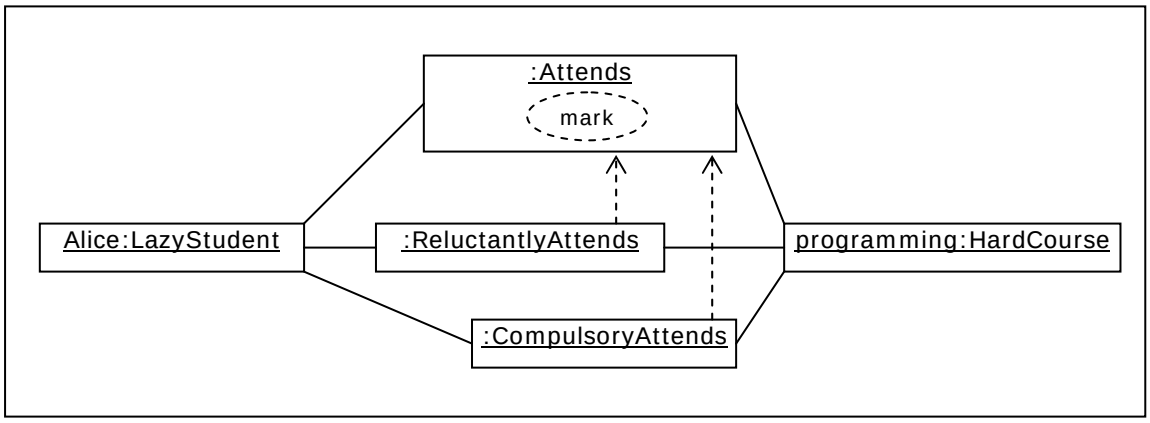


Figur 2.22 Objektdiagram med duplisering av attributten mark

Vi ser at instansene av CompulsaryAttends og ReluctantlyAttends har en mark attributt hver. På denne måten blir attributten mark duplisert og

Alice får to karakterer i programmering, mens vi ønsker at Alice bare skal ha én karakter i dette faget.

Bierman og Wren konkluderer derfor med at spesialisering av associationer ikke følger samme regler som for spesialisering av klasser. De ser for seg en løsning hvor hvert spesialiseringsnivå blir representert med en egen instans som har ansvaret for attributtene som er definert på det respektive nivået, og at objektet delegerer ansvaret for eventuelle andre attributter til ”super”.



Figur 2.23 Delegering av ansvar for attributten ’mark’

Figur 2.23 viser hvordan instansene av klassene og associationene forholder seg til hverandre om Bierman og Wren sin tolkning brukes. Denne løsningen vil forhindre at vi får multiple kopier av attributter.

Eksemplet som Bierman og Wren presenterer i sin artikkel er overbevisende i seg selv, men er det representativt for spesialisering av associationer generelt? Og er det vanlig at vi får instanser av flere av spesialiseringene samtidig, mellom de to samme objektene som vist i Figur 2.23? Om vi analyserer dette eksemplet og sammenligner det med andre eksempler vil vi kanskje komme frem til et annet resultat.

Spesialisering av association aktualiserer problemstillingen om vi skal tillate flere linker av samme type mellom to gitte instanser. Bierman og Wren mener det er klart at det kun kan finnes én link av samme type mellom to gitte instanser, men er dette fornuftig med hensyn på spesialisering og subtyping? I så fall må vi ha et bevisst forhold til hva det innebærer at en association er unik.

2.4 Metode

En passende metode for dette prosjektet vil være først å lage et lite objektorientert modelleringsspråk med vekt på associationer, connectorer og parter. Språket kan lages ved å definere den abstrakte syntaksen i en metamodell og implementere elementenes tolkning i et runtime-system som er i stand til å kjøre modellen.

Det neste steget i metoden vil være å teste modelleringsspråket med eksempler. Da vil vi finne ut om den abstrakte syntaksen er rik nok til å favne de viktige egenskapene ved association og connector, og om

modellene blir tolket riktig og dermed gir programmene den riktige funksjonaliteten.

Eclipse-plattformen med Eclipse Modelling Framework (EMF) er godt utviklet for metamodellering og egner seg derfor godt som utviklingsverktøy for metamodellen i dette prosjektet.

Vi ønsker å lage et runtimesystem i Java siden Java er en åpen plattform, og siden det er et objektorientert språk som har mange av de fasilitetene vi skal modellere med. Vi vet at vi skal modellere med elementer som Java ikke har. Dette gjelder portface, port, part, portApplication, association og connector, og vi må derfor bygge vår egen runtime-plattform som støtter alle disse elementene.

De egenskapene som modelleringsspråket har som allerede eksisterer i Java vil vi bruke slik Java har dem. Det vil si at vi ikke vil ha en annen tolkning av spesialisering av klasser enn det Java har. Dette gjelder også synlighet av medlemmer og implementasjon av indre klasser. Dette gjør vi for å ha mer fokus på de elementene Java ikke har. Det eneste vi vil definere på vår egen måte er hvilke egenskaper en generell klasse har. Vi må regne med at Java-klasser, slik de er i dag, ikke er tilstrekkelige til å favne de egenskapene vi er ute etter, og at vi derfor må regne med å lage en utvidelse av Java-klassen.

Overgangen fra modellen, som er en instans av metamodellen i XML/XMI, til Java-kode, velger vi å gjøre med et MOFScript som er en plugin til Eclipse-plattformen. Dette scriptet vil transformere modellen til Java-kode som kjører på runtime-plattformen.

Modelleringsspråket skal være et generelt modelleringsspråk som har mange av de samme egenskapene som UML, men det vil være rettet mot en implementasjon på en Java-plattform. Språket vil ikke støtte modellering av oppførsel i noen annen form enn direkte i Java-kode. Denne koden skal likefullt være en del av modellen.

3.0 Eksempler

Det er en fordel å ha gode eksempler i tillegg til de som ble presentert i problemstillingen. Eksempler er viktige for å prøve ut forskjellige bruksområder for connector og association samtidig som vi vil kunne se resultatet av tolkningene i praksis. Som omtalt tidligere kan en connector brukes i flere situasjoner. I en situasjon kan vi bruke connector til å delegere, et annet sted kan en connector være basert på en association for å beskrive hvordan denne associationen arter seg. Eksempler kan brukes for å forsikre oss om at vi kommer frem til en entydig tolkning av connectorer, associationer og deres omgivelser. Ved å ha eksempler som bruker disse begrepene på flest mulig måter, vil de kunne avdekke svakheter og tvetydighet i tolkningen. Disse eksemplene vil også kunne være forsøksmodeller for å sjekke om en MDA-teknologi klarer å tolke modellene entydig og om det resulterer i et kjørbart program som utfører det som er forventet. Her følger noen eksempler som vi vil bruke for å teste tolkningene av connector og association.

3.1 Forskning initiert av Utdanningsdepartementet

Formålet med dette eksemplet er å lage en modell som inneholder parter typet med portface og at disse partene samarbeider via connectorer og porter. Eksemplet skal omfatte bruk av både en vanlig connector og en broadcast connector.

Vi ser for oss at vi skal modellere forskning på undervisning i Norge. Denne forskningen baserer seg på empiriske undersøkelser på undervisningsstedene. Denne modellen er veldig forenklet. Forskningsoppdraget kommer fra Utdanningsdepartementet og blir gitt til en forskningsorganisasjon som gjennomfører en undersøkelse av undervisningen på ett eller flere undervisningssteder. Etter undersøkelsene gjør forskningsorganisasjonen en evaluering av resultatene og leverer resultatet til Utdanningsdepartementet. Resultatet inneholder forslag til endringer for å kunne forbedre undervisningen ytterligere. Utdanningsdepartementet følger opp rådene ved å gi instrukser til alle undervisningsstedene om hvordan de skal endre undervisningspraksis, og undervisningsstedene gir til slutt tilbakemelding til Utdanningsdepartementet om hvordan endringene i undervisningspraksisen gikk.

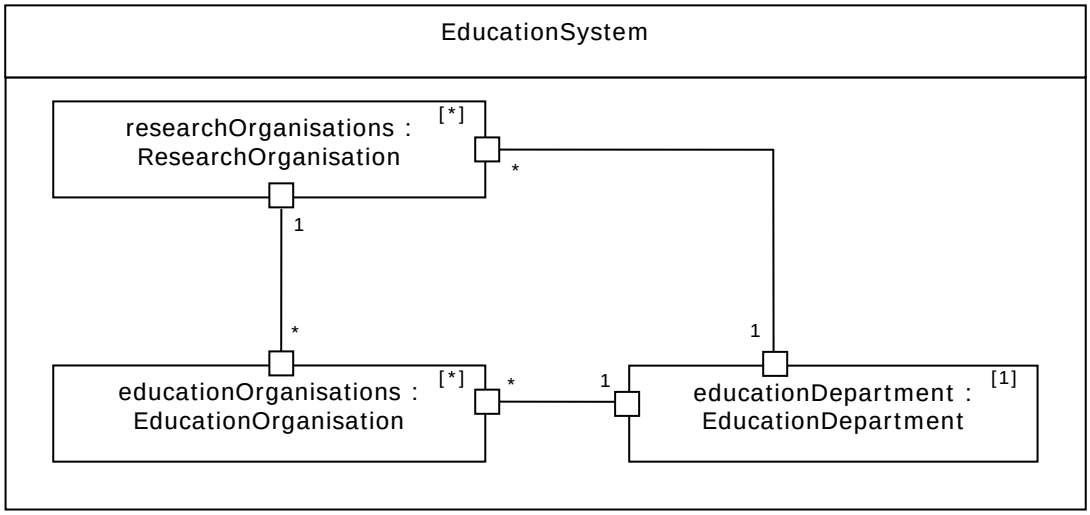
Når vi skal modellere dette må vi identifisere hva slags enheter vi har. Vi har et generelt undervisningssted som får instrukser om hvordan undervisningen skal gjennomføres, og som deltar i undervisningsundersøkelser som sjekker hvordan undervisningen fungerer. I tillegg gir de tilbakemelding til Utdanningsdepartementet om hvordan gjennomføringen av endringene gikk.

Vi har en generell forskningsorganisasjon som får forskningsoppdrag, utformer undersøkelse av undervisningen, og som evaluerer resultatene av disse undersøkelsene, for så å returnere forskningsresultatene til oppdragsgiver.

Vi har et undervisningsdepartement som gir forskningsoppdrag og som instruerer om hva slags endringer undervisningsstedene skal gjennomføre i undervisningspraksisen sin basert på resultater av forskningen. De tar også imot informasjon om hvordan gjennomføringen av endringen i undervisningspraksisen gikk.

Om vi nå ser på reelle objekter, finner vi at vi kan ha vanlige skoler som Sogn VGS, og om vi antar at Høgskolen i Oslo ikke bedriver forskning på undervisning så vil de også være en vanlig skole i denne modellen. Derimot ser vi at et universitet, som UIO og NTNU, vil kunne være både et undervisningssted og en forskningsorganisasjon. Vi kan også ha egne forskningsorganisasjoner som bare gjennomfører forskningsoppdrag, uten å bedrive undervisning selv.

Dersom UIO fikk forskningsoppdraget, ville det ikke være unaturlig å gjennomføre undersøkelse av undervisningen på UIO. Dette vil føre til at UIO vil inngå i to forskjellige roller i denne modellen. UIO vil både være en forskningsorganisasjon og et undervisningssted.



Figur 3.1 Skisse av EducationSystem

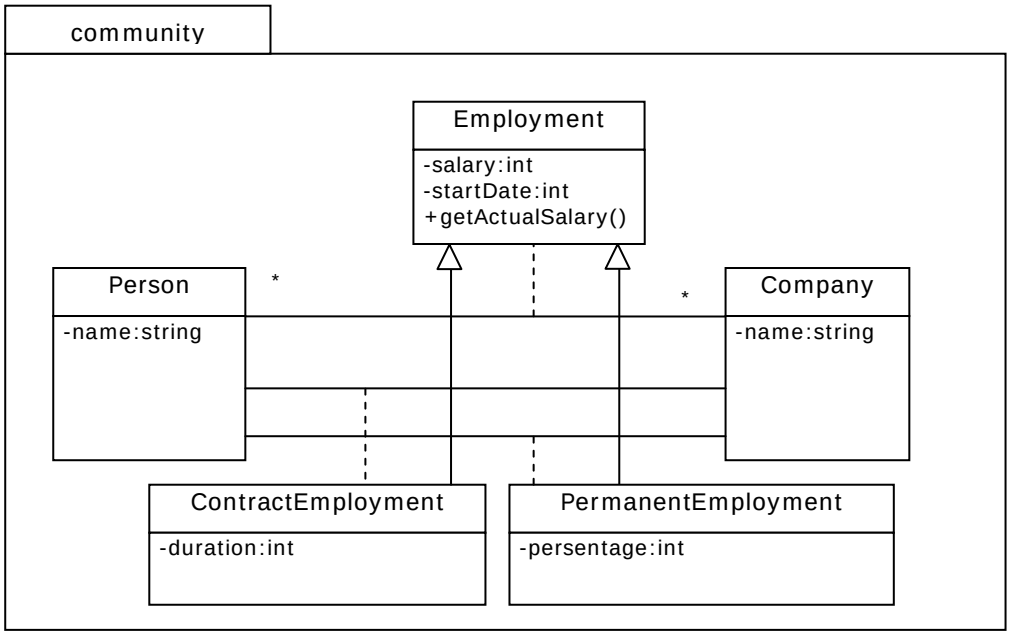
I Figur 3.1 kan vi se en skisse over løsningen. Vi har valgt å beskrive forskningsorganisasjonenes rolle i parten ”researchOrganisations” og rollen som undervisningssted i parten ”educationOrganisations”. Parten educationDepartment består av bare ett objekt og det er en instans av EducationDepartment. Det er dette objektet som tildeler forskningsoppdrag til en forskningsorganisasjon i parten researchOrganisations. Parten educationOrganisations er samlingen av alle undervisningsstedene som er tilstede i modellen. Det er disse som skal få instrukser av departementet om hvordan de skal undervise. Det er et utvalg av disse som skal være med på en undervisningsundersøkelse arrangert av en forskningsorganisasjon.

3.2 Community

I dette eksemplet skal vi prøve ut spesialisering av association, og en connector som er basert på en association. Denne connectoren er basert

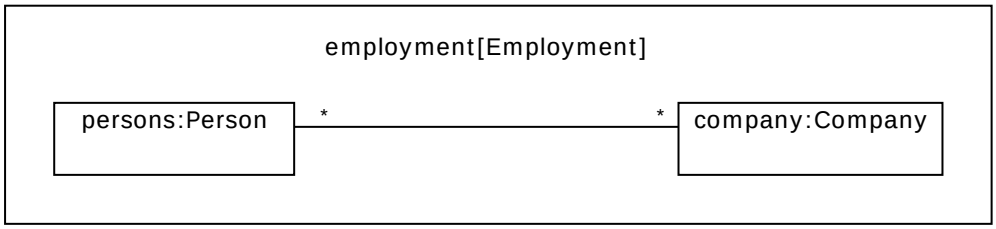
på en association samtidig som at associationen er spesialisert. Når en connector er basert på en association kan connectoren nyttegjøre seg av implementasjon og informasjon i linkene av denne associationen til å utføre oppgaver.

Vi kan tenke oss en modell av et lokalsamfunn med personer (”Person”) og bedrifter (”Company”), hvor personene er ansatte i bedriftene. En ansettelse (”Employment”) modellerer vi som en association mellom personklassen og bedriftsklassen. Vi har to typer ansettelser, fastansettelse (”PermanentEmployment”) og kontraktansettelse (”ContractEmployment”). Når en bedrift ønsker å betale lønnen (”salary”) til de ansatte, vil bedriften kunne bruke ansettelsen direkte for å betale riktig lønn. Meldingen for å betale lønn kan sendes over en connector som er basert på associationen ”Emloyment”.



Figur 3.2 Skisse over community pakken

Vi ser av Figur 3.2 at vi har to spesialiseringer av associationen ”Employment” og at hver av de har egne attributter som gir informasjon om denne ansettelsestypen. Metoden ”getActualSalary” kan implementeres av ”ContractEmployment” og ”PermanentEmployment” slik at vi alltid kan få informasjon om riktig lønn når vi kaller denne metoden.



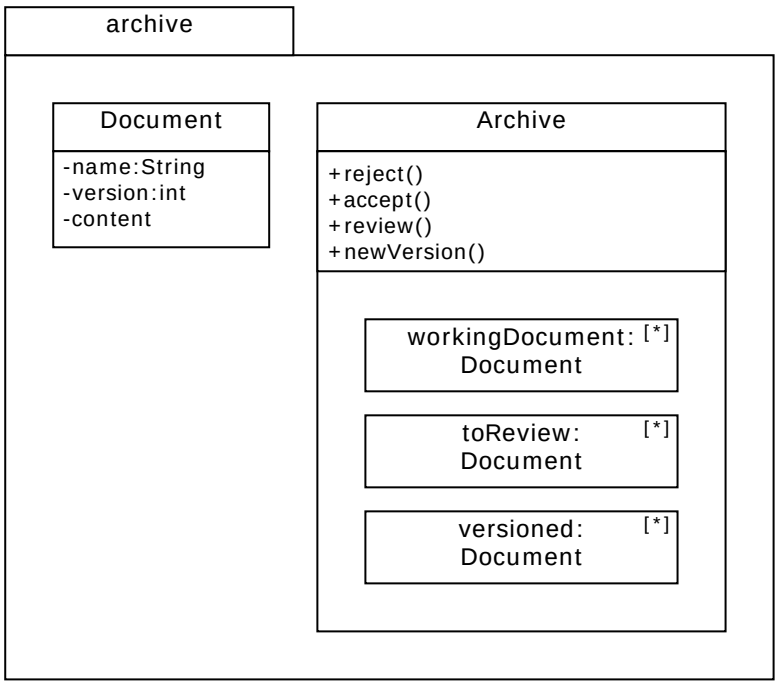
Figur 3.3 Skisse over connector basert på associationen Employment

I Figur 3.3 ser vi en skisse over en connector som er basert på associationen ”Employment” fra Figur 3.2. Denne connectoren kobler sammen en part med personer (”Person”) og en part med selskaper (”Company”). Connectoren er basert på associationen Employment og kan dermed bruke informasjon om personenes lønn (”salary”) fra linkene.

3.3 Archive

Formålet med dette eksemplet er å la parten beskrive romlig plassering av objekter og prøve ut flytting av objekter fra en part til en annen.

Dette er en modell av et dokumentarkiv med dokumenter i forskjellige stadier. Et dokument er enten klassifisert som arbeidsdokument, til gjennomsyn og godkjenning, eller som godkjent dokument. Et dokument blir opprettet som et arbeidsdokument, og deretter sendt til gjennomsyn og godkjenning. Dokumenter til gjennomsyn og godkjenning kan enten bli godkjent eller avvist. Avviste dokumenter sendes tilbake til utarbeidingsfasen som arbeidsdokument, og godkjente dokumenter ender som et godkjent dokument. Når et godkjent dokument skal endres må det opprettes en ny versjon av dokumentet som blir klassifisert som et arbeidsdokument. En skisse over løsningen er gitt i Figur 3.4.



Figur 3.4 Skisse av archive eksemplet

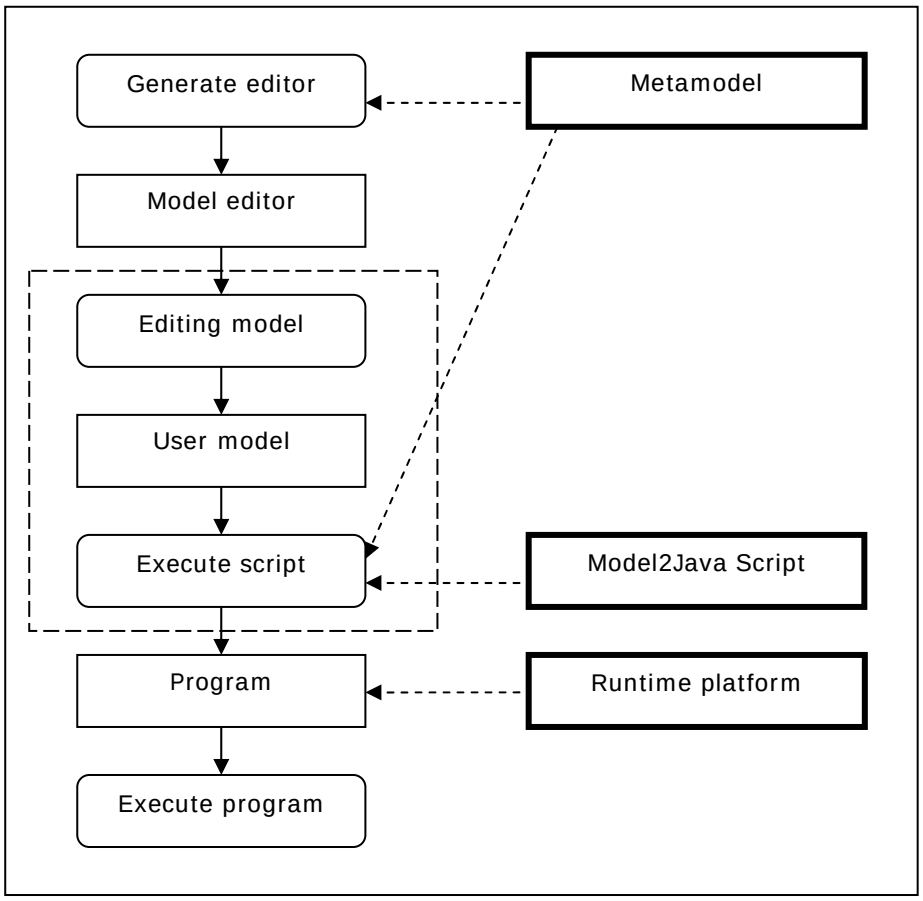
Dette blir da en modell på versjonshåndtering av dokumenter, og vil føre til at objekter flyttes fra en part til en annen.

4.0 Modelleringsspråk med omgivelser

Arbeidet begynte med utvikling av en metamodell. Utfordringen var å få modelleringsspråkets egenskaper så likt UML sine som mulig, samtidig som vi klarte å få frem de egenskapene vi ønsket. Metamodellen for UML er en flittig bruker av multippel arv og vi ønsket å bruke enkel arv for å lette implementeringen i Java. Da metamodellen begynte å ta form var det klart for å bygge en runtime-plattform som implementerte runtime-elementer som svarte til de samme byggesteinene som i metamodellen. Vi bygde runtime-plattformen gradvis og testet den ut med håndkodede eksempler som testet funksjonaliteten.

Da metamodellen hadde de fleste av de egenskapene som vi ønsket og runtime-plattformen omfattet de fleste elementene, var tiden inne for å lage et script for å automatisere overgangen fra eksempelmodellene i XML/XMI format til Java-kode. Denne Java-koden er referert til som ”Program” i både Figur 4.1 og Figur 4.2.

Nå skal vi vise hvordan artefaktene i denne oppgaven brukes i modellbasert systemutvikling.



Figur 4.1 Oversikt over hvordan artefaktene i oppgaven henger sammen

Figur 4.1 viser hvordan artefaktene i oppgaven blir brukt i utviklingsprosessen av et programvaresystem. Firkanter er artefakter og de avrunda figurene er aktiviteter. Oppgavens viktigste artefakter er

merket med fet ramme. Disse er ”Metamodel”, ”Model2Java script” og ”Runtime platform”. Pilene viser flyten i arbeidet og de stiplede pilene indikerer i hvilke stadier i utviklingen oppgavens artefakter blir brukt. Den stiplede firkanten indikerer de aktivitetene som en utvikler er nødt til å utføre for hver iterasjon for å teste programmet.

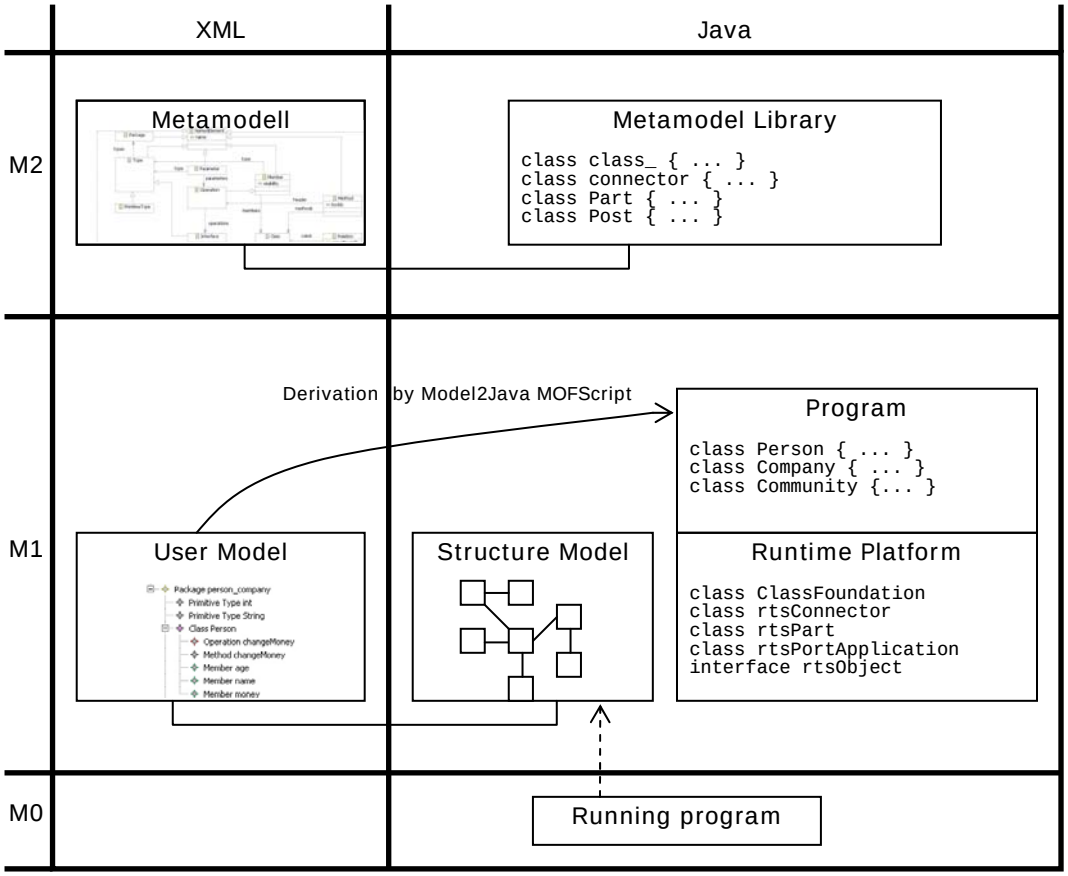
Artefaktene i Figur 4.1 er som følger:

- ”Metamodel” er metamodellen som beskriver den abstrakte syntaksen til modelleringsspråket.
- ”Model editor” er en tre-editor (tree view editor) som er i henhold til metamodellen.
- ”User model” er brukermodellen som inneholder all den informasjonen som trengs for å lage det kjørbare programmet i Java.
- ”Program” er selve programkoden som importerer og bruker runtime-plattformen. All denne koden er i Java.
- ”Modell2Java Script” er et modell-til-kode script som oversetter en brukermodell til Java-kode som bruker runtime-plattformen.
- ”Runtime platform” er en runtime-plattform i Java for å implementere egenskaper som Java ikke har, som f.eks. parter, connectorer og associationer.

Aktivitetene i Figur 4.1 er som følger:

- ”Generate editor” er en automatisk transformasjon i et EMF prosjekt, i Eclipse, hvor metamodellen først er lastet inn i et EMF prosjekt. Når denne transformasjonen blir kjørt blir det laget et Java-program. Dette Java-programmet er en tre-editor for å lage brukermodeller som er i henhold til metamodellen.
- ”Generate editor” er en en-gangs-aktivitet. Vi trenger ikke generere editoren mer enn en gang.
- ”Editing model” er aktiviteten hvor modelleringen skjer. Her blir brukermodellen utformet og gitt all den logikk som programmet skal ha.
- ”Execute script” er aktiviteten hvor ”Modell2Java Script” kjøres i MOFScript som er en Eclipse-plugin. MOFScript kjører dette scriptet med brukermodellen som input og metamodellen som omgivelse, slik at brukermodellen blir tolket som en instans av metamodellen. Scriptet genererer programkoden som tar i bruk runtime-plattformen. Når så programmet kjøres bruker programkoden runtime-plattformen. Denne plattformen oppretter objekter som har implementert egenskapene til associationer og connectorer.

Nå har vi forklart hvordan artefaktene i denne oppgaven støtter utvikling av en modell, og hvordan de skal brukes. Nå er det på tide å se hvordan denne teknologien som helhet fungerer i et metamodelleringsperspektiv.



Figur 4.2 Oversikt over rammeverket mhp. metamodelleringsnivåer

Figur 4.2 viser hvordan rammeverket forholder seg til andre artefakter i oppgaven. Figuren er delt vertikalt mellom XML og Java. Til venstre har vi ”Metamodel” og ”User Model” som er i XML format, og til høyre har vi de tingene som er i Java-verdenen, i en eller annen form. Horisontalt er figuren delt inn etter OMGs standard for nivåer innen metamodellering.

M0 er metanivå 0 og det er der objekter eksisterer. Her har vi Java-objekter som ”lever” og kommuniserer. Sagt på en annen måte så er dette det kjørende programmet. M1 er metanivå 1 og er en beskrivelse av hvordan programmet skal være med hensyn på logikk, struktur og oppførsel. Typisk inneholder dette nivået instanser av metamodellelementer som Class, Member, Operation og Association. M2 er metanivå 2 og beskriver konseptene som Java og modelleringsspråket består av. Her blir konseptene class, expression, method og interface definert.

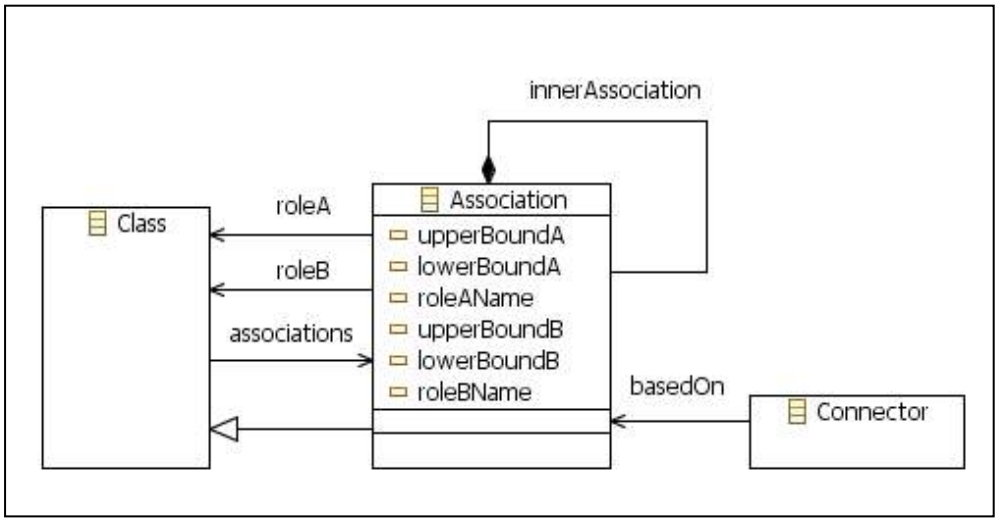
Vi begynner å forklare Figur 4.2 med ”User Model” som er i XML på nivå M1. ”User Model” blir tolket av MOFScriptet som genererer et Java-program som er gitt navnet ”Program”. Dette Java-programmet består av to deler, hvor den ene delen er selve programmet og den andre er kode for å opprette ”Structure Model”. Denne modellen er en objektstruktur i

Java over modellens struktur og tilhører M1. Det vil med andre ord si at denne modellen er en kopi av strukturen i ”User Model”. Dette blir beskrevet mer detaljert i kapittel 4.5 Runtime-plattform side 39. Øverst i høyre hjørne har vi ”Metamodel library”, som har en Java-klasse for hver metaklasse definert i metamodellen. Klassene i ”Metamodel Library” blir instansiert for å lage ”Structure Model”. ”Program” importerer ”Runtime Platform”, og når programmet kjører så fungerer ”Runtime Platform” som en plattform som utvider Java til et språk som inneholder associationer, connectorer og parts.

4.1 Association

Association i dette modelleringsspråket skiller seg lite fra Association i UML 2. Som i UML beskriver en Association med to ender et forhold mellom to klasser, og de to klassene har dermed en rolle overfor hverandre.

Selv om betydningen av begrepet ikke skiller seg så mye fra UML, så er metamodellen ganske annerledes. Et utdrag av metamodellen med fokus på Association er vist i Figur 4.3.

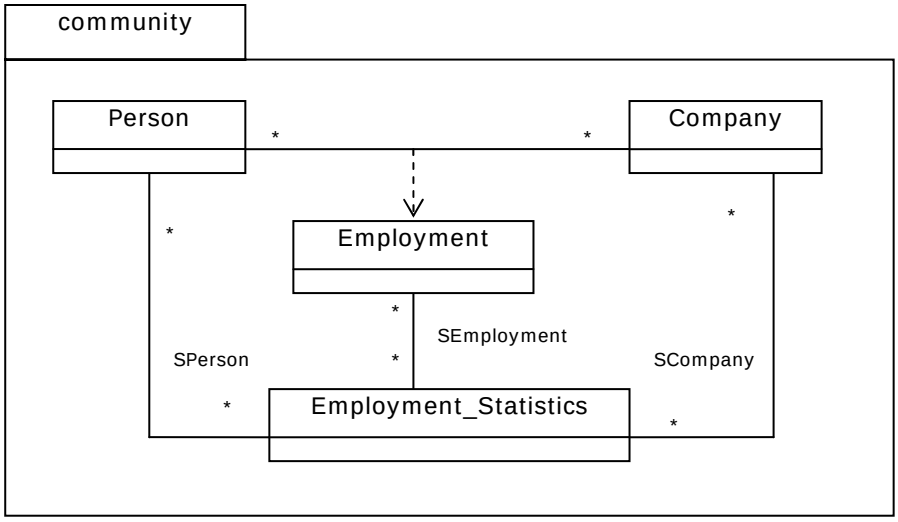


Figur 4.3 Utdrag av metamodell med omgivelsene til Association

Associationen har to roller, en for hver av endene. Hver rolle består av tre egenskaper som illustrert i Figur 2.3, side 7. Disse tre er kardinalitet, rollenavn og rolleklasse. De tre egenskapene er fordelt på fire egenskaper i metamodellen. Alle fire egenskapene er navngitt slik at vi vet hvilken side de tilhører. De tilhører enten side A eller side B. Kardinaliteten er delt opp i de to attributtene lowerBound og upperBound. Disse definerer øvre og nedre grense for kardinaliteten. Vi har en kardinalitet for hver av sidene A og B, hvor kardinalitet for side A bestemmes av ”lowerBoundA” og ”upperBoundA”, og hvor ”lowerBoundB” og ”upperBoundB” bestemmer kardinaliteten for side B. Rolleklassene heter ”roleA” og ”roleB” og er referanser til klassene som innehar de navngitte rollene i ”roleAName” og ”roleBName”.

Associationer kan også ha nestede associationer som eies via innerAssociation. Dette er nærmere beskrevet i kapittel 7.3.

Selv om Association er en spesialisering av Class, betyr det ikke at en association har alle egenskapene til Class. En link er noe annet enn et objekt. En link er passiv og eksisterer i utgangspunktet for å beskrive et bestemt forhold mellom to objekter. På mange måter kan en link sammenlignes med et dataobjekt som holder informasjon. En link kan inneholde mye logikk for å ivareta den riktige informasjonen, men den er ikke aktiv. Disse forskjellene medfører at en association ikke har parter eller connectorer, og dermed ikke bruker ”connectors” og ”parts” som Class. Vi beskriver med andre ord ikke en association med en composite-struktur.



Figur 4.4 En tenkt utvidelse av community klassen

Figur 4.4 viser at linker også kan bli knyttet sammen av andre linker. Vi tenker oss en utvidelse av Community eksemplet som har personer og bedrifter som er assosiert med ansettelser. Disse Employment linkene inneholder informasjon som lønn. Objekter av klassen Employment_Statistics regner ut lønningsstatistikker. Informasjonen til disse statistikkene er hentet fra Employment linkene og er selv linket til de ansettelsene som statistikken er basert på. Dette resulterer i at linkene av associationen SEmployment linker et Employment_Statistics objekt og en Employment link.

Spesialisering av association fungerer på samme vis som for klasser, så en subassociation har en kopi av alle attributtene som superassociationen har.

En association kan ha flere linker mellom to gitte instanser. Dette gjør det mulig å ha linker av flere associationer i ett hierarki av spesialiserte associationer mellom to gitte instanser.

På mange måter har vi analysert den kompliserte bruken av Association med associationklasser og navigerbarhet begge veier. Vi har lagt lite vekt på associationer som er en til mange og navigerbar en vei. Denne typen associationer blir ofte implementert som en liste med referanser. Denne

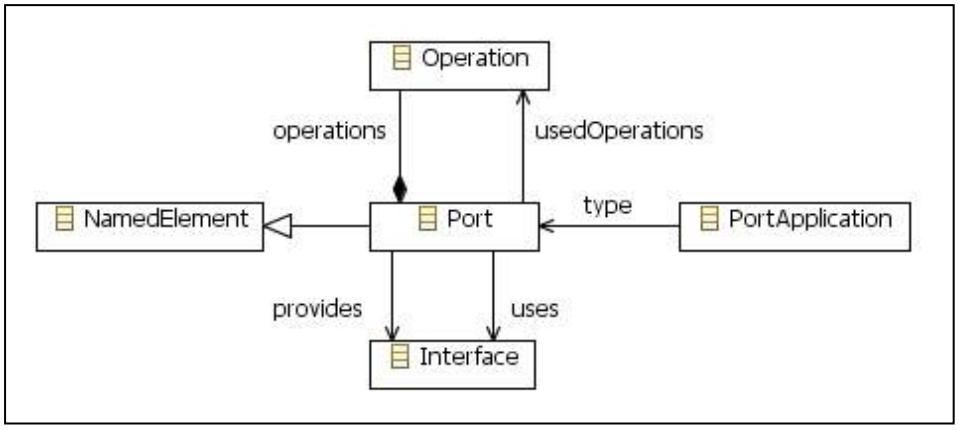
funksjonaliteten støttes fullt ut i runtime-systemet og er i virkeligheten spesialtilfeller av den generelle teorien, så lenge det ikke finnes krav om at navigerbarhet ikke skal være mulig. Navigerbarhet er implementert i runtime-systemet ved at det er mulig å spørre associationene om oppslag for å finne linker og partnere. To instanser er partnere om de inngår i samme link. På denne måten kan hvem som helst få tak i associationen og finne informasjon om linkene. Det kan virke tungvint og lite effektivt med oppslag, men resultatet er at vi får samlet implementasjonen av associationen på ett sted. Implementasjonen blir ellers distribuert uten at associationen er en grunnleggende byggestein.

Når en instans dør vil alle linker som har denne instansen referert i en av rollene bli meningsløse og bør slettes.

Eksempelet Community i kapittel 3.2 og bruk av associationer blir gjennomgått i kapittel 6.0 og i kapittel 7.0 gjennomgår vi bruk av spesialisering av associationer.

4.2 Port, Portface og Part

En Port er et begrep som ble innført i UML 2.0 for å spesifisere kommunikasjon over connectorer. Port er i denne oppgavens modelleringsspråk litt annerledes enn i UML. Den er utvidet med operasjoner, så den kan støtte og bruke operasjoner som ikke er en del av et interface.



Figur 4.5 Utdrag av metamodell med omgivelsene til Port

Figur 4.5 viser at Port er en spesialisering av NamedElement, samtidig som den via ”type” er en spesifikasjon på en PortApplication. Forklaringen på dette kommer i Figur 4.6, der vi ser at porter til slutt ender som en del av et Portface, og Portface er en type. Vi kan si at selv om Port ikke er en type i vanlig forstand så beskriver den egenskaper som objekter og porter har. ”provides” og ”uses” er som i UML en beskrivelse på hvilke interfacer som denne porten tilbyr og bruker. Operations beskriver alle operasjonene som direkte støttes av denne porten, mens usedOperations er en liste over operasjoner som denne porten krever tilgang til.

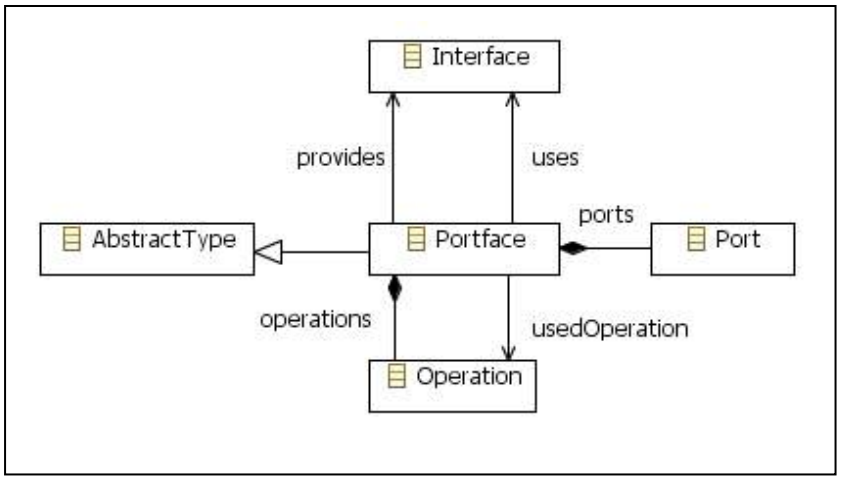
Portface finnes ikke i UML og den er et resultat av å forene modellelementer som finnes i klassediagrammer og composite-

diagrammer. Disse to diagramtypene har opphav i to forskjellige modelleringstradisjoner. Disse to tradisjonene er også opphav til association og connector, hvor association hører hjemme i klassediagrammet, mens connectorer hører hjemme i composite-diagrammet.

UML gir mulighet for å beskrive en samling av porter på en part, via PortApplication i en composite-struktur. En slik beskrivelse gjelder dog bare for denne bestemte composite-strukturen. Det er denne beskrivelsen vi ønsker å generalisere for gjenbruk ved å gi den et navn og definere den som en type. I tillegg til dette er Portface en utvidelse av Interface ved at Portface også beskriver hvilke operasjoner den må ha tilgang til. I likhet med Interface er Portface en navngitt type som ikke har implementasjon knyttet til seg.

Flere egenskaper er forent med Portface slik at disse elementene blir en del av en helhetlig tankegang med flere fasetter, i stedet for to atskilte verdener som har hver sine unike egenskaper.

Hvordan et Portface forholder seg til resten av metamodellen er vist i Figur 4.6.



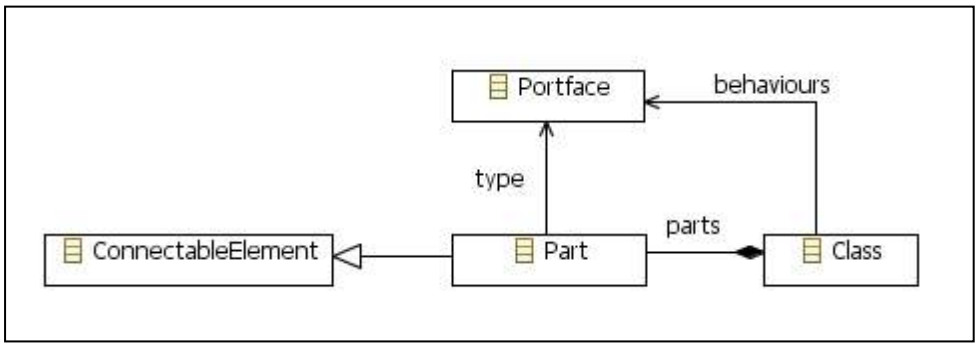
Figur 4.6 Utdrag av metamodell med omgivelsene til Portface

Portface spesialiserer AbstractType og er dermed en navngitt type uten implementasjon. ”provides” er en liste over interfacer som støttes eller realiseres av den definerte typen. En subtype kan implementere interfacet, men det kan også delegeres fra en klasse som implementerer det. ”uses” er en liste over interfacer som brukes av typen. Dette vil egentlig si at vi forventer at subtypene som realiserer dette portfacet, vil bruke disse interfacene for å realisere funksjonaliteten. Dermed blir dette et krav som omgivelsene må tilfredsstille for at et objekt av denne subtypen skal være i en lovlig tilstand. På mange måter gjelder det samme for ”operations” og ”usedOperations”. Dette er lister over operasjoner som tilbys eller brukes av typen. Legg merke til at ”operations” er en composition, mens ”provides”, ”uses” og ”usedOperation” er lister med referanser til objekter som eies av andre. ”ports” er en liste over alle portene som eies av portfacet. I metamodellen

er Portface det eneste elementet som eier porter og den blir derfor sentral i composite-strukturer.

For å beskrive felles egenskaper med hensyn på støttede operasjoner har vi begrepet interface. Dette begrepet sier ingen ting om hvordan metodene er implementert eller direkte oppførsel annet enn at vi vet hvilke operasjoner som tilbys. En part er, i en composite-struktur, et element som har porter og interfacer. En part har porter som tilbyr og bruker interfacer. Dette er et nytt nivå i modelleringen og det vil vise seg nyttig å kunne beskrive mer abstrakt enn bare via én klasse som passer denne beskrivelsen. Vi kan jo ha objekter som er komplekse selv om vi er innenfor et system med enkel arv. Et universitet vil kunne fylle rollene som en arbeidsplass, undervisningssted, forskningsmiljø og en organisasjon. I forskjellige situasjoner vil dette objektet kunne ta forskjellige roller.

Forøvrig kan vi nevne at Portface spesialiseres av Class og at en Part er typet med Portface som vist i Figur 4.7.



Figur 4.7 Utdrag av metamodell med omgivelsene til Part

En part er en spesialisering av ConnectableElement som er endepunktene for en connector. Part eies av Class og types med Portface. Vi ser også at Class kan realisere mange portfacer akkurat som med interfacer.

Når en formell parameter på en metode er typet med et interface vil det føre til at alle objekter som støtter dette interfacet kan være en aktuell parameter til denne metoden. Akkurat på samme måte gjelder det for Parter som er typet med Portface. Når en part er typet med et portface, vil alle objekter av en klasse som realiserer dette portfacet kunne inngå som en del av denne parten.

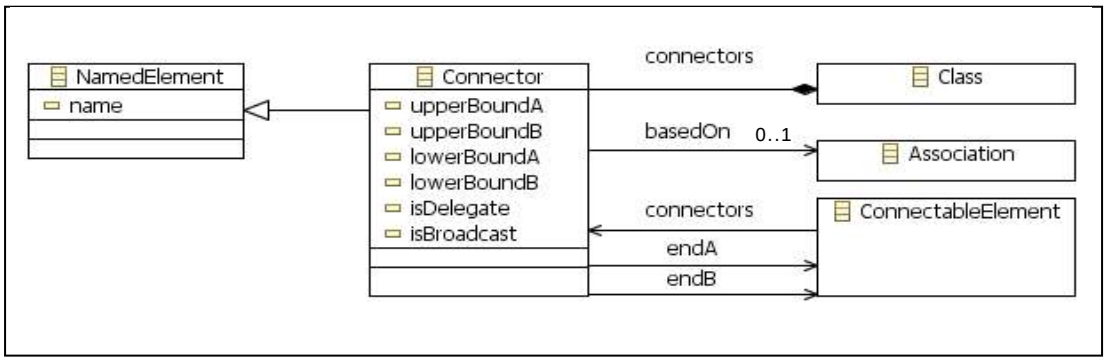
De sammenhengene som er omtalt i dette avsnittet er brukt i praksis i eksemplet om Utdanningsdepartementet i kapittel 3.1, som gjennomgås i kapittel 5.0.

4.3 Connector

UML 2s Connector er utgangspunktet for Connector i dette modelleringsspråket. En Connector knytter sammen to Parter, og dette skjer direkte eller via to PortApplicationer. Connectorene eies av klasser og fungerer på mange måter som en infrastruktur internt i en klasse. En

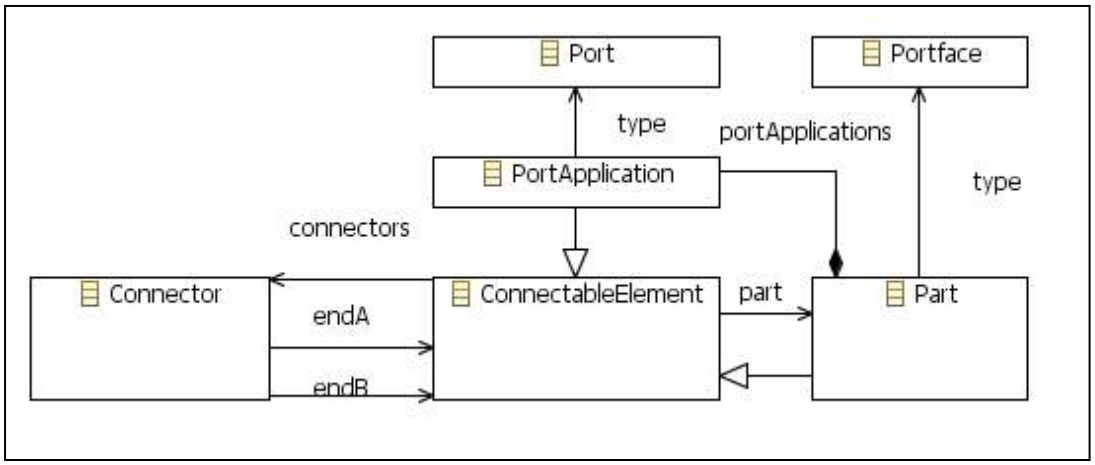
connector knytter partene sammen til et kommunikasjonsnettverk slik at partene kan samarbeide om å realisere klassens funksjonalitet.

En connector kobler sammen to ConnectableElementer via ”endA” og ”endB”. ConnectableElement er en abstrakt klasse som blir spesialisert av Part og PortApplication (Figur 4.9). En connector kobler enten sammen to PortApplicationer, to Parter eller en av hver.



Figur 4.8 Utdrag av metamodell med omgivelsene til Connector

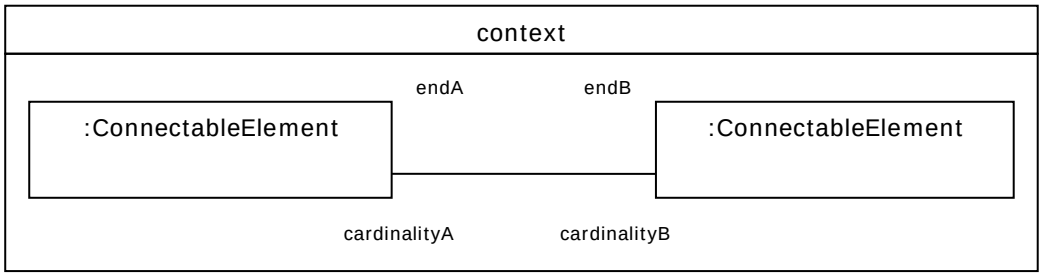
Figur 4.8 viser at en connector ikke er en type, men at den kan være en nærmere beskrivelse av hvordan en association blir applisert. En connector har to ender og hver av endene er et ConnectableElement. Dette er nærmere illustrert i Figur 4.9.



Figur 4.9 Utdrag av metamodell med elementer for composite-struktur

Figur 4.9 viser at Part og PortApplication er spesialiseringer av ConnectableElement. Part er typet med et Portface og en PortApplication er typet med en Port.

Connectoren holder rede på hvilke objekter som er koblet sammen på samme måter som en association. Derfor har også Connectoren en øvre og en nedre grense på kardinalitetene (Figur 4.8 og Figur 4.10).



Figur 4.10 Prinsipptegning av omgivelsene til en Connector

Det er streken i Figur 4.10 som representerer connectoren. Om en connector er knyttet til en part av ”reference” type, vil connectoren kunne oppleve at en instans i parten tas ut av parten. Selv om instansen fortsatt eksisterer vil den ikke opptre i den situasjonen som er gitt som premisser for connectorens beskrivelse, og derfor må vi anse instansen for borte og slette alle linker som refererer denne instansen.

Når et objekt gjør et metodekall kan metodekallet gjøres på en port eller direkte i parten. Et metodekall på en port vil føre til et metodekall via hver av de connectorene som er tilkoblet denne porten, mens et kall direkte på parten vil føre til bruk av alle connectorer som er koblet direkte til parten.

Denne oppgaven har ikke gått inn på retningsorienterte connectorer. En connector med retning er et spesialtilfelle av en vanlig connector og vil på de fleste måter fungere likt, selvsagt med det unntaket at den tar imot metodekall bare fra den ene enden.

Et metodekall som blir formidlet til en connector, blir videreformidlet til de objektene som initiatoren av kallet er koblet til via denne connectoren. Hele denne prosessen er beskrevet i kapittel 4.5.6.

4.3.1 Based on

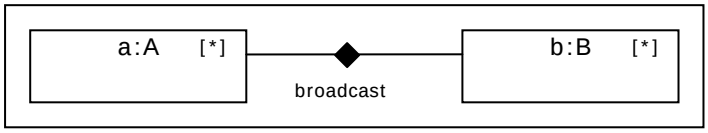
Om en connector er basert på en association blir dette beskrevet i referansen ”basedOn” i Figur 4.8. Om referansen peker på en association så er connectoren basert på denne Associationen. En connector som er basert på en association fungerer som en vanlig connector med det unntaket at hver link må ha en tilsvarende relasjon i associationen. Hver link har en instans av associationsklassen knyttet til de to objektene. En association gjelder for alle objektene av de involverte klassene, mens en part beskriver objektene i en spesiell situasjon, hvor objektene kan være fordelt på flere parts. Dette fører til at selv om en connector er basert på en association så snakker vi om at connectoren arbeider på en delmengde av associationslinkene. Her finnes det to mulige tolkninger, hvor den ene tolkningen gir eksplisitt de relasjonene vi ønsker skal ha en link, mens den andre tolkningen er å lage connectoren transparent slik at alle relasjoner mellom de tilstedeværende objektene blir linket. Runtime-plattformen i denne oppgaven bruker bare den første tolkningen.

I det tilfellet hvor connectoren er basert på en association kan et metodekall kunne tilby flere muligheter enn ved en vanlig connector.

Årsaken er at koblingen i connectoren er fundert på en link av en association, og denne linken kan ha informasjon. Vi har et associationsobjekt som kan tilby informasjon og tjenester som er entydig definert i denne sammenhengen, som beskrevet i eksemplet om betaling av lønn i Community modellen i kapittel 6.0. Her forsyner associationsobjektet informasjonen om lønnen og utfører selve transaksjonen ut i fra den informasjonen de har.

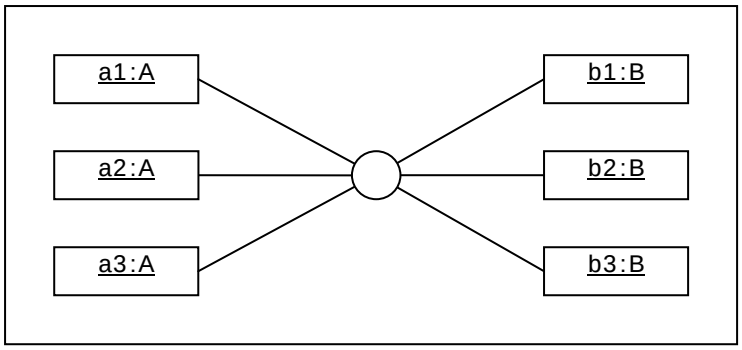
4.3.2 Broadcast-connector

En broadcast-connector er en connector som linker alle objektene på den ene siden med alle objektene på den andre siden. Figur 4.11 viser den konkrete syntaksen vi bruker i denne oppgaven for å vise denne egenskapen ved en connector.



Figur 4.11 Konkret syntaks for broadcast-connector

Vi har beskrevet linker som en bipartitt relasjon og med denne synsvinkelen vil en broadcast connector se ut som i Figur 4.12.



Figur 4.12 Illustrasjon av linkene til en broadcast-connector

Den viktige forskjellen er at for vanlige connectorer og associationer blir hver link definert som et par instanser, med en instans fra hver side. Men for en broadcast connector vil alle mulige koblinger til en hver tid eksistere, med den ene betingelsen at den fortsatt er en bipartitt relasjon. Eksemplet Undervisningsdepartementet i kapittel 3.1 bruker broadcast-connector og dette blir gjennomgått i kapittel 6.0.

4.3.3 Delegate-connector

En delegate-connector kan ses på som en forlengelse av en annen connector. Den blir brukt der en ytre klasse skal tilby noe den ikke selv tilbyr, men som en part tilbyr. Delegate-connectorer delegerer ansvaret for interfacet eller tjenesten til en annen som tilbyr det. Dette kan på mange måter sammenlignes med det å viderekoble en telefonlinje til en annen telefon.

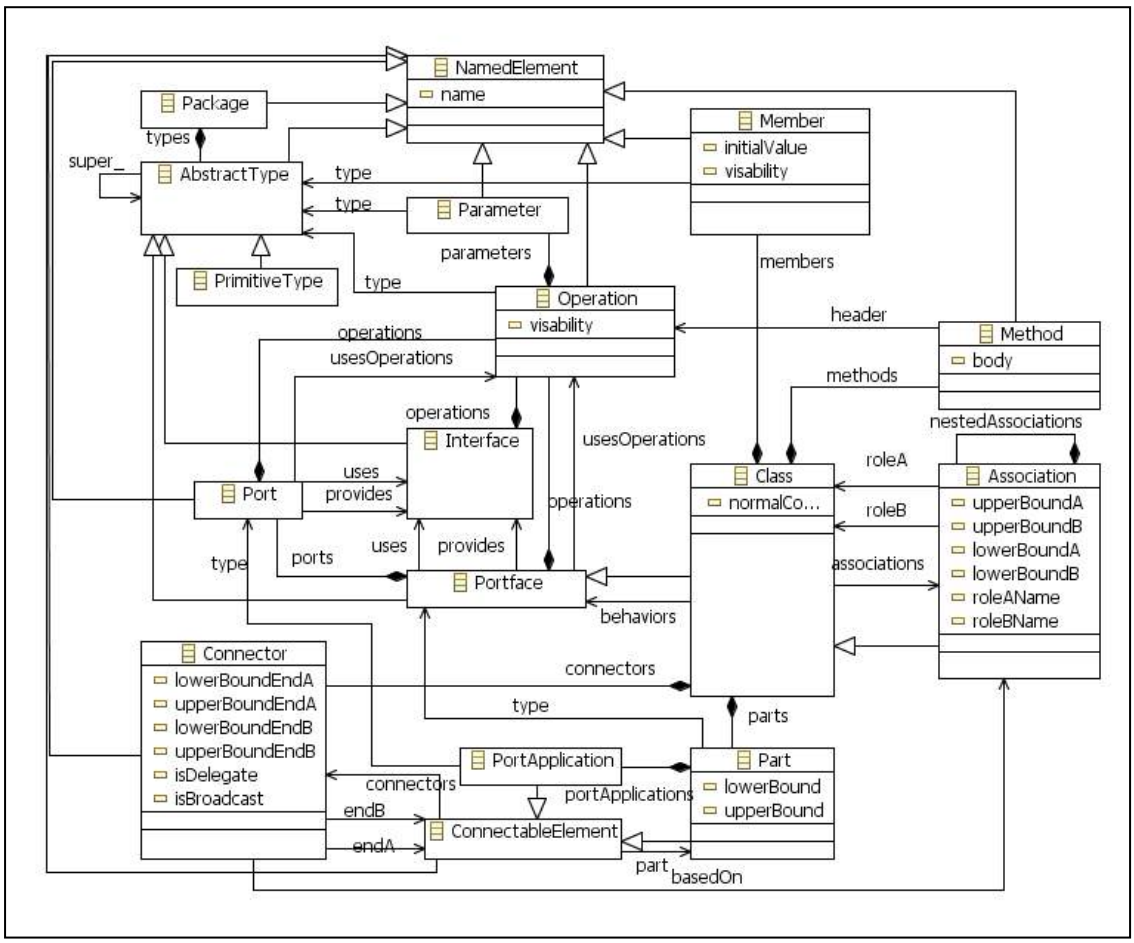
En delegate connector må ha øvre og nedre grense på begge sider lik 1, og parten som delegeres må også ha øvre og nedre grense lik 1.

En delegate connector kan i UML beskrive hvilke interfacer som blir delegert. Dette vil i liten grad påvirke den grunnlegende funksjonaliteten. en retningsorientert connector vil kunne delegere spesifikke interfacer og operasjoner til forskjellige steder, men den grunnlegende funksjonaliteten vil ikke bli endret av den grunn.

Delegate-connector blir brukt i et aksempel i problemstillingen i kapittel 2.0. Dette eksemplet blir gjennomgått i kapittel 6.0.

4.4 Metamodell

Etter at vi nå har presentert det aller viktigste ved modelleringsspråket er det på tide å se på helheten. Metamodellen er modelleringsspråkets abstrakte syntaks og setter rammene for det språket omfatter og kan beskrive. Metamodellen er beskrevet i sin helhet tekstlig i kapittel 10.0, men her presenterer vi metamodellen grafisk og med tilhørende forklaringer. Nå kan vi se hvordan Portface, Association, Class, Part og Connector føyer seg inn i en helhet sammen med velkjente objektorienterte begreper som Class, Member, Interface og Method.



Figur 4.13 Metamodell

Figur 4.13 viser den abstrakte syntaksen til modelleringsspråket. Språket er utarbeidet for å kunne modellere med klasser, associationer, parts og connectorer på en helhetlig måte. Som vist i Figur 4.13 er metamodellens egenskaper forenklet i forhold til UML, men den inneholder fortsatt de essensielle egenskapene vi er ute etter i objektorientert modellering med klasser.

For å skille på metodens signatur og selve impementasjonen har vi valgt å benytte navnet Operation for å beskrive metodens signatur. Selve implementasjonen er gitt navnet Method, og er Java-kode som er lagret som tekst i attributten body i Method-elementet. Denne Java-koden er implementasjon av oppførsel, algoritmer og slikt. For å utvikle denne Java-koden trengs kunnskap om implementasjonsspesifikke ting fra runtime-plattformen. Dette er ikke ideelt, men er et akseptabelt kompromiss når målet ikke er å lage et ferdig produkt men å komme frem til en tolkning av noen elementer som fungerer på en god måte. Informasjon om API som kan brukes i denne Java-koden er gitt i kapittel 4.5.7.

UML er et rikt og omfattende språk som har en stor og kompleks metamodell. Metamodellen i denne oppgaven er ikke lik UML 2s metamodell. Dette ble gjort for å kunne ha større oppmerksomhet på de egenskapene som dette prosjektet omhandler.

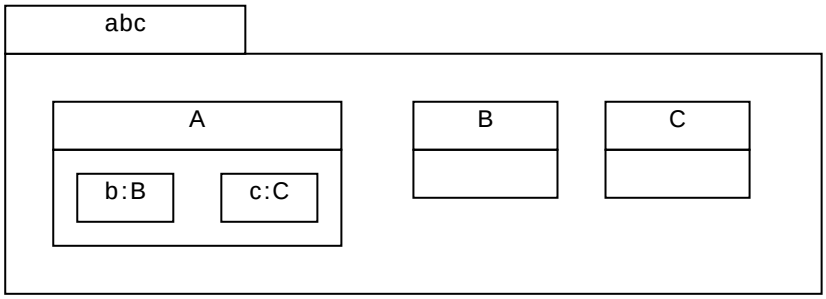
Det finnes også en rekke forenklinger i metamodellen. Den støtter ikke signaler, asynkrone kall eller tilstandsmaskiner. Oppførselen til programmet er heller ikke en del av metamodellen men er lagret som tekst i modellen og er ren Java-kode. Forenklingene er igjen gjort for å minimalisere omgivelsene samtidig som vi beholder essensielle egenskaper for å undersøke associationer og connectorer mellom porter og parter. Selv om UML er et omfattende språk var det egenskaper UML ikke hadde som ble sentrale i modelleringen av eksemplene.

En annen grunn til at metamodellen avviker fra UMLs metamodell er at vi ønsket å bruke teknologi som Eclipse, Java, ATL, MOFScript og Ecore. Når hele rammeverket med metamodellen skulle implementeres i Java var det en stor fordel at metamodellen kun benyttet seg av enkel arv. Siden metamodellen til UML 2 er en flittig bruker av multippel arv, så medfører det en del strukturforskjeller mellom de to metamodellene.

4.5 Runtime-plattform

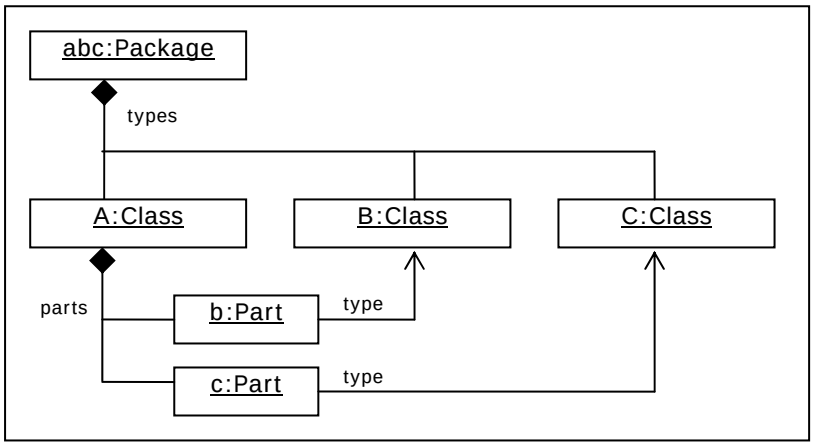
Runtime-plattformen gir modellen liv og mening. Den er en implementasjon av tolkningen av alle elementene i metamodellen, slik at når programmet kjøres så blir de forskjellige elementene i modellen behandlet etter den tolkningen som modelleringsspråket bygger på. Runtime-plattformen er implementert i Java og bygger på grunnleggende Java-teknologi.

Tidligere har vi nevnt at den genererte Java-koden, som er referert til som ”Program” artefakten, er todelt. Den ene delen er ansvarlig for å opprette en objektstruktur i Java-runtime som vi har kalt ”Structure Model”. Nå skal vi forklare hva denne strukturen er og hvordan den blir laget.



Figur 4.14 Eksempel på en modell

For å beskrive hva"Structure Model" er har vi laget en eksempelmodell, som vist i Figur 4.14. Modellen har klassene A, B og C, hvor klasse A har en part typet med hver av klassene B og C.



Figur 4.15 Eksempel på strukturmodell av eksempelmodell

Figur 4.15 viser et eksempel på hva strukturmodellen inneholder og hvordan det vil se ut med modellen gitt i Figur 4.14. Pakken "abc" eier de tre typene A, B og C. Klassen "A" har de to partene "b" og "c" og de er typet med henholdsvis B og C. Når vi studerer disse to figurene kommer det klart frem at strukturmodellen beskriver modellens struktur.

```
Class A extends ClassFundation { ... };
Class B extends ClassFundation { ... };
Class C extends ClassFundation { ... };

Class initClassA {
      initClassA() {
            Class_ A = new Class_( “A” );
            A.addPart( new Part( “b”, ”B” ) );
            A.addPart( new Part( “c”, ”C” ) );
            Package.addClass( A );
      }
}
Class initClassB {
      initClassB() {
            Class_ B = new Class_( “B” );
            Package .addClass( B );
      }
}
Class initClassC {
      initClassC() {
                        Class_ C = new Class_(
“B” );
            Package.addClass( C );
      }
}
```

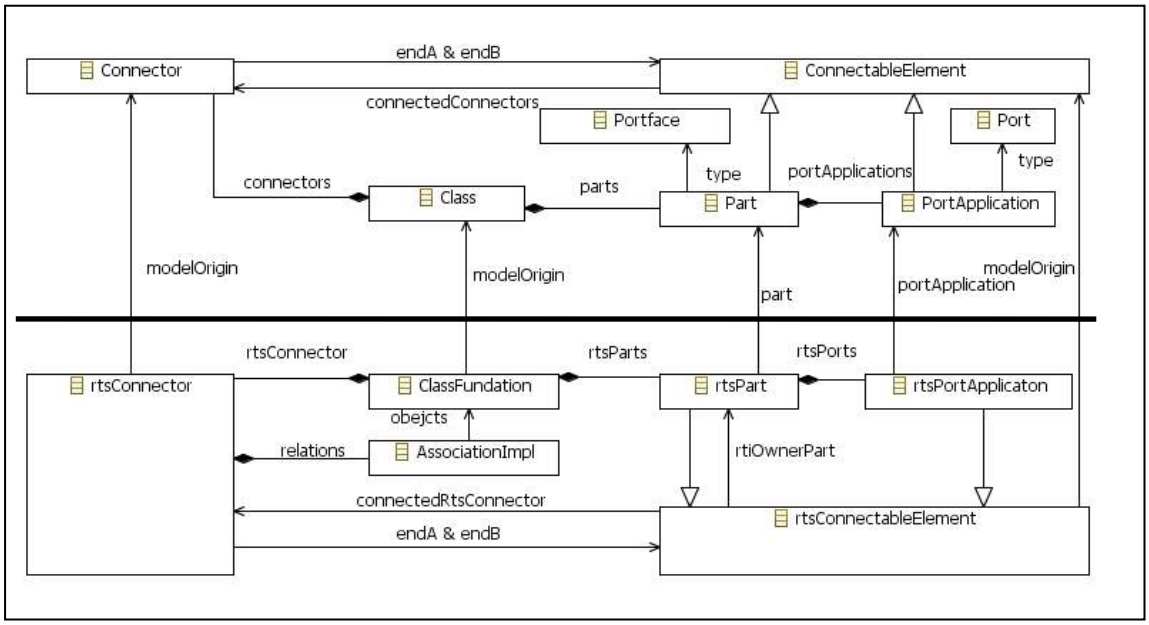
Figur 4.16 Skisse av Java-koden ut fra eksemplet

I Figur 4.16 ser vi en skisse av Java-koden som ville komme ut av Modell-til-kode scriptet med modelleksemplet som input. Vi ser klart at koden er delt i to, hvor de tre øverste klassene ”A”, ”B” og ”C” er Java-implementasjonen av klassene. Deretter kommer vi til tre klasser som er implementasjon for å opprette strukturmodellen. Konstruktøren i klassen ”initClassA” lager et Class_ objekt med navn A og legger to parts inn i dette objektet. Partene blir laget med navn og type. Til slutt i konstruktøren til ”initClassA” legges Class_ objektet til pakkens typer. De to siste klassene oppretter hvert sitt klasseobjekt som beskriver hver sin klasse. Disse blir så lagt til som typer i pakken og dermed er klassene B og C også registrert i strukturmodellen.

For å beskrive litt nærmere hva denne runtime-plattformen er, så er det en plattform som tilbyr de egenskapene modelleringsspråket har. På lik linje med at Java vet hvordan et metodekall skal finne riktig kode og hvordan to strenger skal legges sammen, så vet runtime-plattformen hvordan en part oppfører seg og hvordan en connector kan delegere et interface.

4.5.1 Runtime-plattformens oppbygning og bruk av strukturmodellen

Nå skal vi se på hvordan runtime-plattformen virker, og hvordan den bruker strukturmodellen for å tolke modellen. Navnet runtime-plattform er valgt fordi den består av klasser som instansieres og lager utførende elementer som implementerer egenskapene til associationer, connectorer, parter og porter. Disse er implementert ut ifra den tolkningen modelleringsspråket har av disse. Det hele fører til en plattform som kan kjøre et Java-program som bygges med elementer som vanligvis ikke er tilgjengelige i programmeringsspråk.



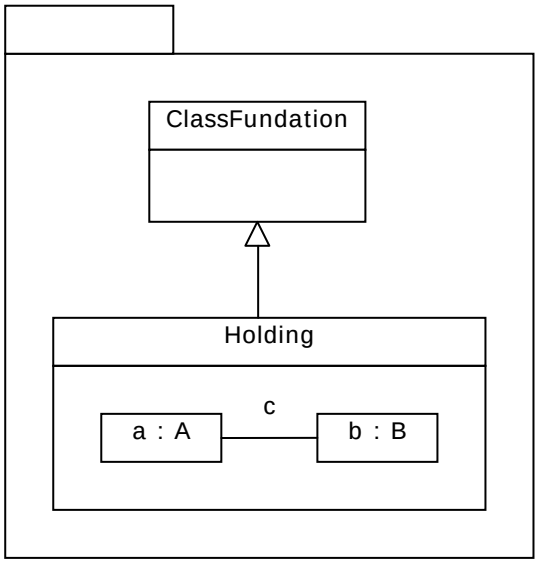
Figur 4.17 Hvordan runtime klassene er koblet til metamodellen

Figur 4.17 viser et klassediagram over hvordan klassene i runtime-plattformen bruker modellelementene fra strukturmodellen. Diagrammet er delt i to, hvor den øvre delen inneholder begrepene som er definert i metamodellen. Dette er Java-implementasjonen av metamodellelementene som i Figur 4.2 på side 29 blir kalt for ”Metamodel Library” og som tilhører M2. Under streken finner vi klassene som til sammen utgjør runtime-plattformen og de tilhører metanivå M1.

Når vi lager instanser av en klasse havner vi ett metanivå ned. Når vi instansierer klassene over streken så får vi en strukturmodell som tilhører metanivå M1, mens instanser av klassene under streken får vi når vi kjører programmet i Java som tilhører metanivå M0. Det kjørende programmet i M0 er vist som ”Running program” i Figur 4.2 side 29.

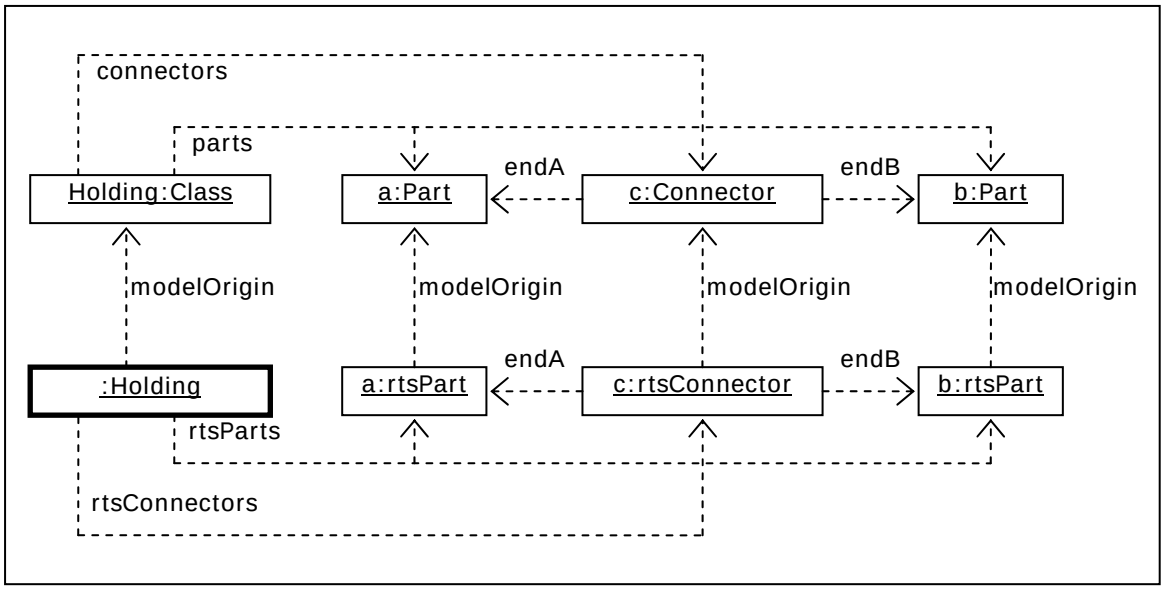
I skissen av Java-koden i Figur 4.16 ser vi at de tre første klassene spesialiserer klassen ClassFoundation som vi finner igjen i Figur 4.17 under streken. Alle brukerdefinerte klasser og associationer er en utvidelse av ClassFoundation og implementerer dermed også et interface som heter rtsObject. Dette fører til at alle instanser av brukerdefinerte klasser og associationer er en subtype av ClassFoundation og støtter interfacet rtsObjekt. Vi ser i diagrammet at ClassFoundation har en referanse ”modelOrigin” som er en referanse til et objekt i strukturmodellen som beskriver denne typen med metoder, attributter, connectorer, parter, interfacer og portfacer.

Når vi lager en instans av en brukerdefinert klasse, vil vi automatisk få en instans av ClassFoundation som har en referanse til beskrivelsen av klassen i form av et Class objekt. ClassFoundation sørger for at når instansen blir opprettet blir det automatisk opprettet objekter av rtsPart, rtsConnector og rtsPortApplication fra de respektive forekomster av Part, Connector og PortApplication.



Figur 4.18 Eksempel med klassen Holding med to parter

Figur 4.18 viser et eksempel på en brukerdefinert klasse ”Holding”, som har to parter a og b og en connector c. Som alle brukerdefinerte klasser er Holding en spesialisering av ClassFoundation. Konstruktøren til ClassFoundation tar inn et Class objekt i konstruktøren, og for Holding klassen vil dette Class objektet være en beskrivelse av Holding klassen som en del av strukturmodellen. Øvre del av Figur 4.19 viser denne strukturen.



Figur 4.19 Objektstruktur med et objekt av Holding

Figur 4.19 viser hvordan et objekt av Holding-klassen har en referanse til et Class objekt, og at ClassFoundation sørger for at runtime-objektene

trsPart og trsConnector blir opprettet i henhold til strukturbeskrivelsen av klassen i Class-objektet.

4.5.2 Oppstart og initiering.

Modellen som er laget i henhold til metamodellen blir transformert til Java-kode av et MOFScript. Denne Java-koden inneholder all informasjonen som modellen bestod av og tar i bruk runtime-plattformen slik at det blir et kjørbart program. Alle definerte klasser og associationer i modellen vil ha en tilsvarende Java-implementasjon.

Det første som gjøres er at all koden for å opprette strukturmodellen blir kjørt. Når strukturens elementer er opprettet finnes det mange referanser til elementer ved navn. Disse navnene blir brukt til å finne referansen til de navngitte elementene. Dette gjøres får å sy sammen modellen. Dette gjør at strukturen blir sammenhengende og at alle referanser er funnet før kjøring av selve programmet begynner. Årsaken til at dette gjøres er at elementene trenger å ha referanser på kryss av hierarkiet, og for at vi skal kunne ha gjensidige avhengigheter. Da er dette en enkel måte å løse det på. Når dette er gjort har vi en tilsvarende objektmodell i Java som beskriver strukturen i vår opprinnelige modell og programmet kan begynne.

Når en klasse instansieres blir objektet opprettet, og deretter kalles konstruktøren til ClassFoundation med klassens modellelement. Dette modellelementet har lagret all informasjon om denne klassen som rammeverket trenger for å opprette en instans riktig. Her ligger det informasjon om parter, connectorer, porter og portenes krav.

4.5.3 Objekter og rtsConnector

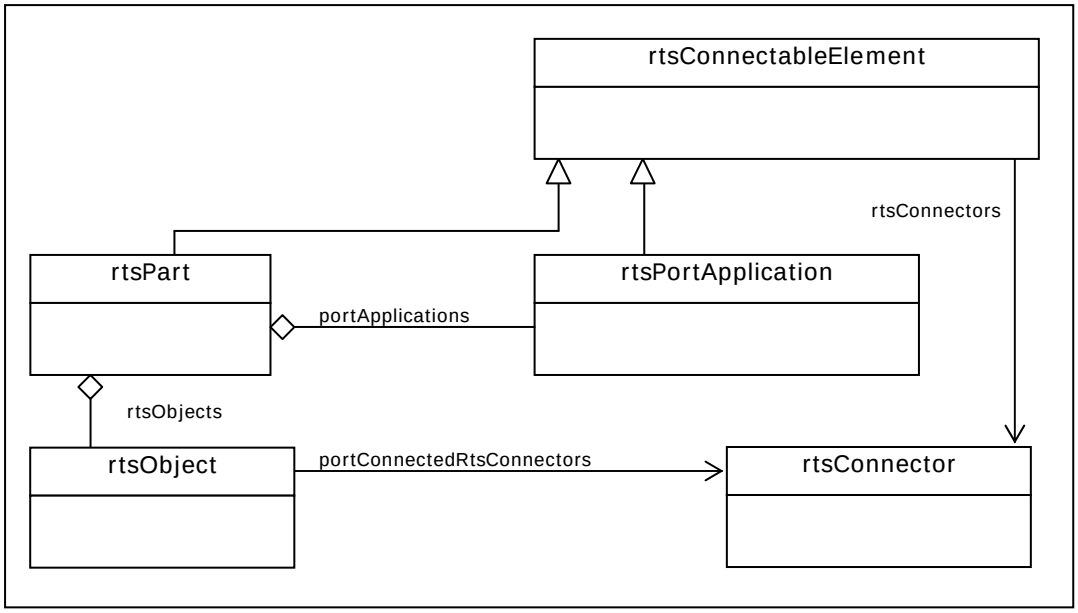
Connectorer er tilkoblet parter direkte eller via porter. Klassen rtsConnector er en runtime implementasjon av vår tolkning av en connector. Når et objekt er en del av en part er objektet knyttet til de rtsConnector objektene den kan bruke for å få tilgang til de forskjellige partene i ”nabolaget”. rtsConnectorene gir et objekt tilgang på portene og interfacene som blir tilbudt av andre objekter i andre parter. For at et objekt skal kunne bruke et annet objekt må de være koblet via connectoren ved at connectoren har informasjon om at de to objektene er koblet. I tillegg må objektene være en del av hver sin part, og disse partene må samtidig være knyttet sammen av en connectoren.

Koblingen til selve rtsConnector skal skje idet objektet blir lagt inn i en part som connectoren er koblet til, mens linken som knytter sammen de enkelte objektene med hverandre blir opprettet senere.

Om det er en broadcast Connector, trenger vi ikke mer informasjon enn hvilke objekter som er på hver side av rtsConnectoren. For da kan hvert objekt nå alle på den andre siden.

Derimot hvis det er en annen type connector så må koblingen mellom disse to objektene legges inn eksplisitt, med unntak av en delegator connector. En delegator vet vi skal kun ha én kobling og den er mellom en part og dens eier. Dermed blir denne koblingen automatisk laget, når et objekts referanse blir lagt inn i en part. Dette er fortalt i detalj i 4.5.4 Hva skjer når vi legger et objekt til en rtsPart. Kort sagt kan vi si at

connectoren blir holdt informert om hvilke objekter som kommer og går i partene på connectorens ender. Derfor har den til enhver tid oversikt over hvike objekter som er på hver side.

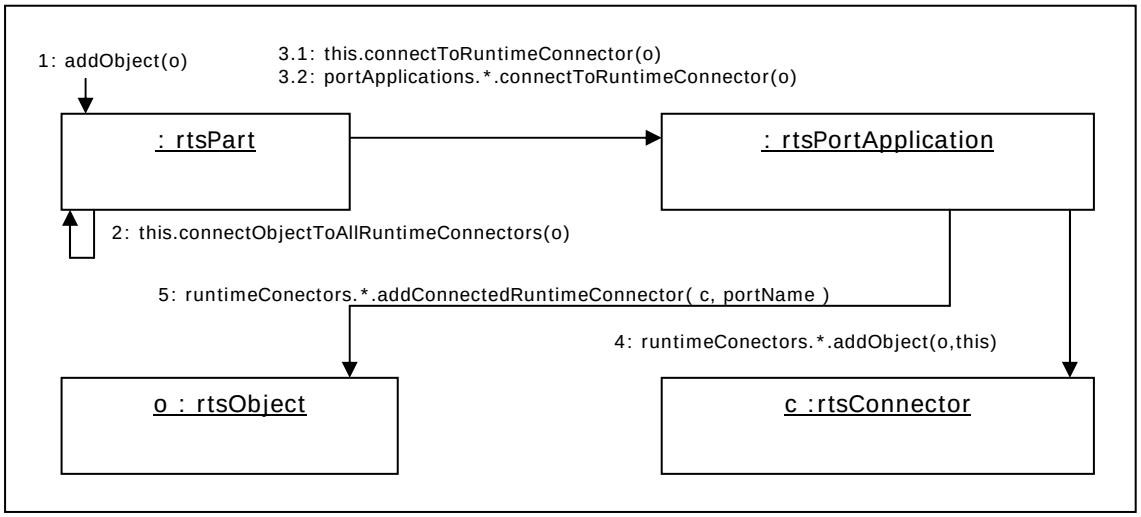


Figur 4.20 Klassediagram over omgivelsene til rtsConnector

Figur 4.20 viser den viktigste informasjonen om hvordan klassene forholder seg til hverandre når vi snakker om koblingen mellom objekter og connectorer.

4.5.4 Hva skjer når vi legger et objekt til en rtsPart

Klassen rtsPart er en implementasjon av vår tolkning av part. Når en klasse har en part vil objekter av denne klassen ha objekter av rtsPart. Nå skal vi se litt på hvordan rtsPart fungerer og oppfører seg.

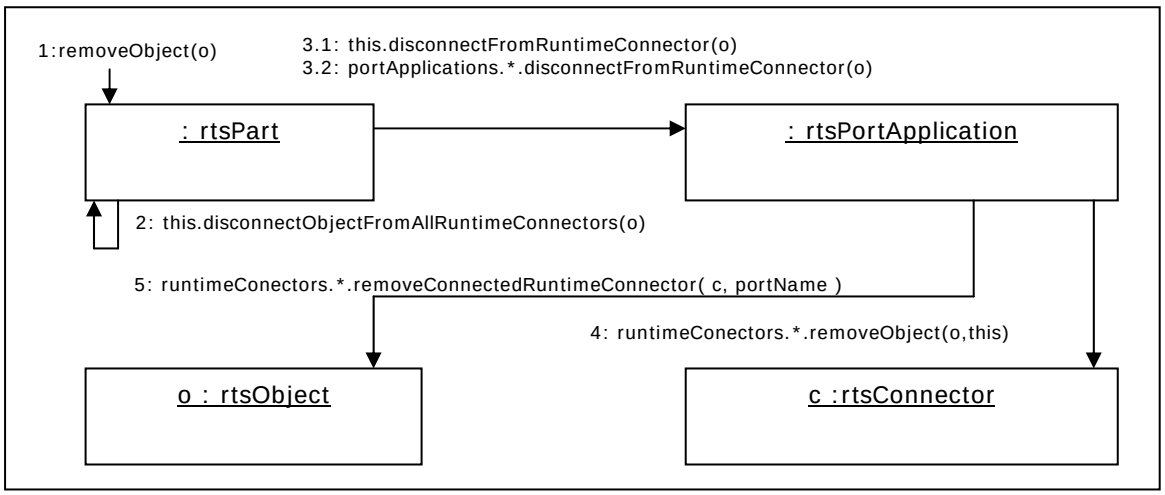


Figur 4.21 Objektdiagram over hvordan et objekt blir lagt til en part

Figur 4.21 viser hvordan kallgrafen blir når vi legger til et objekt i en rtsPart. Resultatet er at objektet blir koblet til de rtsConnectorene som den skal ha tilgang til. rtsConnectorene blir koblet til slik som modellen beskriver, ved bruk av porter, dvs. at objektets logikk bare får tilgang til rtsConnectorene ved å bruke de rette portene. Kallet 3.1 angir at en connector kan være koblet direkte til en part og ikke bare via en portApplication. Begge kallene 3.1 og 3.2 fører til kall 4 og 5.

4.5.5 Hva skjer når vi fjerner et objekt fra en rtsPart

Nå har vi sett hvordan vi legger et objekt inn i en rtsPart, så for å slutte ringen skal vi nå ta for oss hvordan et objekt blir fjernet fra en rtsPart.

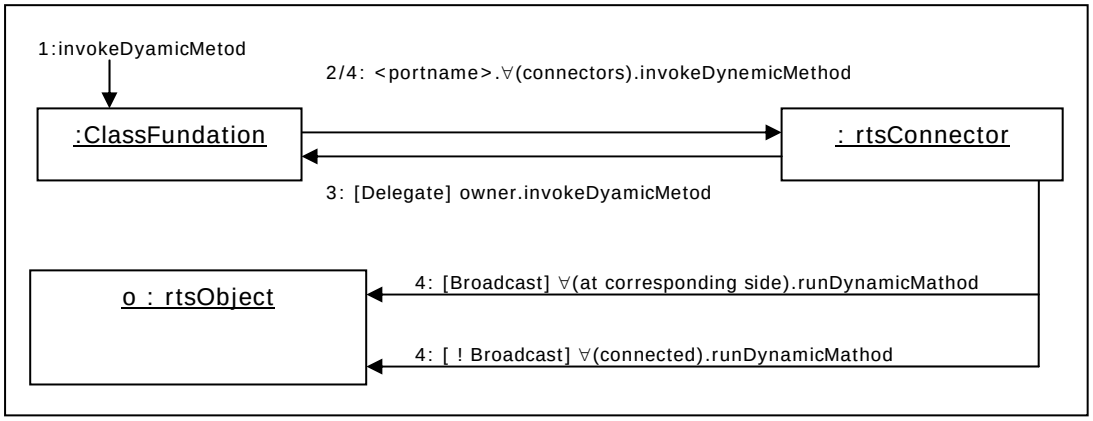


Figur 4.22 Objektdiagram over hvordan et objekt blir fjernet fra en part

Figur 4.22 viser at kallgrafen for å fjerne et objekt fra en rtsPart er veldig lik den for å legge til et objekt. Den eneste forskjellen er at metodenavnene reflekterer en annen aktivitet.

4.5.6 Aktivering av Metoder

Aktivering av en metode begynner ved at en kodelinje kaller på en metode som objektet skal få tilgang til via en connector. Metoden blir lokalisert ved et kall på metoden på en port eller, om den ikke er på en port, via nøkkelordet ”that”. Nøkkelordet that er et antonym til begreper som ”this”, ”me” og ”self”. Disse begrepene brukes om noe som gjelder internt i det gjeldende objektet. ”that” betyr at vi søker noe utenfor objektet, i denne sammenheng noe på den andre siden av en eller annen connector.



Figur 4.23 Aktivering av metode

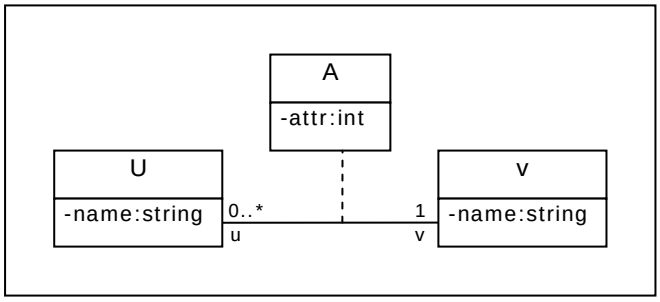
Kallet til porten blir realisert av et medlem med portens navn, som er en instans av en indre klasse. Denne instansen implementerer en metode-stub for å gjøre det virkelige dynamiske kallet. Dette kallet blir gjort på samlingen av connectedRtsConnector på PortApplicationen med porttypen med samme navn som portnavnet. Disse rtsConnector finner så ut om den andre enden av connectoren støtter metoden, og om den gjør det, blir det kjørt et dynamisk kall på metoden på alle objekter som er koblet til initiatoren. Når det dynamiske kallet blir mottatt blir det oversatt til et vanlig metodekall lokalt i dette objektet.

4.5.7 Runtime-plattform API

De foregående delkapitlene har beskrevet hvordan utvalgte aktiviteter blir utført i runtime-plattformen. Det er viktig å kunne legge et objekt til en part og kunne opprette en link mellom to objekter. Uten tilgang på disse aktivitetene er det liten vits i å ha støtte for parter, connectorer og associationer.

Tilgangen til disse viktige aktivitetene blir gitt på to måter, den ene er at en del aktiviteter for associationer er gjort tilgjengelig via et fast API for associationer. Den andre måten er at Modell-til-kode scriptet bruker modellen og skreddersyr et API for utvikling av kode i denne modellen. Dette fører til at navnene til partene og portene kan brukes direkte i

koden. Hvordan dette gjøres blir forklart litt senere, nå skal vi se hvordan associationene blir brukt.



Figur 4.24 Eksempel på association

For å forklare hvordan APIet skal brukes skal vi gi noen eksempler med utgangspunkt i Figur 4.24. Vi har klassene U og V, hvor ett V objekt kan være koblet til flere U objekter. Associationen A har en attributt med navnet attr som er av typen int. Vi skriver litt eksempelkode:

```
// to make the objects u, u2 and v
U u  = new U( “u” );
U u2 = new U( “u2”);
V v = new V( “v” );

// making the link between u, v. with the integer 10
new A( u, v, 10 );

// making the link between u, v. with the integer 7
new A( u, v,  7 );

// making links between u2, v. with the integer 200 and 400
new A( u2, v, 200 );
new A( u2, v, 400 );

// getting the implementation of the association A
AssociationImpl ai = Association.getAssociationSet("A");

// getting the array of the links between u and v
Collection<RuntimeObject> rc = ai.getAssociations ( u, v );

System.out.println( “All between u and v“ );
A a;
for( RuntimeObject o : rc ) {
    a = (A) o;
    System.out.println( “----“ );
    System.out.println( “A value: ” + a.getAtt( ) );
    System.out.println( “U name: ”  + a.getU().getName() );
    System.out.println( “V name: ”  + a.getV().getName() );
}

// getting the array of the links with v
rc = ai.getAssociationsFromB ( v );

System.out.println( “/nAll linked with v“ );
for( RuntimeObject o : rc ) {
    a = (A) o;
    System.out.println( “----“ );
    System.out.println( “A value: ” + a.getAtt( ) );
    System.out.println( “U name: ”  + a.getU().getName() );
    System.out.println( “V name: ”  + a.getV().getName() );
}
```

```
All between u and v
----
A value: 10
U name: u
V name: v
----
A value: 7
U name: u
V name: v

All linked with v
----
A value: 10
U name: u
V name: v
----
A value: 7
U name: u
V name: v
----
A value: 200
U name: u2
V name: v
----
A value: 400
U name: u2
V name: v
```

Figur 4.25 Eksempelkode t.v. og printout t.h.

Figur 4.25 viser eksempelkode som bruker klassene og associationen illustrert i Figur 4.24. Koden er ikke så vanskelig. Først oppretter vi de tre objektene u, u2 og v, som er av henholdsvis klassene U , U, og V. Objektene blir gitt navn som er i overensstemmelse med variabelnavnene de er er gitt i koden. Det neste som skjer er at vi lager linker. Vi lager to linker mellom objektene u og v. Så lager vi to linker mellom u2 og v. Dette gjør at u og u2 er linket til to linker hver, mens v er linket til 4 linker.

Nå har vi laget alle linkene våre, så nå er det på tide å finne de igjen. Først skaffer vi oss implementasjonen av associationen A. Denne kaller vi ”ai”. Vi bruker ”ai” for å finne alle linker mellom objektene u og v. For å finne ut hva vi fikk, lager vi en loop for å printe ut litt informasjon. Fra linken finner vi objektene som er linket med denne linken ved å bruke metodene get<roleNameA>() og get<roleNameB>(). Attributtene til både klassene og associationene kan hentes med metodene get<attributeName>(). I dette eksemplet blir metodene hetende getName() for å hente attributtene fra klassene U og V. For å hente informasjon fra linkene av associationen A må vi bruke metodene getU() og getV() og getAttr().

Når vi ber om å få alle linker mellom objektene u og v får vi, som vi ser av utskriften til høyre i Figur 4.25, to linker. Linkene har verdi 10 og 7. Dette

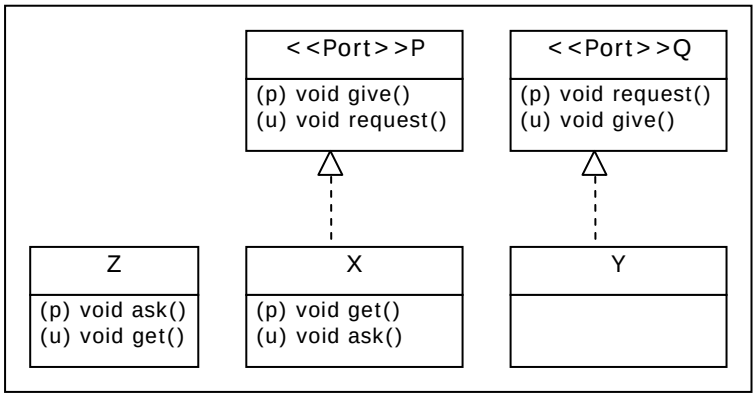
stemmer godt overens med koden. Deretter ber vi om å få alle linker objektet v er involvert i, og da får vi fire linker. To linker som er mellom objektene u og v, og to linker som er mellom objektene u2 og v.

Generelt kan vi si at de fleste ønskede funksjonene er tilgjengelig når implementasjonen av associationen er tilgjengelig. Her kommer en oppsummering av APIet som er tilgjengelig via en implementasjon av en association:

- hasAssociation(u, v) : Denne metoden sjekker om instansene u og v er assosiert med denne associationen.
- getAssociations(u, v) : Denne metoden returnerer en liste med alle linkene som linker instansene u og v. Som oftest er den enten tom eller inneholder kun ett element. Med spesialisering av associationer og associationer som ikke er unike, vil det kunne forekomme flere linker mellom de to instansene.
- getManyPartnersFromA(u) : Denne metoden vil gi en liste over alle instanser u er assosiert med via denne associationen. Dette kallet gir dog ingen mening om det er tillatt for u å være assosiert med kun én.
- getOnePartnerFromA(u) Denne metoden vil returnere den instansen u er assosiert med via denne associationen om det er noen. Dette kallet gir dog ingen mening om det er tillatt for u å være assosiert med flere.
- getManyAssociationsFromA(u) : Denne metoden gir en liste over alle linkene u er med i via denne associationen.
- getManyPartnersFromB(v) : tilsvarende som for getManyPartnersFromA(u), bare for den andre siden .
- getOnePartnerFromB(v) : tilsvarende som for getOnePartnerFromA(u), bare for den andre siden .
- getManyAssociationsFromB(v) : tilsvarende som for getManyAssociationsFromA(u), bare for den andre siden .
- removeAssociation(u, v, a) : Denne metoden fjerner linken a mellom u og v i denne associationen.
- removeA(u) : Denne metoden brukes for å fortelle at instansen u skal fjernes helt fra associationen. Typisk brukt når en instans skal dø.
- removeB(v) : Samme som for removeA(u), bare for den andre siden.

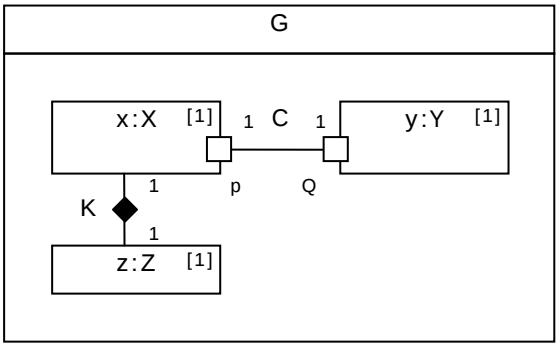
Associationsimplementasjonen er det mulig å få tak i via et kall til klassen Association: ”Association.getAssociationSet("<AssociationName>")”

Nå har vi fortalt om bruk av associationer, så nå er det på tide å forklare hvordan bruk av porter og connectorer foregår i kode. Først har vi et eksempel som vi forklarer og etterpå kommer en generell beskrivelse av bruken.



Figur 4.26 Klassediagram over eksemplet

Figur 4.26 viser et klassediagram over hvordan klassenes grensesnitt er. Klassen Z tilbyr metoden ask og bruker metoden get. Klassen X er helt motsvarende til klasse Z. X tilbyr metoden get og bruker metoden ask. Klassen X har i tillegg en port P som tilbyr metoden give og bruker metoden request. Port P har også en motsvarende port Q som tilbyr metoden request og bruker metoden give. Klassen Y har porten Q. Når vi nå viser en composite-struktur vil vi se hvordan objekter av disse klassene kommuniserer.



Figur 4.27 Composite-struktur for eksemplet

Figur 4.27 viser en composite-struktur hvor partenes typer er beskrevet i Figur 4.26. Parten x har multiplisitet 1 og er direkte koblet til parten z via en broadcast-connector. Dette fører til at de to objektene i partene x og z blir koblet uten at denne informasjonen blir gitt. Parten x og parten y er koblet sammen via connectoren C som er tilkoblet portene P og Q. Denne connectoren er en vanlig connector og trenger derfor informasjon om hvilke objekter den skal koble. I dette tilfellet er det ganske opplagt. Siden begge partene har kardinalitet 1 vil det måtte være de to objektene som er i henholdsvis part x og part y som skal kobles. Dette spesialtilfellet har ikke implementasjonen av connectoren tatt hensyn til, så vi må informere om hvilke objekter som skal kobles. Vi får nå se en skisse av hvordan kodene i de forskjellige klassene kan se ut.

```
class G {

  // constructor
  G( ) {

    // making the three objects
    X x = new X( );
    Y y = new Y( );
    Z z = new Z( );

    // adding the three object to the parts
    addXObject( x );
    addYObject( y );
    addZObject( z );

    // connect x and y
    addCLink( x, y );
  }
}
```

```
class Z {
  void ask( ) {
    that.get( );
  }
}

class Y implements Q{

  void request( ) {
    // using the port Q
    Q.give( );
  }
}

class X implements P{
  void method( ) {
    that.ask();
    P.request();
  }
}
```

Figur 4.28 Implementasjon fra klassene i eksemplet

Til venstre i Figur 4.28 ser vi deler av implementasjonen av klassen G som har de tre partene x, y og z. Koden er ment som et utdrag av konstruktøren til klassen G. Det første som gjøres er å lage tre objekter, ett av hver klasse, og så legge de til partene. Vi ser at metoden for å legge til et objekt til parten x er ”addXObject”. Etter at alle tre objektene er lagt til i sine respektive parter, er det på tide å koble objektene sammen via connectoren. Vi kobler objekt x og y via connectoren C med metodekallet ”addCLink(x, y);”

Til høyre i Figur 4.28 har vi deler av implementasjonen av klassene X, Y og Z. Klassen Z implementerer metoden ”ask”, som brukes av klasse X via connectoren K. Metoden ”ask” gjør et metodekall på ”get” som klassen Z har tilgang på via connectoren K. Her ser vi en praktisk bruk av ”that” som ble kort beskrevet tidligere. Når i implementerer klassen Z vet vi fra definisjonen at vi skal ha tilgang på metoden ”get” via en eller annen connector som er direkte koblet på parten instansen er en del av. Med instansen mener vi en instans av klassen Z. Ved å bruke ”that” forteller vi at dette er et kall på en metode som ikke tilbys lokalt men fjernt. Uttrykk som ”this”, ”self” eller ”me” blir ofte brukt for å snakke om noe lokalt i objektet, og da passer det bra å bruke ”that” for å fortelle at det ikke er lokalt. Klassen Y implementerer metoden ”request” som tilbys via porten Q. Metoden blir implementert med et metodekall og det er ”Q.give();” som kaller metoden ”give” på porten ”Q”. Til slutt har vi litt av implementasjonen av klassen X. Metodens navn er ikke viktig, men den har to metodekall; ett på en port og ett med bruk av ”that”. Metoden ”ask” er tilgjengelig over connectoren K som er direkte koblet til parten x, mens metoden ”request” er tilgjengelig via porten P som har connectoren C tilkoblet.

Nå har vi sett et par eksempler på bruk av runtime-APIet, så nå kan vi beskrive det litt mer generelt. Metodene som er tilknyttet klassens porter og connectorer er:

- add<ConnectorName>Link(x, y) : Denne metoden lager en kobling mellom objektene x og y over den navngitte connectoren.

- remove<ConnectorName>Link(x, y) : Denne metoden fjerner koblingen mellom objektene x og y over den navngitte connectoren.
- that.<MethodName>( ) : Dette er et metodekall over connectorer som er direkte koblet til parten dette objektet er en del av. Dette er tenkt kode i implementasjonen av en klasse med objekter som opptrer i composite-strukturer.
- <PortName>.<methodName>() : Dette er et metodekall over connectorer som er tilkoblet den navngitte porten på den parten dette objektet er en del av. Dette er tenkt kode i implementasjonen av en klasse med objekter som opptrer i composite-strukturer.

Her er en oversikt over det generelle APIet som er tilgjengelig inne i en klasse med parter:

- add<PartName>Object(o) : Denne metoden brukes for å legge til objekt til en part. <PartName> byttes ut med med navnet på parten.
- get<PartName>Objects() : Denne metoden returnerer en liste med objektene denne parten består av.
- remove<PartName>Object(o) : Denne metoden fjerner objektet o fra parten.

4.6 Modell til Java script

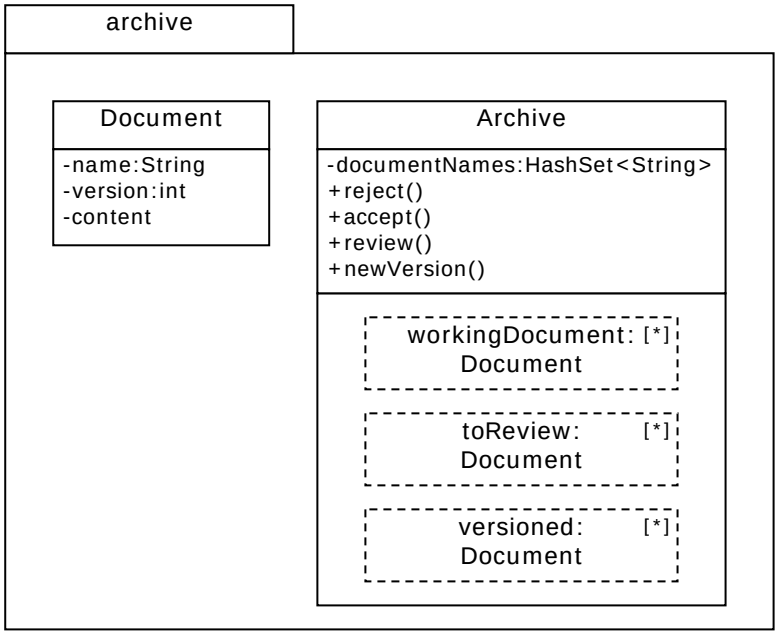
Scriptet som er laget i forbindelse med denne oppgaven er et ”Modell til Java” script. Dette scriptet traverserer en brukermodell, som er i henhold til metamodellen, og produserer Java-kode. Denne koden kan vi si består av to deler. Den første er initierings-koden som kjøres før programmet starter. Denne koden oppretter en brukermodell i Java runtime. Denne modellen blir brukt til typesjekking og til å finne ut av tilbud og bruk av operasjoner, interfacer, porter og portfacer.

Den andre delen er selve programmet som blir bygget av elementer fra rammeverket. Akkurat som Java-programmer blir bygget av medlemmer, metoder, interfacer og klasser, blir disse programmene i tillegg bygget opp av parter, portApplicationer, connectorer, associationer, portfacer og porter.

5.0 Part og Portface

I kapittel 4.0 har vi gjennomgått hvordan modelleringsspråket er og hvordan det tolker Part og Portface, mens her skal vi se hvordan modelleringsspråket kan brukes til modellering av eksemplene. Vi vil se hvordan vi egentlig bruker Port, Portface og Part for å realisere eksemplene i modelleringsspråket.

Eksemplet Archive er en modell av et dokumentarkiv med dokumenter i forskjellige stadier. Noen er i utarbeidingsstadiet, noen er til gjennomsyn og godkjenning og andre igjen er godkjente dokumenter. Når et godkjent dokument skal endres blir dokumentet duplisert. Det nye dokumentet har samme navn, men har et høyere versjonsnummer enn det opprinnelige. Deretter blir den nye versjonen lagt inn i parten for arbeidsdokumenter. Partene er en representasjon av stadiet dokumentene er i, og derfor vil dokumentobjektene flyttes mellom partene ettersom dokumentene endrer stadier. Til denne bruken er det naturlig å bruke reference parter som vist i Figur 5.1.



Figur 5.1 Klassediagram over pakken 'archive'

Klassen Archive har en part for hver tilstand dokumentet kan være i, og har metoder som passer på at de er ordnet riktig. Objekter av Document klassen er dataobjekter som inneholder informasjon og er utstyrt med vanlige get-set metoder.

Partene er av reference type og blir brukt slik at Document objektene dukker opp i parten workingDocument og ender i parten versioned. I mellomtiden er de innom parten toReview en eller flere ganger.

Implementasjonen er vist i Figur 5.2 og viser at oppførselen til Archive i hovedsak er å sjekke hvor et dokument er og å flytte et dokument fra en part til en annen. Koden som bruker partene minner mye om bruk av container-klasser i Java. En av de få forskjellene er at parter kan ha en

øvre og en nedre grense for hvor mange det kan være der. Disse må ikke brytes og derfor bør vi ha en swap funksjonalitet som muliggjør å bytte ut ett objekt med et annet. Implementasjonene er mer en skisse enn en virkelig implementasjon, og det er ikke viktig hvilken type innholdet i Document har eller hvordan dette endres. Det viktige her er hvordan dokumenter kan flyttes fra en part til en annen.

```
Class Archive {
   public Boolean add( Document doc ) {
      if ( documentNames.contains( doc.name ) {
         return false;
      }
      else {
         documentNames.add( doc.name );
         workingDocument.add( doc );
      }
   }
   public void accept( Document doc ) {
      if ( toReview.contain( doc ) ) {
         toReview.delete( doc );
         versioned.add( doc );
      }
   }
   public void reject( Document doc ) {
      if ( toReview.contain( doc ) ) {
         toReview.delete( doc );
         workingDocument.add( doc );
      }
   public void review( Document doc ) {
      if ( workingDocument.contain( doc ) ) {
         workingDocument.delete( doc );
         toReview.add( doc );
      }
   }
   public void newVersion( Document doc ) {
      if ( versioned.contain( doc ) ) {
         Document newDoc;
         newDoc = new Document( doc );
         workingDocument.add( newDoc );
      }
   }
}
```

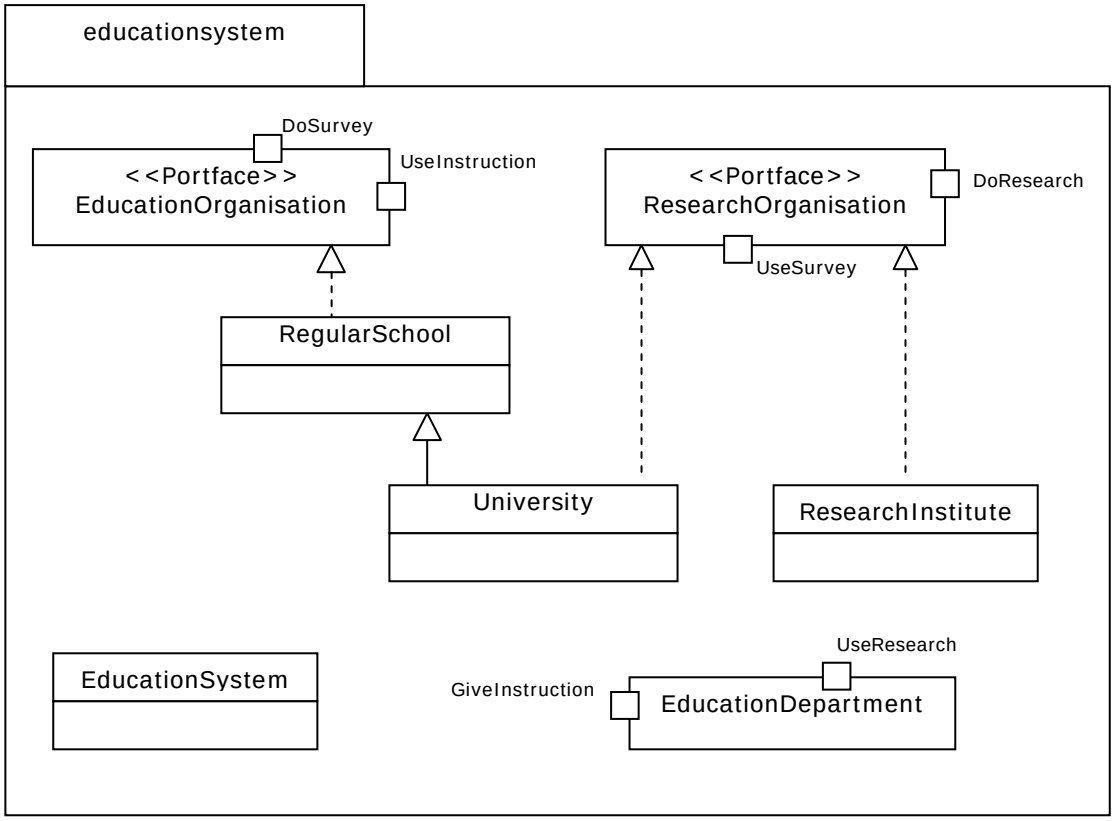
```
Class Document {
   Document( Document doc){
      name = doc.name;
      content = doc.content;
      version = doc.version +1;
   }
   Document( String inName ){
      name = inName;
      version = 1;
   }
}
```

Figur 5.2 Implementasjon av Document og Archive

I eksemplet om Utdanningsdepartementet, som er beskrevet i avsnitt 3.1, har vi tre forskjellige roller som spilles i composite-strukturen som er skissert i Figur 3.1. De tre rollene er Utdanningssted (EducationOrganisation), forskningsorganisasjon (ResearchOrganisation) og Utdanningsdepartementet (EducationDepartment). Disse tre rollene er beskrevet i Figur 5.3 som EducationOrganisation, ResearchOrganisation og EducationDepartment, hvor EducationOrganisation og ResearchOrganisation er portfacer.

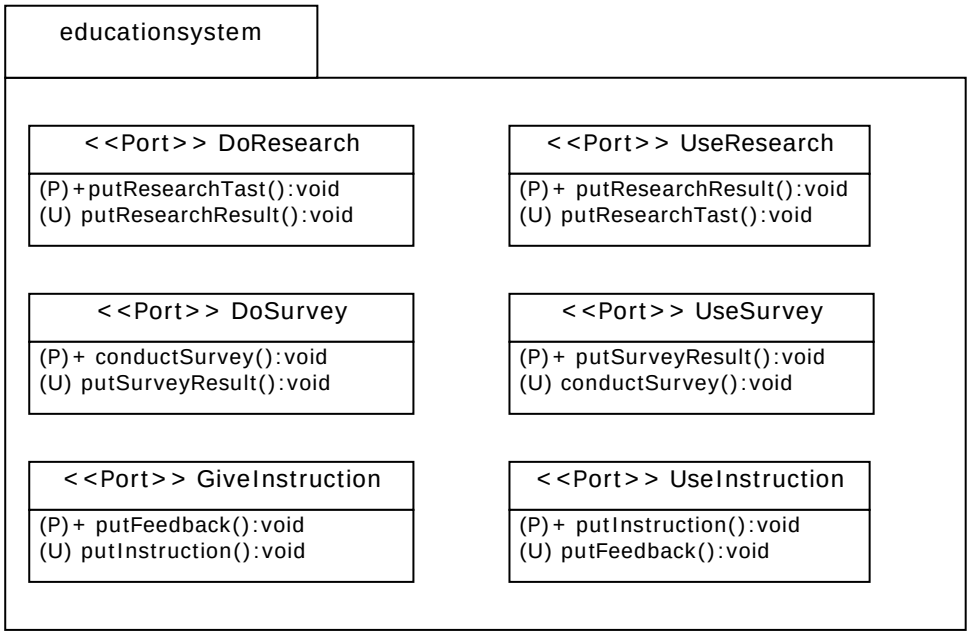
Portfacene EducationOrganisation og ResearchOrganisation blir implementert av klassene RegularSchool og University. Legg merke til at vi bruker ”realization” pilen i Figur 5.3 til å vise at en klasse implementerer et portface. Denne pilen blir vanligvis brukt for å fortelle at en klasse implementerer et interface. Dette resulterer i at klassen tilfredstiller enda en typebeskrivelse, og dette kan av noen anses for en form for svak spesialisering. Derfor er denne pilen så lik spesialiseringspilen. Et University objekt vil nå kunne inngå i både rollen som EducationOrganisation og som ResearchOrganisation.

Klassen RegularSchool realiserer eller implementerer portfacet EducationOrganisation. Klassen University spesialiserer RegularSchool og implementerer ResearchOrganisation.



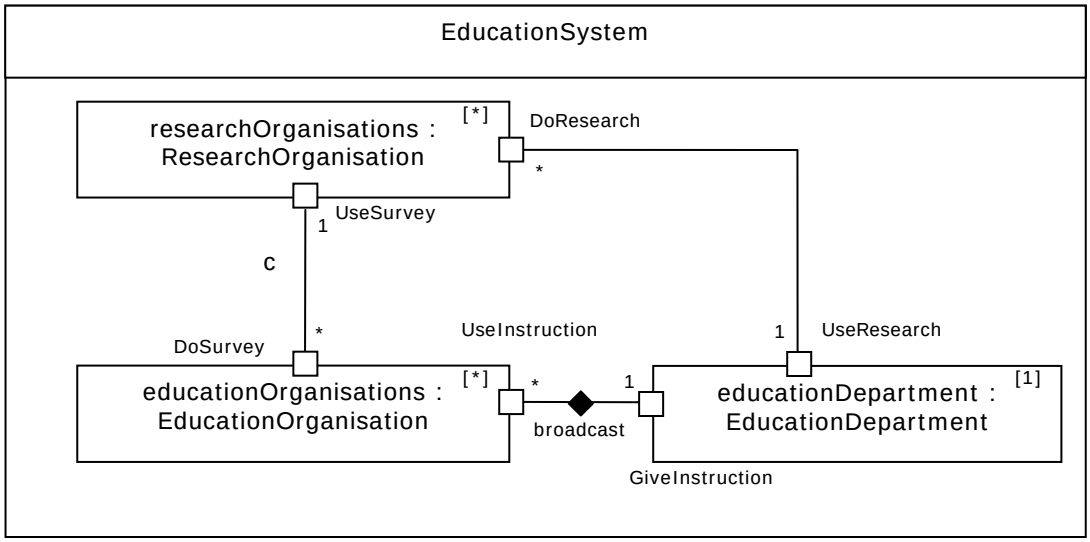
Figur 5.3 Klassediagram over pakken educationsystem

EducationSystem er klassen hvor hele systemet er beskrevet, og er beskrevet i detalj i Figur 5.5 som viser en composite-struktur over klassen.



Figur 5.4 Oversikt over port definisjonene

Figur 5.4 viser hvordan portene er definert. Disse portene er brukt i klassebeskrivelsene og vil bli aktivt brukt i composite-strukturen for å etablere en god beskrivelse av kommunikasjonen mellom partene. I UML bruker vi +, -, # og ~ for å fortelle synligheten til en operasjon. Det vil være naturlig å overføre dette til definisjonen av porter og portfacer. I tillegg til dette må vi vise om operasjonen er tilbudt eller brukt, så vi utvider den konkrete syntaksen til UML med å bruke (P) for provided og (U) for used for å synliggjøre forskjellen i diagrammene.



Figur 5.5 Composite-diagram over EducationSystem klassen

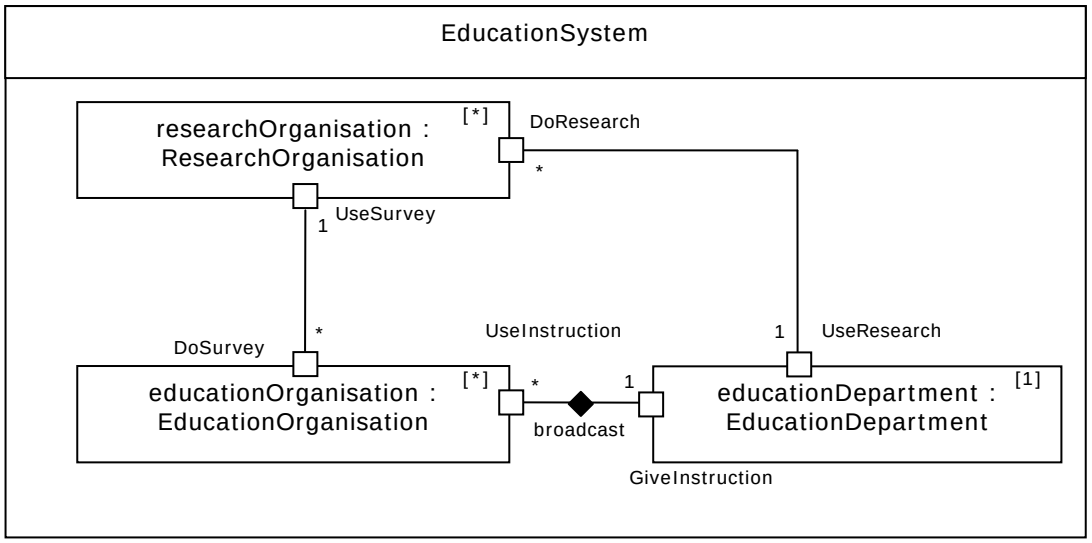
Figur 5.5 viser hvordan portface-typene brukes til å beskrive de forskjellige rollene og hvilke egenskaper som skal til for at et objekt passer inn i rollen. En Part er derfor typet med et Portface.

6.0 Association og Connector i samme kontekst

Nå skal vi se hvordan vi kan bruke dette modelleringsspråket slik at connectorer og associationer utfyller hverandre og brukes side om side.

Utdanningsdepartementet

I dette eksemplet har vi et Utdanningsdepartement som initierer forskning på undervisning for å kunne gi undervisningsstedene instrukser om hvordan de kan forbedre undervisningen sin. Forskningen går ut på å analysere empiriske undersøkelser på et utvalg av undervisningssteder.



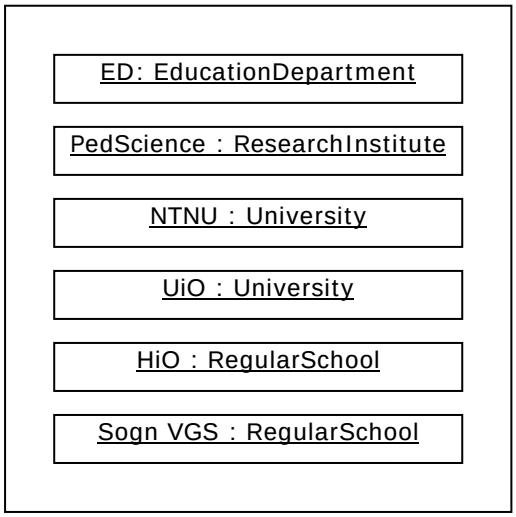
Figur 6.1 Composite-diagram over EducationSystem klassen

Vi ser at Figur 6.1 har tre parter med hver sin rolle. Connectorene beskriver sammen med portene hvilke parter som kommuniserer og hvordan de kommuniserer. Partene gir klar informasjon om hvilke operasjoner som blir kalt av hvilke parter.

```
class EducationDepartment {
    public void putResearchResult() {
        GiveInstruction.putInstructions();
    }
    public void startResearch() {
        UseResearch.putResearchTask(selectedObject);
    }
    public void putFeedback(){
        // storing feedback
    }
}

class RegularSchool implements EducationOrganisation {
    public void putInstructions(){
        UseInstruction.putFeedback();
    }
    public void conductSurvey() {
        DoSurvey.putSurveyResult();
    }
}

class ResearchInstitution implements ResearchOrganisation {
    public void putSurveyResult() {
        // storing the result
    }
    public void putResearchTask(){
        UseSurvey.conductSurvey();
    }
}

class University implements ResearchOrganisation extends RegularSchool {
    public void putSurveyResult() {
        // storing the result
    }
    public void putResearchTask(){
        UseSurvey.conductSurvey();
    }
}
```

Figur 6.2 Skisse over implementasjonen av klassene

Figur 6.2 viser en skisse over hvordan implementasjonen av javaklassene er i dette eksemplet. EducationDepartment kaller på putResearchTask på porten UseResearch og meldingen når den organisasjonen som har fått oppdraget. Denne connectoren støtter muligheten for at det kan være flere forskningsoppdrag. Vi ser at den utvalgte forskningsorganisasjonen i parten researchOrganisations også har tilgang på flere undervisningssteder i parten educationOrganisations. Men her brukes alle instansene som er koblet og ikke bare en. Det som avgjør forskjellen mellom denne connectoren og den andre er bruken. Med andre ord er det implementert en annen logikk i forskningsorganisasjonene enn det utdanningsdepartementet har. Så blir det sendt meldinger motsatt vei tilbake helt til utdanningsdepartementet. Når så utdanningsdepartementet skal instruere alle undervisningsstedene om endring av undervisningspraksis gjøres dette enkelt ved å kalle på putInstruction på porten GiveInstruction. Denne porten er koblet til porten useInstruction på parten educationOrganisation via en broadcast connector. Denne broadcast connectoren sørger for at en melding som blir gitt fra den ene siden blir distribuert til alle objektene i parten tilknyttet på den andre siden.



Figur 6.3 Liste over objekter som er med i partene.

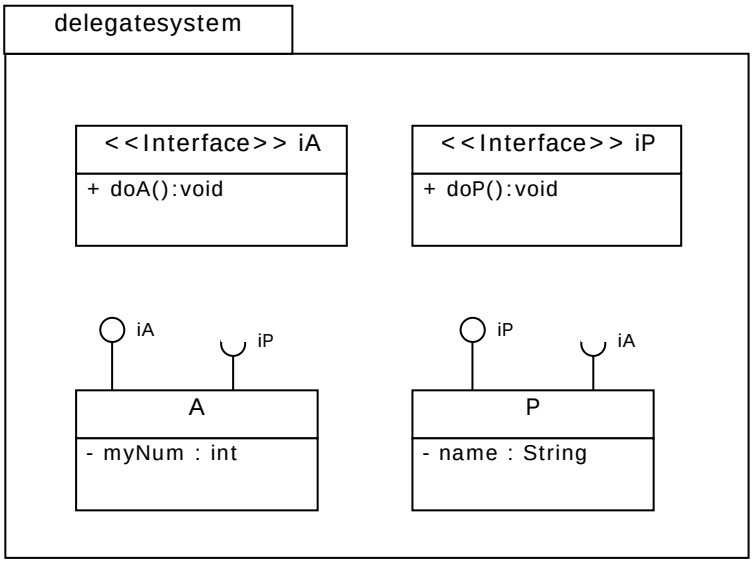
Figur 6.3 viser en liste over objekter som er med i dette eksemplet i en tenkt situasjon. Objektet ED er det eneste objektet som er i parten educationDepartment. Parten researchOrganisations har objektene PedScience, NTNU og UiO. Og til slutt så består parten EducationOrganisation av objektene NTNU, UiO, HiO og Sogn VGS.

Det er slik at UiO er utvalgt til å gjennomføre forskningen, og NTNU har reservert seg fra å delta i undervisningsundersøkelsene grunnet et annet prosjekt. Dette fører til at det skal gjennomføres en undervisningsundersøkelse på UiO, HiO og Sogn VGS.

En gjennomgang av kommunikasjonen kan klargjøre hva denne composite- strukturen betyr. ED sender forskningsoppdraget til UiO, og UiO gjennomfører så en undervisningsundersøkelse på UiO, HiO og Sogn VGS. Disse gir et resultat hver tilbake til UiO som analyserer resultatene og sender sitt forskningsresultat til ED. ED tar en beslutning basert på forskningsresultatet og sender så melding til alle undervisningsstedene i parten EducationOrganisations ved å bruke broadcast connectoren. Alle undervisningsstedene får instrukser og de rapporterer tilbake hvordan endringene gikk.

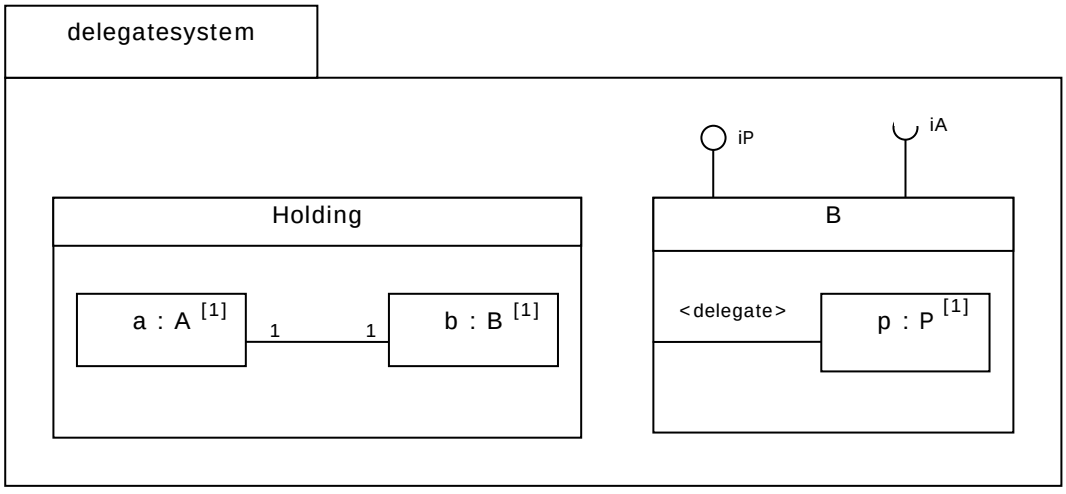
Delegate-connector

Det neste eksemplet vi skal gå igjennom som beskriver connectoren er ”Delegate Connector” og er beskrevet i avsnitt 2.2.2. Eksemplet går ut på at klasse B skal tilby interfacet iP ved å delegere implementasjonen av iP fra parten p. Denne parten skal inneholde et objekt som tilbyr dette interfacet. Denne modellen tester den grunnleggende funksjonaliteten rundt delegate connectors.



Figur 6.4 Klassediagram over delegatesystem pakken

Vi ser i Figur 6.4 at klassen P tilbyr interface iP og bruker interface iA og det samme gjør Klassen B. Det stikk motsatte gjelder for klasse A som tilbyr iA og bruker iP. Klassen Holding er nærmere beskrevet til venstre i Figur 6.5, hvor vi ser at partene a og b kan kommunisere over en connector.



Figur 6.5 Composite-diagrammer av klassene Holding og B

Det som gjenstår nå er å se hvordan B realiserer iP, og det er beskrevet til høyre i Figur 6.5, hvor vi ser at parten p sitt grensesnitt blir delegert via en delegate connector.

Når vi ser på figuren og ser hvordan delegate-connectorer blir brukt kan vi argumentere for at den verken trenger navn eller kardinalitet. Kardinaliteten må være 1 i begge ender for at det skal være entydig. Vi

har tolket en delegate-connector til å være uten funksjonalitet for routing av meldinger og slikt.

```
Class P {
    doP() {
        print( name );
        that.doA( 12, false );
    }
}

Class A {
    doA( inNum : int, callP: boolean ) {
        print( myNum, inNum, ( myNum + inNum ) );
        if ( callP ) {
            that.doP();
        }
    }
}
```

Figur 6.6 Implementasjonen av interfacene iP og iA

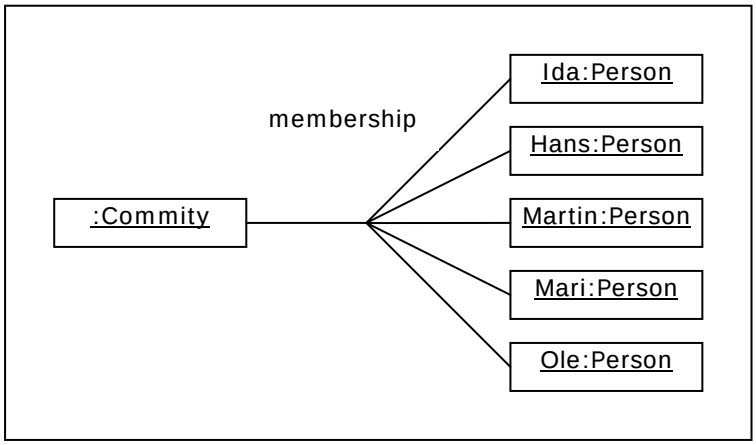
Figur 6.6 viser implementasjonen av interfacene iP og iA. Det er ikke så mye nyttig her annet enn at om vi gjør kallet doA(5, true) vil doA kalle doP som igjen vil gjøre kallet doA(12, false) som vil stoppe rekken av kall og returnere. Her får vi effektivt sjekket at kallene fungerer begge veier via delegator connectoren.

Dynamisk connector

En dynamisk connector er en type connector som på mange måter er tatt ut av sitt opprinnelige element. Det er en connector som knytter sammen objekter som ikke er en del av en part. På mange måter er det connectorens svar på en lokal variabel.

Vi kan forklare nytten av den og hva det er ved å bruke bussruter som et eksempel. Vanligvis vil en bussrute ha to endestasjoner, en rutetabell, ha regulerte takster og være en del av en turnusordning med bussjåfører. Vi kan tenke oss at vi har et arrangement på sommeren, en teltleir, der det skal fraktes folk fra teltleiren til et aktivitetsområde. Til dette formålet settes det opp en buss for å frakte alle menneskene. Denne bussruten skal være aktiv i en uke og trenger ikke den samme grad av organisering som Hønefoss-expressen, som skal gå hver dag til de samme tidene og være forutsigbar.

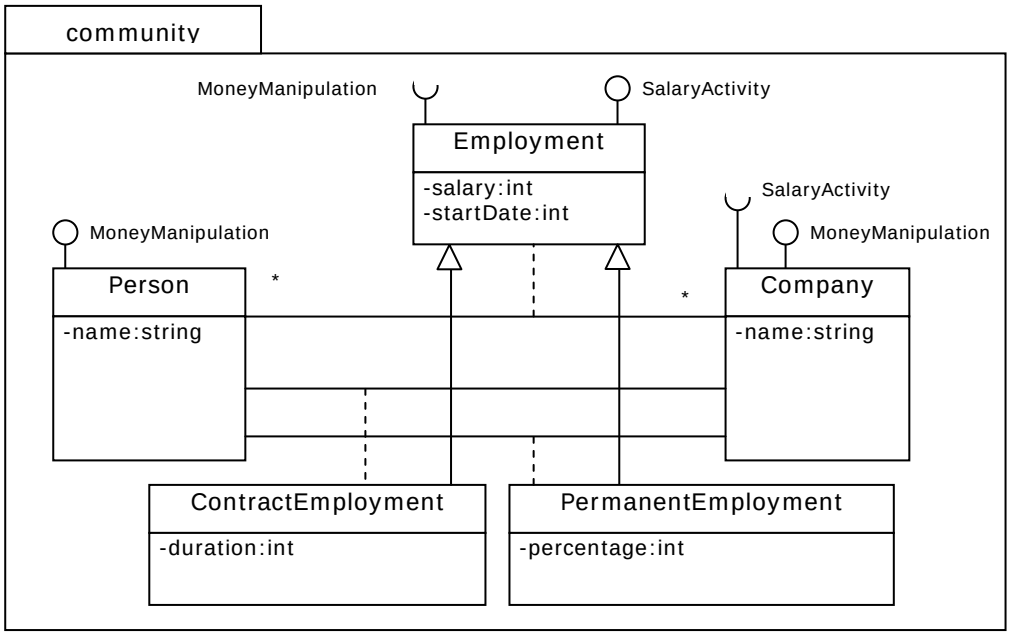
En dynamisk connector kan minne om en ad hoc løsning. Vi kan bruke en dynamisk connector til å beskrive en komité som opprettes for et bestemt formål. Komitéen skal ha fra 5 til 10 medlemmer som skal avgjøre et bestemt valg.



Figur 6.7 illustrasjon over en dynamisk conector

Community

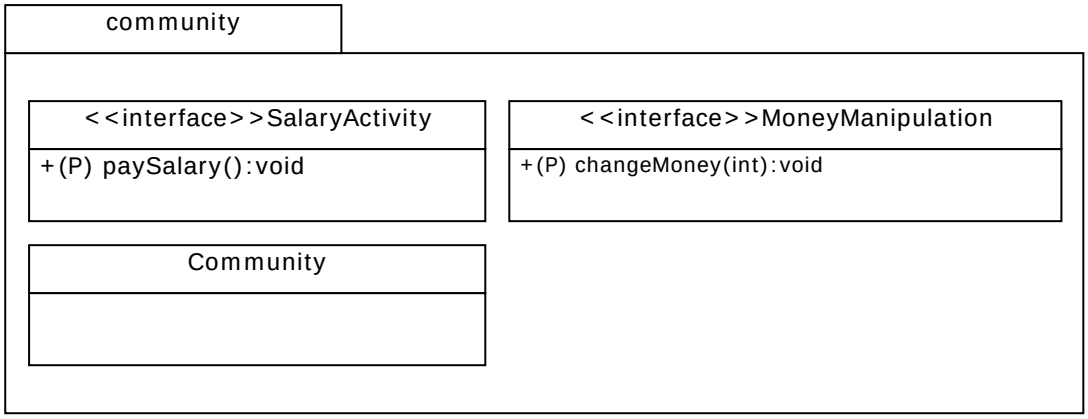
Det neste eksemplet vi skal gå igjennom her er community eksemplet hvor vi ønsker å bruke connectoren til å utfylle en association ved at den er en ”basedOn” connector. Eksempler beskrevet i kapittel 3.2 og er en modell over et lokalsamfunn med personer og bedrifter. Personene er ansatt i bedriftene og denne ansettelsen er modellert som associationen Employment. Company beskriver en generell bedrift og klassen Person beskriver en generell person.



Figur 6.8 Klassediagram 1 over community pakken

Figur 6.8 viser hvilke interfacer som er implementert av hvilke klasser. Hvis vi studerer litt nøyere ser vi at Company realiserer MoneyManipulation og bruker SalaryActivity, mens person realiserer bare

MoneyManipulation og ikke SalaryActivity, for det er Associationen Employment som realiserer SalaryActivity. For å forklare litt har vi vist et klassediagram til over resten av klassene i community pakken.



Figur 6.9 Klassediagram 2 over community pakken

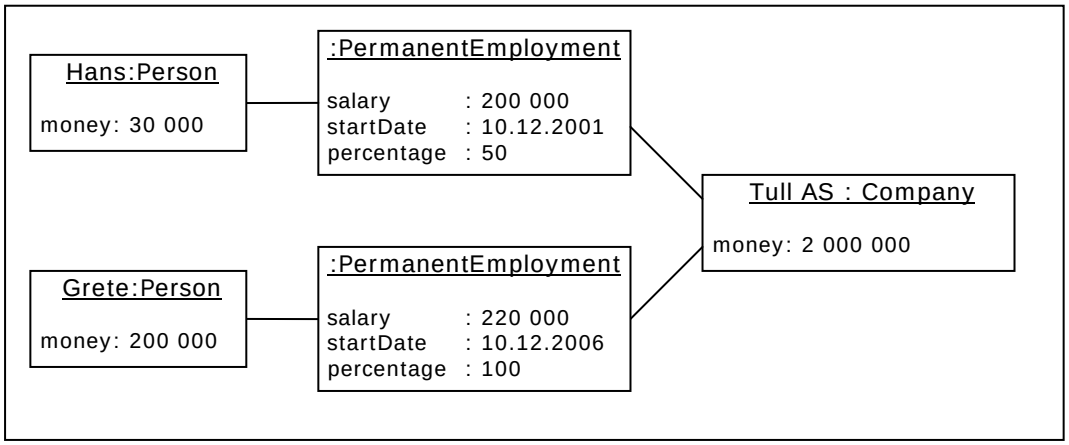
Figur 6.9 viser at interfacet SalaryActivity er det interfacet som tilbyr operasjonen paySalary og er ment å bli implementert slik at det er her selve pengetransaksjonen fra bedriften (company) til arbeidstakeren (Person) skal skje.

```
class PermanentEmployment implements SalaryActivity {
   ...
   private Person person; // role A
   private Company company; // role B

   public paySalary() {
      int actualSalary = salary * percentage ) / 100;
      company -= actualSalary;
      person   += actualSalary;
   }
}
```

Figur 6.10 Implementasjon av SalaryActivity i PermanentEmployment

Figur 6.11 Composite-diagram for Community klassen

Figur 6.11 viser composite strukturen til Community klassen som har to parter og en connector. Connectoren ”employment” er typet med Associationen Employment for å vise at den er basert på denne associationen.

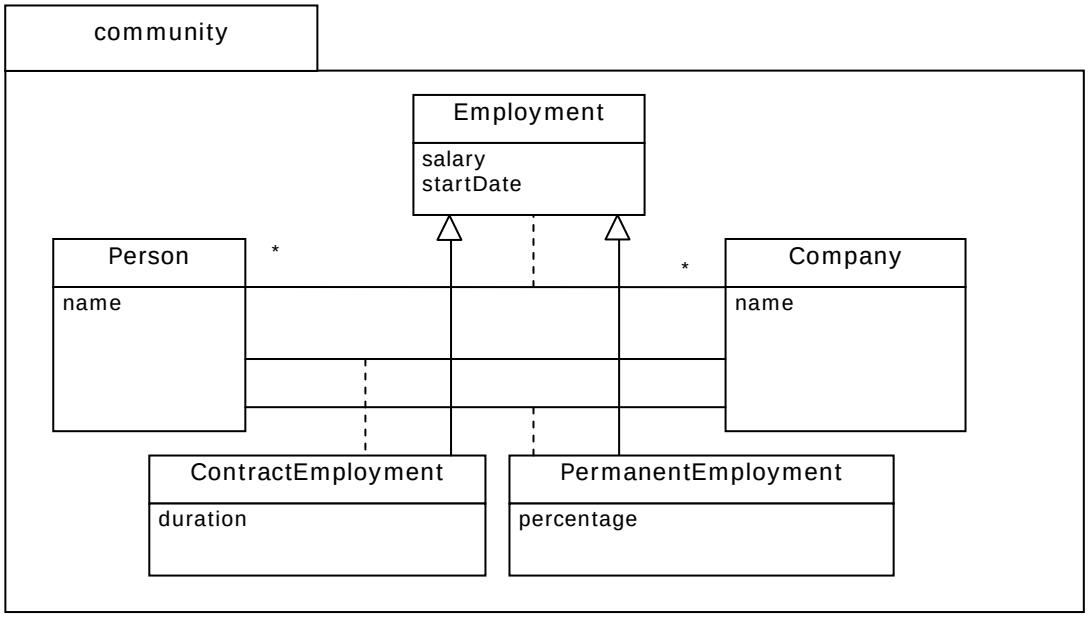


Figur 6.12 Tull AS her to ansatte med fast ansettelsesforhold

Vi kan tenke oss en situasjon som i Figur 6.12 har Tull AS har to ansatte og begge har fastansettelse. Vi kan tenke oss at Hans og Grete er en del av parten ”persons” i Community klassen (Figur 6.11), og at Tull AS er en del av ”companies” parten. Siden ”employment” connectoren er basert på Employment krever den en Employment link som er knyttet til koblingen i connectoren. Når nå objektet ”Tull AS” kaller operasjonen paySalary som er en del av interfacen SalaryActivity, vil connectoren godta kallet siden Employment realiserer dette interfacet og dermed tilbyr denne operasjonen. Denne operasjonen er implementert av PermanentEmployment som bruker informasjonen i linken for å gjennomføre transaksjonen fra bedriften til arbeidstakeren. Det kallet ender med at Hans får tildelt 50 % av 200 000 som er 100 000 og ender med å ha 130 000 i money attributten. Grete får tildelt 100% av 220 000 og ender med en money attributt på 420 000, mens Tull AS ender med å bli trukket for 320 000 og ender med 1 680 000 i money attributten.

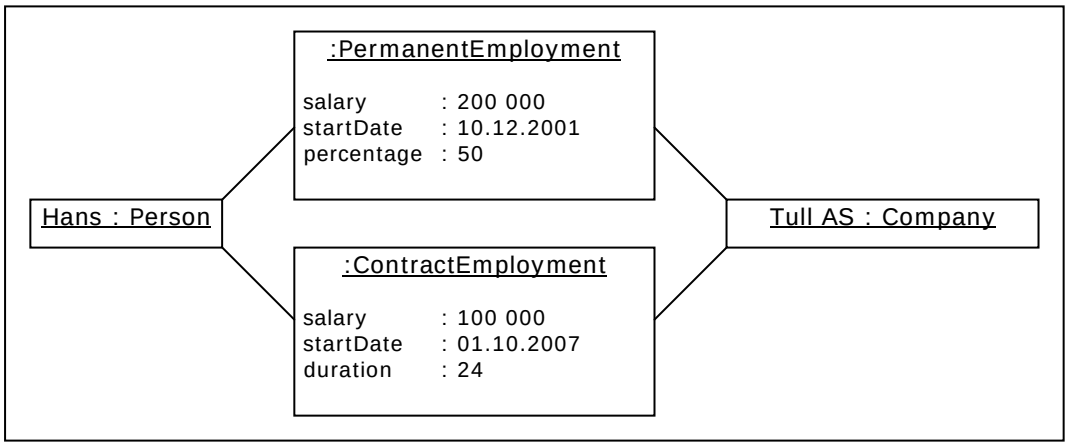
7.0 Spesialisering av association

I avsnitt 2.3, gjengir vi et eksempel fra Bierman og Wren som de bruker til å analysere problemer rundt spesialisering av associationer. Her kommer det et annet eksempel som avslører at de ønskede egenskapene som de ser etter i eksemplet beskrevet i Figur 2.21 og Figur 2.23 ikke dekkende i alle situasjoner.



Figur 7.1 Illustrasjon av modellen 'community'

Vi ser for oss en situasjon som beskrevet i Figur 7.1. Her har vi en association ansettelse som er mellom person og selskap. Denne ansettelsen har en attributt lønn. Om vi spesialiserer ansettelse med fastansettelse og kontrakt vil vi få en lignende situasjon som i eksemplet over.



Figur 7.2 Associationene mellom objektene Hans og Tull AS

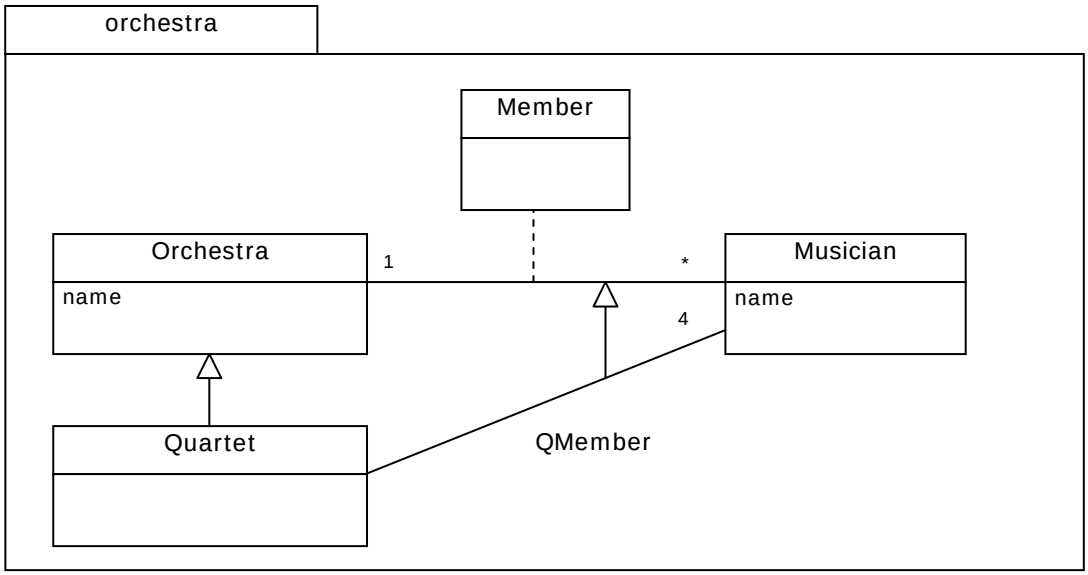
Som vist i Figur 7.2 kan en person, Hans, være fast ansatt i Tull AS i en 50% stilling samtidig som han er ansatt på kontrakt hos det samme selskapet. I dette tilfellet er det helt naturlig at attributten lønn skal dupliseres. Her ønsker vi denne egenskapen som er ankepunktet fra Bierman og Wren sin argumentasjon.

Det vi klart ser her er at modellene i Figur 2.21 og Figur 7.1 ser helt like ut med tanke på struktur, men i virkeligheten mener vi to forskjellige ting. Bierman og Wren ser etter en mulighet for å gi nye egenskaper til en eksisterende association mens i eksemplet illustrert i Figur 7.1 ser vi etter subtyping og spesialisering av en mer generell association.

Vi kan også legge merke til at innholdet i Figur 7.2 bryter med UMLs beskrivelse av associations-klasser. Figuren beskriver en situasjon hvor det finnes to instanser av associationen Employment mellom objektene Hans og Tull AS. UML sier i kapittelet om associationer og associations-klasser at associations-klassenes kardinalitet alltid er en. Eksemplet med ansettelsestyper viser at vi i noen tilfeller virkelig kan ha nytte av at en associations-klasse ikke er unik og at den tillater mer enn en instans mellom to gitte objekter.

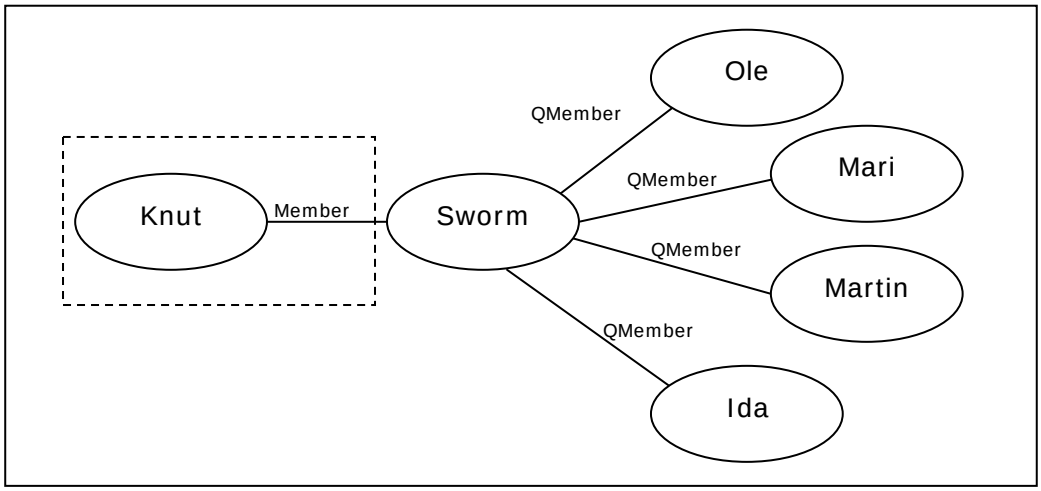
7.1 Når kan vi opprette en link

Akkurat som vi omdefinerer et objekts oppførsel ved å implementere metoder på nytt, vil det være naturlig at vi ved spesialisering av association beskriver en endring til hvilke roller et objekt inngår i.



Figur 7.3 Illustrasjon av modellen 'orchestra'.

Vi ser i figuren at associationen Member blir spesialisert til QMember og at QMember bare tillater fire medlemmer.



Figur 7.4 Knut som den femte musikeren i en kvartett.

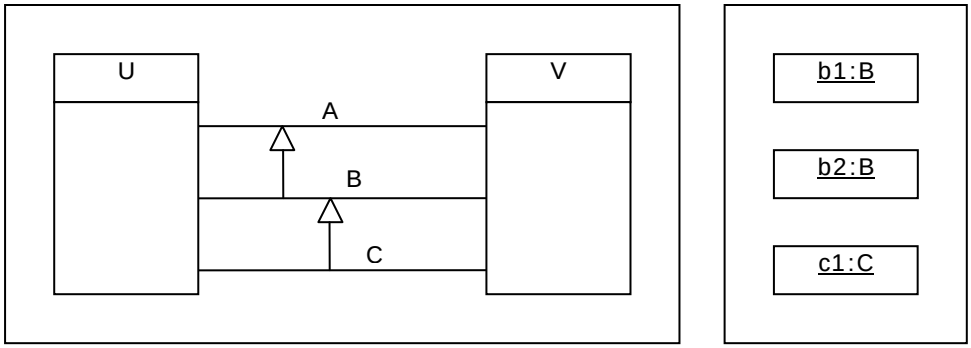
I Figur 7.4 har vi ett Quartet objekt, Sworm, og fire Musician objekter Ole, Mari, Ida og Martin. Videre har vi fire QMember linker, en mellom hver av Musician objectene og Sworm. Dermed har vi nådd QMembers øvre grense for hvor mange Musician objekter vi kan knytte til ett Quartet objekt. Hva om vi har muligheten til å bruke associationen Member, den har ikke nådd sin øvre grense. Så vi legger til Knut via Member. Da har plutselig Sworm 5 musikere selv om det er en kvartett.

Konklusjonen vår er at det er en egenskap ved Quartet objektene at Member ikke lenger er gyldig, og at QMember er den som skal instansieres og ikke Member. Årsaken til at vi redefinerer Member til QMember er at vi vet at et Quartet objekt forholder seg annerledes til objekter av Musician enn det et Orcestra objekt gjør. Om vi har et Quartet objekt så forholder det seg annerledes til et Musician objekt enn et Orhcestra objekt ville ha gjort. Det er altså selve Quartet objektet som setter denne begrensningen og det vil derfor være dette objektets ansvar å forhindre at instansieringen av Member(Sworm:Orchestra, Knut:Musician) blir tillatt. Sworm som i dette tilfellet er av typen:Quartet kjenner til både Member og QMember må kunne avvise instansieringen av Member. Det vil si at Associationen må ha bekreftelse fra objektet om intansieringen er lovlig. Vi snakker her om selve instansieringen og ikke bruken. Vi vil fortsette å bruke Member i logikk implementert i Orchestra selv om det er et Quartet objekt.

Vi ser allikevel at det kan finnes situasjoner der spesialisering ikke nødvendigvis betyr at ”tidligere” definerte associationer ikke er presise nok og derfor ikke gyldige. Dette vil kunne skje om flere ledd spesialisering skjer mellom de samme to klassene, slik som i eksemplet med ContractEmployment og PermanentEmployment. I slike tilfeller vil vi kunne få situasjoner hvor det kan være naturlig å kunne instansiere associationer av flere generasjoner samtidig.

7.2 Linker og spesialisering av association

Når vi tenker oss bare én association er det ganske enkelt, men hva om vi ser for oss tre associationer som i Figur 7.5, C spesialiserer B og B spesialiserer A.



Figur 7.5 T.v. spesialisering av association, t.h. linker av type B og C

En link av type C er også en link av typene B og A i form av subtyping.

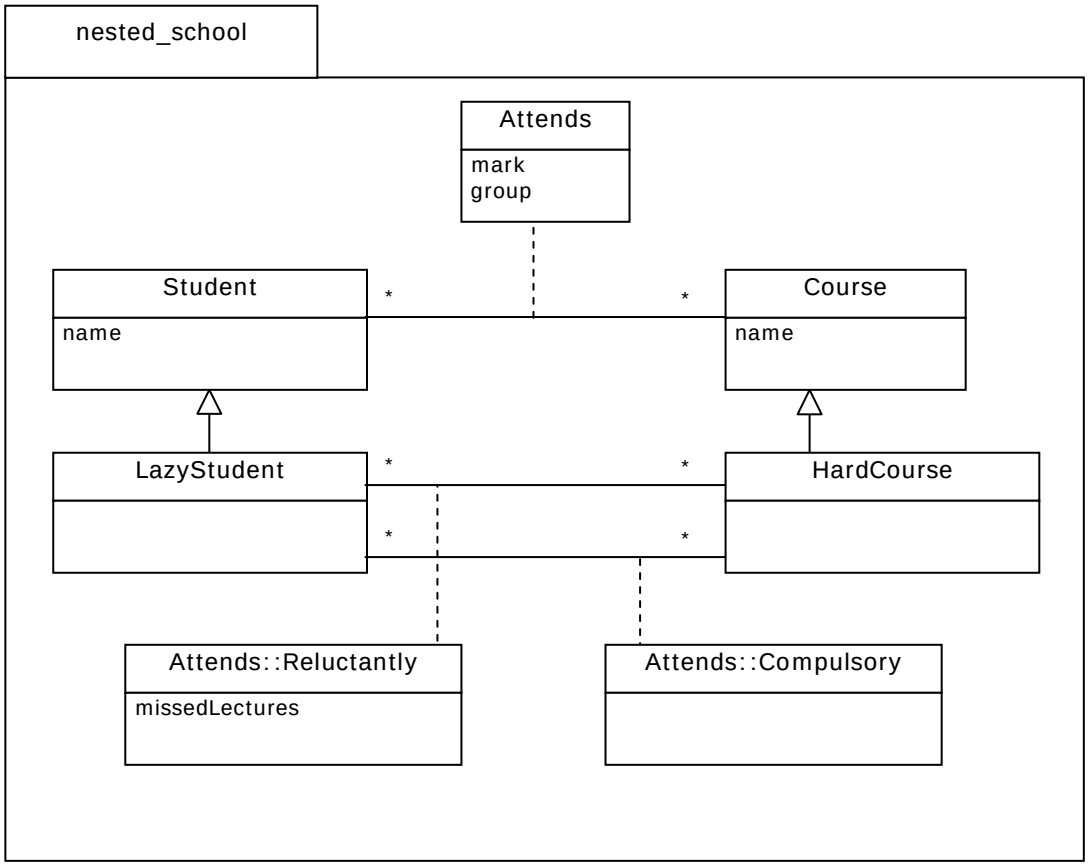
Vi har to linker av B og en av C. Vi spør om hvor mange A linker vi har. Vi kan få to svar på dette spørsmålet. Det første svaret er at det ikke finnes en eneste link som er av selve typen A, og det andre er at vi har tre linker som er av en eller annen subtype av A og derfor har de egenskapene vi søker i A.

Om vi velger det siste svaret som riktig vil dette føre til at associationen A akkumulerer linker fra både B og C, generelt vil dette også kunne medføre flere linker mellom to gitte objekter. Da er den bipartitte relasjonen ikke lenger enkel. Bierman og Wren mener den alltid må være enkel mens UML åpner for flere linker ved å si at linken ikke er unik.

7.3 Nestede associationer

Nestede associationer minner på mange måter om nestede klasser. Instansen av den indre associationen har en instans av den ytre som omgivelse, og har tilgang på dens attributter og operasjoner.

Bierman og Wren sin problemstilling kan minne om at de to spesialiseringene i utgangspunktet ikke er spesialiseringer. Det minner mer om et ønske om å kunne utdype eller gi tilleggsinformasjon til et allerede eksisterende attends objekt. Det virker ikke som det er egenskaper ved subtyping de er ute etter. Så i stedet for å spesialisere klassen attends kan det løses med å definere nye indre associationer til attends, og ved at instansene av CompulsoryAttends og ReluctantlyAttends er instanser av de indre associationene med et attends objekt som omgivelse.

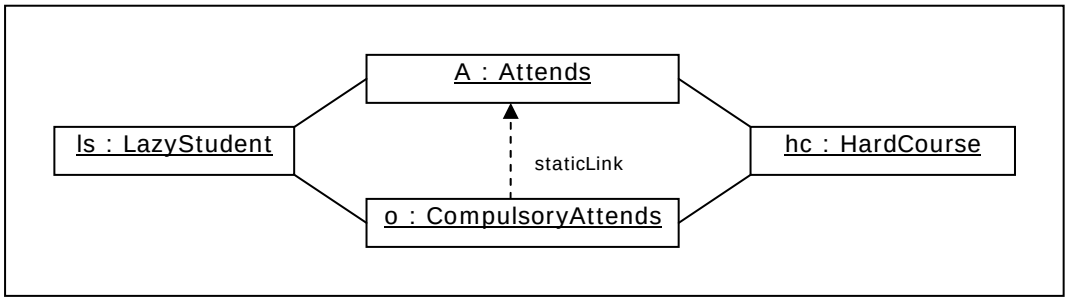


Figur 7.6 Forslag til løsning på Bierman og Wren sitt problem

Denne løsningen fører til at vi kan tilføre tilleggsinformasjon til et attends objekt, men vi lager ikke en mer detaljert beskrivelse av attends. Vi kan derfor ikke endre definisjoner som f.eks operasjoner. Den indre associationen må være tilknyttet samme klassene eller subtyper av de klassene som den ytre er tilknyttet.

Tankegangen ved instansiering av et objekt er også ganske annerledes. Ved spesialisering av Attends tar vi mål av oss å lage et fullverdig objekt av Attends med group og mark, men ved nesting vil vi regne med at Attends objektet allerede eksisterer. Derfor forventer vi ikke å måtte tilføre informasjon om group og mark.

Om vi skal opprette et Attends::Compulsory objekt o mellom to objekter ls og hc, er vi avhengige av at vi allerede har et Attends objekt a mellom de to objektene ls og hc, eller at vi kan opprette dette. En oppretting av dette Attends objektet a vil generelt ha den informasjonen dette objektet må ha for å bli opprettet gyldig. Med gyldig mener vi at Attends kan ha attributter som må tilordnes en verdi i det objektet blir opprettet. Siden o bare er en tilleggsinformasjon med a som omgivelse er det ikke naturlig at vi generelt vet nok til å opprette dette objektet på en gyldig måte, men om a ikke har noen krav til initialisering av attributtene vil vi kunne opprette o via en konstruktør uten parametere.



Figur 7.7 Objektet o har en Attends link som omgivelse

Multiplisitet på indre associationer vil fungere på samme måte som ved spesialisering av associationer. Vi kan nevne ett eksempel med Attends::Favourite som vil fortelle oss om hvilket fag denne studenten liker best. Selv om en student tar fire fag vil kun ett av de være studentens favoritt. Multiplisiteten til den indre associationen må kunne være inneholdt av den ytre associationen.

Når vi nå introduserer nestede associationer må vi nevne at de fungerer på samme måte som vanlige associationer når en connector er basert på den. Kriteriet for at en link av denne nestede typen kan opprettes er at det finnes en link av den ytre associationen mellom de to aktuelle objektene.

8.0 Konklusjon og videre arbeid

I dette prosjektet har vi fått en bedre forståelse av hva en association er og hvordan connector og association kan brukes for å utfylle hverandre i modellering.

Vi har funnet at UMLs tolkning av parter fører til unødvendige begrensninger i modellering. Vi mener at mengdene av objekter i parter ikke trenger å være disjunkte. Dette fører til at et objekt kan opptre i mer enn én part samtidig.

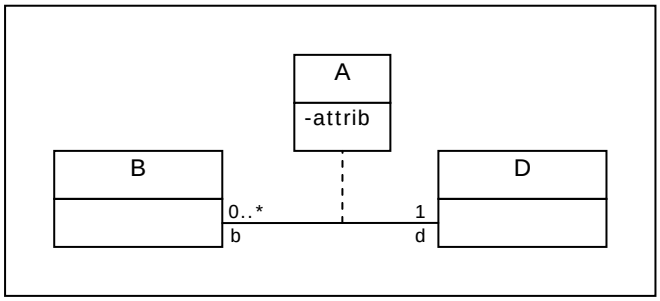
Vi har funnet det praktisk å forene beskrivelsen av klassens interaksjon på alle plan innenfor et begrep som vi har kalt Portface. Portface er en sammensmelting av en klasses interaksjon modellert både i klassediagrammer og i composite-strukturer. Dette fører til en samlet beskrivelse av hvordan en instans av denne klassen kommuniserer med omverdenen. Dette fører også til at vi klart får beskrevet hvilke krav den har til sine omgivelser.

Det viser seg at association kan spesialiseres på samme måte som for klasser, nemlig ved at den får en kopi av alle attributtene superassociationen har. Dette er beleilig med hensyn på hvor ren teorien blir i objektorienteringen.

En viktig problemstilling innen spesialisering av associationer er presentert av Bierman og Wren[3]. Med vår tolkning av spesialisering av association får vi flere kopier av attributtene. Det er dette Bierman og Wren bruker som ankepunkt mot denne tolkningen. Dette problemet løste seg imidlertid da vi fant ut at Bierman og Wren etterlyste egenskaper vi får ved nesting av associationer.

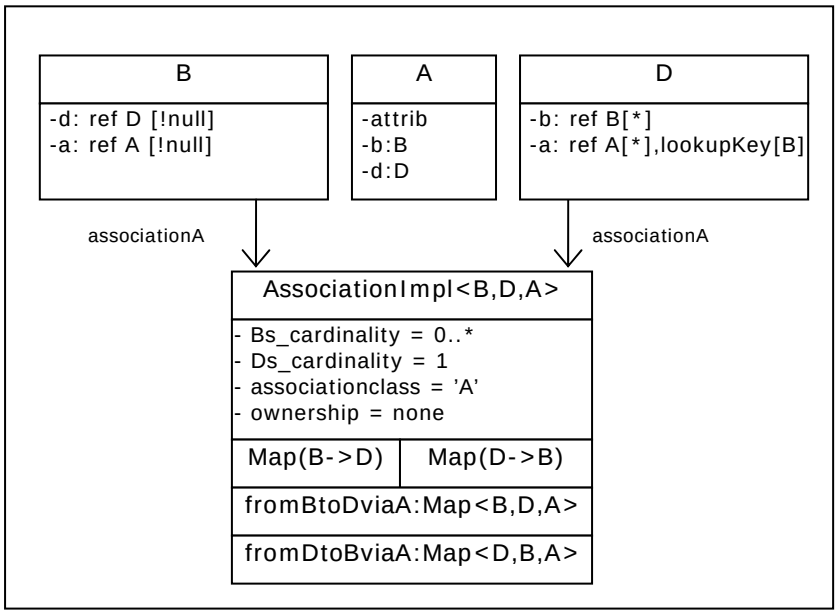
Nestede associationer minner på mange måter om nestede klasser. Instansen av den indre associationen har en instans av den ytre som omgivelse, og har tilgang på dens attributter som dermed ikke blir dupliserte. Nesting av associationer viser at associationer har mange likheter med klasser, samtidig som de gir unike muligheter som klasser ikke har. De er derfor av stor verdi og bør vurderes som en del av de grunnleggende byggesteinene som vi bygger våre programvaresystemer med.

Vi ser at våre resultater synliggjør områder som det nå er naturlig å studere nærmere. Angående associationer har vi i denne oppgaven konsentrert oss om selve associationen og hvilke egenskaper den har. Det er ikke slik at det bare er associationene som påvirkes av sine rolleklasser, associationene selv påvirker og setter krav til instansene av disse klassene. Vi har ikke fokusert på problemstillinger knyttet til at nedre og øvre multiplisitet må være innfridd for at et objekt av en klasse skal være i en lovlig tilstand. Vi har heller ikke tatt for oss problemstillinger rundt ”composition” associationer som fører til direkte eierskap. Objekter som eies må dø sammen med alle sine tilknyttede linker idet eieren opphører å eksistere. Runtime-plattformen er bygget på Java sin ”garbage collection”, og derfor vil alle objekter eksistere så lenge de er referert ett eller annet sted.



Figur 8.1 Beskrivelse av associationen A

Figur 8.1 viser en beskrivelse av associationen A, og vi skal se hvordan dette vil kunne se ut i en runtime-plattform.



Figur 8.2 Skisse over en realisering av associationen A

Figur 8.2 viser en skisse over hva en tenkt implementasjon av en association må være i en runtime-plattform som den vi har. Det eneste vi ikke har i vår implementasjon er attributtene i klasse B og D. Figuren viser at attributtene i klassen B har restriksjoner på at de ikke kan være "null". Det er slike restriksjoner vi i dag ikke tar hensyn til i vår implementasjon.

I metamodellen kan vi se at PortApplication via ConnectableElement er en spesialisering av NamedElement og eies av Part. Det gir ingen mening at klassenes implementasjon skal kunne bruke portApplicationens navn. Med vår implementasjon av runtime-plattformen blir portenes navn brukt for å bruke de metodene portene gir tilgang til via connectorene. Dette navnet må være avklart når klassen implementeres.

Når en klasse implementerer et portface, fører dette til at objektene av denne klassen kan inngå i de rollene som dette portfacet beskriver. Navnet til portApplicationen dukker først opp når vi lager en part i en bestemt composite-struktur med parter som types med dette portfacet. Implementasjonen kan ikke være avhengig av navnet på portApplicationen, men den må bruke navnet på selve porten. Når vi nå bruker portens navn i koden har vi ikke mulighet for å ha flere porter av samme type på samme part. For å få til dette må vi tenke at portApplicationene eies av klassen og ikke parten. Dette vil føre til at portApplication blir som en variabel med en port som type.

Dette er vurderinger som har kommet for seint inn i arbeidet til at vi kunne analysere dette videre og teste det ut med implementasjon.

9.0 Referanser

1: Thomas Meservy og Kurt D. Fenstermacher: ” Transforming Software Development: An MDA Road Map”, “IEEE Computer magazine” September 2005

2: James Rumbaugh: ”Relations as Semantic Constructs in an Object-Oriented Language”, OOPSLA ’87 Proceedings, October 4.8,1987, (ACM 0-89791-247-0/87/0010-0466)

3: Gavin Bierman og Alicdair Wren: “First-Class Relationship in an Object-Oriented Lanuage”, A.P.Blanck (Ed): ECOOP 2005, LNCS 3586, PP. 262-286, 2005.

4: Scott W. Ambler: “The Elements of UML 2.0 Style” ISBN-13: 978-0-521-61678-2

5: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, verson:formal/2007-11-02 (http://www.uml.org/)

6: Booch, Jacobson og Rumbaugh : ”Using UML Software Engineering With Objects and Components”, ISBN : 0-201-64860-1

7: Booch, Jacobson og Rumbaugh : ”The Object Constraint Language Second Edition”, ISBN : 0-321-17936-6

10.0 Tillegg A : Gjennomgang av metamodellen

10.1 Forklaring

kardinalitet: []

1 : Det må være en.
? : Enten en eller ingen.
* : null til mange.
+ : en til mange.

() : restriksjon, eks : (>0 || =-1) betyr større enn null eller lik –1.

10.2 Metamodell elementene

NamedElement

Spesialisering av Ingen
Abstrakt klasse

Beskrivelse

Dette er grunnelementet for alle elementer som er navngitte. Generelt er det elementets navn som blir brukt for å skjelne mellom elementer i modellen.

Attributes

name : String , Elementets navn

Package

Spesialisering av NamedElement

Beskrivelse

Dette er elementet på øverste nivå. Alle forekomster som ikke er av typen Package, er direkte eller indirekte inneholdt i en instans av Package. Dette er elementet som holder hele modellen sammen.

types er alle typedefinisjonene som pakken inneholder

Attributes

types[*] : AbstractType

AbstractType

Spesialisering av NamedElement
Abstrakt klasse

Beskrivelse

AbstractType er den grunnleggende typen i metamodellen, alle typer er en spesialisering av AbstractType, fra primitive typer til associationer. Samtidig så er det den felles betegnelsen på enkle typer, med en verdi.

super beskriver hvilken AbstractType denne er en spesialisering av.

Attributes

super : AbstractType [?]

PrimitiveType

Spesialisering av AbstractType

Beskrivelse

Dette er de primitive typene som all data til slutt er lagret i, som strenger, tall og boolske verdier.
En primitiv type kan ha komplekse egenskaper, men den skal være å betrakte som primitiver for modellen.

Parameter

Spesialisering av NamedElement

Beskrivelse

Parameteren er typet med en AbstractType, slik at alle typer kan bli sendt som en parameter. Typen må være definert om modellen skal være entydig.

type definerer parameterens type.

Associations

type : AbstractType ([?])

Member

Spesialisering av AbstractType

Beskrivelse

Beskrivelsen av et medlem. Som en attributt i en klasse.

type sier hvilken type medlemet har.
visability beskriver synligheten til elementet.
initialValue beskriver den verdien dette medlemmet skal gis som standard verdi når objektet opprettes.

Associations

type : AbstractType

Attributes

visability : String [“public”,”package”,”protected”,“private”]
initialValue : String [?]

Operation

Spesialisering av AbstractType

Beskrivelse

Beskrivelse av signaturen til operasjonen. Ofte omtalt i programmering som et metodehode.

type beskriver returtypen til operasjonen.
visability beskriver synligheten til elementet og hvilken type elementet har.

Associations

type:AbstractType

Aggregations

parameters : Parameter [*]

Attributes

visability:String [“public”,”package”,”protected”,“private”]

Interface

Spesialisering av AbstractType

Beskrivelse

En samling som definerer et sett med operasjoner. Et interface definerer ofte de operasjonene som kreves for å tilby en bestemt tjeneste, evne eller et sett med operasjoner som på et vis hører sammen.

Et interface kan kun arve fra ett interface, og med arv mener vi at alle operasjoner som er definert i super interfacet er definert i sub interfacet.

Aggregations

operations : Operation[*]

Portface

Spesialisering av AbstractType

Beskrivelse

Portface er en beskrivelse av en samlet oppførsel. Associationen ”uses” sier hvilke Interfacer den bruker og associationen ”provides” sier hvilke Interfacer den tilbyr. Aggregasjonen “ports” sier hvilke porter som tilbys i sammensetning med andre enheter.

operations sier hvilke operasjoner som tilbys av dette portfacet.

Spesialisering gjøres på samme måte som for interfaces. I tillegg kommer akkumulerte porter. Et portface kan spesialisere både Interface og Portface.

Associations

uses : Interface [*]
provides : Interface [*]

Aggregations

ports : Port [*]
operations: Operation[*]

Port

Spesialisering av NamedElement

Beskrivelse

Port er en beskrivelse av en bestemt oppførsel.

Uses og provides sier hvilke interfaces som tilbys og brukes av porten.

usesOperations sier hvilke operasjoner porten bruker og operations forteller hvilke operasjoner som tilbys av denne porten.

Associations

uses : Interface [*]
provides : Interface [*]
usesOperations : Operation [*]

Aggregations

operations : Operation [*]

Connector

Spesialisering av NamedElement

Beskrivelse

Beskriver en kommunikasjonskobling mellom to ConnectableElements. Connectoren definerer kardinalitet på endene. isDelegate sier om koblingen er mellom elementer på samme nivå (usann) eller om det er en kobling mellom et eier element og et eiet element. Associationen ”basedOn” sier om Connectoren er en selvstendig kobling eller om den er en lokal kobling mellom to instanser av klasser som har en assosciation mellom seg som beskriver den samme koblingen.

Om isBroadcast er sann knytter connectoren sammen to mengder istedet for å koble objekter som en relasjon. Om det er en broadcast connector vil en melding bli sendt til alle på den motstående siden.

Associations

endA : ConnectableElement [1]
endB : ConnectableElement [1]
basedOn : Association [?]

Attributes

lowerBoundEndA : Integer (>-2)
upperBoundEndA : Integer (>0 || =-1)
lowerBoundEndB : Integer (>-2)
upperBoundEndB : Integer (>0 || =-1)
isDelegate : Boolean
isBroadcast : Boolean

ConnectableElement

Spesialisering av NamedElement
Abstrakt klasse

Beskrivelse

Et generelt element som har en bestemt oppførsel, og kan kobles til connectorenes ender.

connectors er en mengde med referanser til alle connectorene dette elementet er direkte tilknyttet.

part sier hvilken part som eier dette elementet, og om det er en part eier den seg selv.

Associations

connectors : Connector[*]
part : Part

PortApplication

Spesialisering av ConnectableElement

Beskrivelse

Dette er en ”bruk” av en port. type beskriver hvilken port som er i bruk

Associations

type : Portface [1]

Part

Spesialisering av ConnectableElement

Beskrivelse

Dette er en beskrivelse av en del av en klasses indre oppbygning. Disse delene kommuniserer via connectorer og har kardinalitet for å kunne holde mange objekter av en bestemt type eller subtyper av denne.

portApplications er samlingen av de porttypene som tilbys av denne parten.

type beskriver hvilken type objektene skal være av eller være en subtype av.

Associations

type : Portface

Aggregations

portAppliations : PortApplication[*]

Attributes

lowerBound : Integer (>0)
upperBound : Integer (>0 || =-1)

Method

Spesialisering av NamedElement

Beskrivelse

Method er en bestemt implementasjon av operasjonen som associationen header viser til.

Associations

header : Operation

Attributes

body : String

Class

Spesialisering av Portface

Beskrivelse

Class er en beskrivelse av en klasse med de egenskapene en klasse har i denne metamodellen.

Klassen har medlemmer (members) som kan være av alle typer og har unike navn.

parts er en samling av de delene klassen blir bygget av som er implementert av andre klasser.

connectors er samlingen av de connectorene som kobler sammen delene klassen består av.

Metodene (methods) er konkrete implementasjoner av de operasjonene som klassen implementerer selv, mens de operasjonene som klassen ikke implementerer og allikevel skal tilby må delegeres fra en del som implementerer den. Dette skjer via en delegate connector.

Associationen “associations” er en liste med associationer som klassen har en rolle i.

uses og provides forteller hvilke interfaces som brukes og tilbys.

connectors er de connectorene som klassen eier, og kobler sammen parter eid av klassen.

normalConstructor er Java-kode som skal kjøres i den vanlige konstruktøren til klassen.

En klasse kan spesialisere Portface og Class.

Spesialisering betyr at klassen kan alle de ting som super klassen kan og at det som er definert i klassen kommer i tillegg. Om en metode blir omdefinert betyr det at den siste definisjonen vil være gjeldende.

Aggregations

connectors : Connector [*]
parts : Part [*]
members : Member [*]
methods : Method [*]

Associations

associations : Association
uses : Interface [*]
provides : Interface [*]

Attributes

normalConstructor : String

Association

Spesialisering av Class

Beskrivelse

En association er en kobling mellom to klasser som spiller roller for hverandre.

En association kan spesialisere Class og Association.

Spesialisering betyr at subassociationen må ha restriksjoner og krav som er delmengder av superassociationen. Alle subassociationer må være lovlige innenfor sin superassociation.

nestedAssociations er de nestede associationene til denne associationen.

Associations

roleA : Class [1]
roleB : Class [1]

Aggregations

nestedAssociations : Association [*]

Attributes

lowerBoundEndA : Integer (>-2)
upperBoundEndA : Integer (>0 || =-1)
lowerBoundEndB : Integer (>-2)
upperBoundEndB : Integer (>0 || =-1)
roleAName : String
roleBName : String

Figurliste