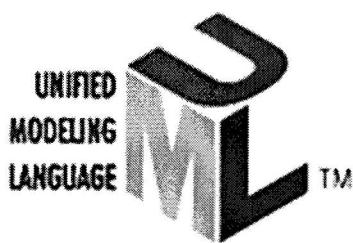


UML-Light

**note til
programmeringsundervisningen
samt semesterprojekterne
på de første semestre på
Ingeniørhøjskolen**



Versionshistorie

Versionsnr.	Dato	Initialer	Versionen omfatter
1.0	14.02.2003	FOH	Første version.
1.01	16.02.2003	FOH	Et par korrekturrettelser samt tilføjelse af <i>extern</i> erklæring i C eksemplerne.
2.0	20.1.2004	FOH	Eksempel 2 er flyttet til UML-Light++ afsnittet og har fået tilføjet et eksempel på hhv. et aktør-kontekstdiagram og et Use Case diagram. I Use Case afsnittet er der tilføjet et eksempel på en minimal Use Case skabelon.
2.1	19.06.2013	FOH	Div. mindre typografiske opdateringer, tekst om SysML og opdatering vedr. anvendelsen i studiet, case værktøjer og nyt logo.

Indholdsfortegnelse

1	Introduktion.....	3
1.1	Formål	3
1.2	Baggrund.....	3
1.3	UML-Light.....	3
1.4	Modulspecifikation kontra modulimplementering.....	4
2	Basale OO begreber	6
2.1	Indkapsling (<i>Information Hiding</i>).....	6
2.2	Klasser og objekter.....	6
2.3	Sammenhæng mellem klasse og implementering.....	8
3	UML-Light diagrammer	10
3.1	UML-Light note	10
3.2	UML-Light klassediagram	10
3.3	UML-Light sekvensdiagram	12
3.4	UML-Light tilstandsdiagram	13
3.5	UML-Light aktivitetsdiagram	14
3.6	Sammenhæng mellem UML-Light diagrammer	16
3.6.1	Sammenhæng mellem klassediagram og sekvensdiagram	16
3.6.2	Sammenhæng mellem klassediagram og tilstandsdiagram	16
3.6.3	Sammenhæng mellem klassediagram og aktivitetsdiagram	16
3.7	Diagrammering af Hardwareenheder	17
3.8	UML-Light oversigtsdiagram	19
4	Sammenhæng mellem klassediagram og kode	20
4.1	Klassediagram og <i>include</i> -filer	20
4.2	Implementering af associationer	21
5	Eksempel 1: Minutur designet med UML-Light.....	24
5.1	Beskrivelse af eksemplet.....	24
5.2	UML-Light model.....	24
5.3	Minutur implementeret i C++	28
5.4	Minutur implementeret i C.....	30
5.4.1	C implementering af maksimum ét objekt pr. klasse	30
5.4.2	C implementering af flere objekter pr. klasse	33
6	Værktøjer til UML-Light, UML-Light++ og UML.....	36
7	UML-Light++ til objekt-orienteret udvikling	37
7.1	UML-Light++ og kravspecifikation vha. Use Cases	37
7.1.1	Basal Use Case notation	37
7.1.2	Udvidet Use Case notation	41
7.2	UML-Light++ klassediagrammer med generalisering/specialisering	43
7.3	UML-Light++ klassediagrammer med udvidede associationsbegreber	44
8	Eksempel 2: Logikanalysator designet med UML-Light++	46
8.1	Beskrivelse af eksemplet.....	46
8.2	UML-Light++ model	47
9	Referencer	53
	Appendix: Anvendelse af UML-Light, UML-Light++ og UML på Ingeniørhøjskolen Aarhus Universitet.....	54

1 Introduktion

1.1 Formål

Formålet med denne note er at introducere en beskrivelsesnotation, der kan anvendes i forbindelse med objektbaseret og objektorienteret softwareudvikling.

Notationen kaldes her ***UML-Light***, da det er en udvalgt delmængde af UML, der er specielt tilpasset til programmeringsundervisningen på de første semestre på diplomingeniøruddannelsen. Samtidig kan UML-Light også anvendes som et design- og struktureringsredskab ved gennemførelse af semesterprojekterne på første til tredje semester. I kapitel 7. introduceres ***UML-Light++***, som en mindre udvidelse af UML-Light, der kan anvendes i forbindelse med programmeringsundervisningen og semesterprojektet på andet semester på E og IKT-retningen.

Anvendelsen af UML-Light, UML-Light++ og UML på Ingeniørhøjskolen Aarhus Universitet, beskrives i Appendix.

1.2 Baggrund

UML står for *Unified Modeling Language* [UML2011], der er navnet på en OMG standard for objektorienteret udvikling. OMG (*Object Management Group*) [OMG] er en sammenslutning af ca. 800 virksomheder. UML anvendes i dag verden over som beskrivelsesværktøj i forbindelse med SW udviklingsprojekter. UML understøttes af et stort antal værktøjer, der spænder lige fra tegneværktøjer til de mere avancerede CASE værktøjer (*Computer Aided Software Engineering*). I kapitel 6 gives der en kort introduktion til forskellige værktøjer.

Indenfor objektorienteret udvikling skelner man mellem *objektbaseret udvikling* og *objektorienteret udvikling*.

Objektbaseret udvikling er den delmængde af objektorienteret udvikling, der også kan implementeres vha. et ikke objektorienteret programmeringssprog som f.eks. C og assembler. Objektbaseret udvikling kan naturligvis også implementeres i objektorienterede programmeringssprog som f.eks. C++, Java og C#. Principperne i objektbaseret udvikling beskrives i kapitel 2. Basale OO begreber.

Objektorienteret udvikling tilføjer begreber som generalisering/specialisering (nedarvning) og polymorfi. Disse begreber indlæres i programmeringsundervisningen på andet semester og kræver, at man anvender et objektorienteret programmeringssprog (f.eks. C++, Java, C#) ved implementeringen.

1.3 UML-Light

Det er vigtigt at gøre sig klart, at UML er en standard, der beskriver en notation, der er baseret på et sæt af objektorienterede begreber og koncepter. UML beskriver således **ikke** en metode eller en udviklingsproces for, hvordan man udvikler et produkt eller et system, hvori der indgår software.

UML kan betragtes som en værktøjskasse – og UML-Light som en mindre værktøjskasse, der kun indeholder de mest basale modelleringsredskaber (UML's "hammer", "svensknøgle", "skævbider" og "skruetrækker").

UML-Light værktøjerne anvendes konstruktivt ved analyse- og design af et ønsket system og som dokumentationsværktøj til at dokumentere systemet med.

De tre vigtigste UML-Light diagrammer er:

- klassediagrammer (*Class Diagrams*)
- sekvensdiagrammer (*Sequence Diagrams*)
- tilstandsdiagrammer (*State Diagrams*)

Disse tre typer af diagrammer anvendes gennem hele udviklingsforløbet.

I UML-Light versionen af UML medtages og beskrives kun de mest basale notationer på disse tre typer af diagrammer.

Ved detaljeret design af en funktions eller procedures virkemåde kan man enten anvende pseudokode eller hvis man ønsker at diagrammere denne, så kan man anvende UML's aktivitetsdiagrammer. UML-Light inkluderer derfor også aktivitetsdiagrammer (*Activity Diagrams*) i deres mest basale udgave.

Da teknisk orienterede SW projekter ofte også omfatter hardwareudvikling, vil det være nyttigt, også at kunne designe og diagrammere hardwaren i projektet. Til dette brug anvendes UML's *Deployment* diagrammer, hvorfor disse også er inkluderet i UML-Light i deres mest basale udgave.

Efter definering af SysML standarden [SysML2012], der er en overbygning på UML, anbefales det fremover at anvende SysML til den overordnede dokumentation af systemer, der består af både hardware, mekanik og software. UML anvendes således udelukkende til design og dokumentation af softwaren.

I denne note vil der, hvor der naturligt findes en dansk oversættelse af et begreb, blive anvendt danske betegnelser. Ellers anvendes den engelske terminologi, da dette fremover vil lette læsningen af litteratur om UML og objektorienteret udvikling.

1.4 Modulspecifikation kontra modulimplementering

Når man skal udvikle og dokumentere et softwaremodul, et hardwaremodul eller en hardwarekomponent, er det meget nyttigt at beskrive modulets specifikation dvs. modulets grænseflade og funktionalitet adskilt fra beskrivelsen af modulets konkrete implementering.

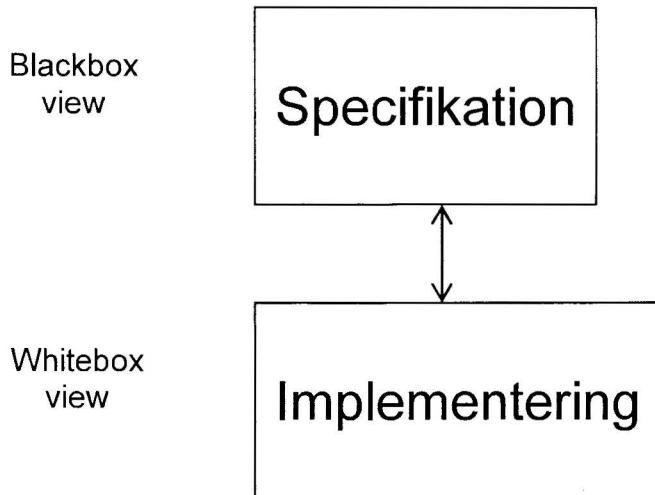
Modulets specifikation (grænseflade og funktionalitet) kaldes for et *blackbox view* for at symbolisere, at man ikke kan se, hvordan modulet internt er implementeret. Modulets implementering kaldes tilsvarende for et *whitebox view*, da man her kan se ind i modulet.

Ved design og strukturering af et system opdeles systemet i et antal moduler. For hvert modul beskrives modulets grænseflade- og funktionsbeskrivelse i en **modulspecifikation**.

Modulspecifikationen tjener to formål, dels beskriver den hvorledes andre anvender modulet (en brugsanvisning) og dels er det specifikationen for den, der skal udvikle selve modulet dvs. implementere modulet.

Modulspecifikation er også et væsentligt input, hvis man senere ønsker at genbruge et modul. Det er derfor vigtigt at beskrive alle de egenskaber, der er relevante for at kunne anvende eller genbruge modulet. Modulspecifikation skal således kort beskrive "Hvordan modulet anvendes" og "Hvad det kan".

Modulets implementering beskriver derimod, hvordan modulet internt er opbygget.



Figur 1. To sider af et vilkårligt modul

For et **softwaremodul** udarbejdes modulspecifikationen som en headerfil (*.h) og modulets implementering beskrives i en kodelinje (f.eks. *.cpp eller *.c).

For et **hardwaremodul** kan et modul specificeres ved at beskrive modulets grænseflade f.eks. via en beskrivelse af stikforbindelser, signalniveauer og signalernes logiske betydning og til tider suppleret med et logisk kredsløbsdiagram. For nogle hardwaremoduler vises både specifikation og implementeringen i form af et kredsløbsdiagram i samme dokumentation.

For integrerede kredsløb (IC'er) er det ofte kun benforbindelser, signalniveauer og logiske betydning, der dokumenteres i form af datablade. Her vil den interne implementering ikke sige den normale bruger noget. For en microprocessor omfatter specifikationen ud over beskrivelse af benforbindelser, niveauer etc. også dokumentation af det instruktionssæt, som processoren kan udføre.

I resten af denne note vil fokus primært være rettet mod anvendelsen af UML-Light i forbindelse med softwaredelen af et udviklingsprojekt, selv om dele af UML-Light også med god fornuft kan anvendes til at dokumentere dele af hardwaredesignet.

2 Basale OO begreber

2.1 Indkapsling (*Information Hiding*)

Objektbaseret udvikling og programmering er baseret på det klassiske indkapslingsprincip, der blev beskrevet tilbage i 1972 af David Parnas under navnet *Information Hiding* [Parnas72].

Dette princip har siden været grundlaget for design af SW-moduler og svarer til det modulbegreb, der f.eks. præsenteres i håndbogen ”*Struktureret Program Udvikling (SPU)*” [SPU88].

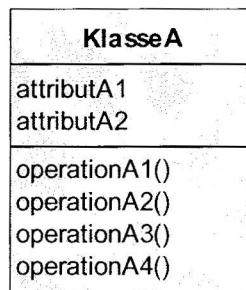
Dette modulbegreb kan implementeres i de traditionelle procedureorienterede programmerings-sprog som f.eks. C og assembler. I dag anvendes der i større og større udstrækning objektorienteerde programmeringssprog (f.eks. C++, Java og C#) som implementeringssprog, og her kan dette modulbegreb direkte implementeres vha. disse sprogs klassebegreb (*class*).

Information Hiding princippet går ud på, at man indkapsler en designbeslutning i et modul, der i OO terminologi implementeres vha. en klasse. En designbeslutning kan f.eks. være, hvordan en algoritme og den tilhørende datastruktur er designet. En anden designbeslutning kan f.eks. være at indkapsle viden om en konkret type hardware i en klasse (f.eks. porte og kontrolord), således at hardwaren senere kan udskiftes uden at skulle ændre i de klasser, der anvender denne.

Samtidig med at man indkapsler viden om algoritmer og datastrukturer i klassen, designer man klassens grænseflade. Al samspil med klassen foregår via dens grænseflade, der i SW består af et sæt af offentlige (*public*) operationer (UML betegnelsen for funktioner og metoder).

2.2 Klasser og objekter

En **klasse** er i software sammenhæng at betragte som en type, der definerer såvel attributter (data) som operationer (funktioner). Et **objekt** kan betragtes som en variabel af en given type (objektets klasse). Man kalder også et objekt for en instans, og man siger, at et objekt instantieres ud fra en given klasse.



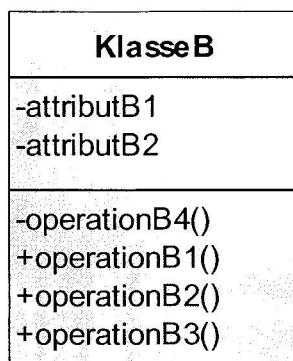
Figur 2. UML notation for en klasse

Figur 2 viser notationen for en klasse med navnet KlasseA, der har to attributter (datafelte) samt fire operationer (funktioner). En **attribut** beskriver en egenskab for en klasse og implementeres vha. en datatype. Klassen Bil kan f.eks. have attributterne: hastighed, gear, omdrejningstal og motorstørrelse. En **operation** beskriver en funktion, der kan udføres på et objekt af en given klasse. Klassen Bil kan f.eks. have operationerne: start, stop, udkoble, skift gear, kør og brems.

Regel:

En klasse navngives altid med et navneord i ental. Der må aldrig anvendes flertal.
Det er hensigtsmæssigt at navngive klasser med stort begyndelsesbogstav.

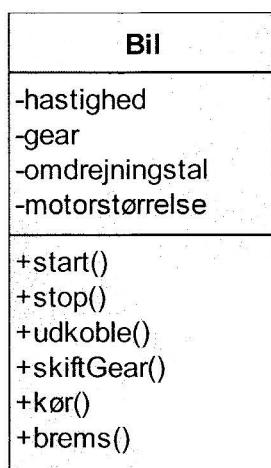
De fire operationer på Figur 2 udgør grænsefladen og bør derfor erklæres som *public*, hvorimod de to attributter bør være private. På denne måde vil KlasseA leve op til indkapslingsprincippet, hvor klassens attributter og operationernes implementering er skjult og kun kan tilgås via klassens grænseflade. Muligheden for at angive at medlemmer i en klasse er *public* eller *private* kaldes for **synlighed** (*visibility*). Det er muligt at indikere synlighed på et klassediagram med '+' for *public* og '-' for *privat*, som vist på Figur 3.



Figur 3. UML notation for en klasse med synlighed (*visibility*)

Eksemplet på Figur 3 viser at en operation, her *operationB4*, også kan være privat. Dette giver god mening, da operationen kan være en hjælpeoperation, der anvendes af en eller flere af de offentlige operationer.

En attribut kan principielt også erklæres som offentlig (*public*), men da det bryder indkapslingsprincippet, må dette stærkt frarådes. Har man brug for udefra at få fat på værdien af en attribut eller at ændre en attribut gøres dette ved at tilføje *public get* og *set* operationer til klassen.



Figur 4. Eksempel på en Bil klasse

Henholdsvis attributfeltet og operationsfeltet kan udelades på et givet klassediagram. Dette er specielt nyttigt ved det indledende analyse- og designarbejde, og når man ønsker at vise et overblikksdiagram.

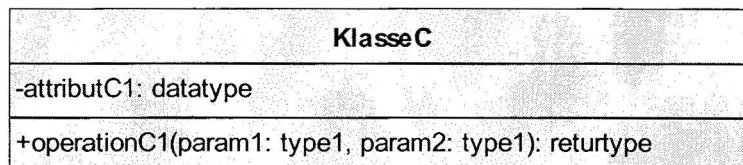
Et systems opbygning beskrives vha. et eller flere klassediagrammer, der viser systemets klasser og deres indbyrdes sammenhæng. Undertiden har man brug for også at vise konkrete instanser (objekter). Dette giver UML-Light også mulighed for vha. følgende objekt-notation. Et objekt navngives vha. objektnavn:klassenavn, der er understreget og diagrammeres som for en klasse. Et objekt kan også indeholde et attributfelt, der i det tilfælde vil vise de konkrete attributværdier for objektet.



Figur 5. UML notation for et objekt

2.3 Sammenhæng mellem klasse og implementering

Figur 6 viser notationen for at specificere en klasses attributter og deres datatype. Figuren viser også en komplet specifikation af en operation med de formelle parametre til operationen og deres datatyper samt operationens returtype. Som typeangivelse kan man anvende det pågældende programmeringssprogs indbyggede datatyper (f.eks. *char*, *int*, *long*, *double*) eller nye brugerdefinerede datatyper, der er defineret i det konkrete projekt. Er der tale om en operation, der ikke returner nogen værdi anvendes typen *void* som returtype. En sådan fuld specifikation af en operation kaldes også for operationens *prototype*.



Figur 6. Detaljeret specifikation af en klasse

Ved anvendelse af objektteknologi kan man ved at benytte følgende regel opnå en meget pån sammenhæng mellem et projekts klasser (specifikationen) og den tilhørende kode.

Regel:

En given klasse på et klassediagram, der anvendes ved design af software, har som standard en specifikationsfil og en implementeringsfil tilknyttet. Disse har samme navn som klassen.

For C++:	KlasseC.h	headerfilen, der udformes som en specifikationsfil
	KlasseC.cpp.	kodefilen, der indeholder implementeringen af klassen

For C:	KlasseC.h	headerfilen
	KlasseC.c	kodefilen

For Java og C#	i disse sprog beskrives begge dele i den samme fil, hvor man ved hjælp af et værktøj kan få udskrevet det der svarer til indholdet af specifikationsfilen
----------------	---

Som eksempel på den påne sammenhæng, der kan opnås mellem UML og koden vises her, hvorledes KlasseC på Figur 6 kan implementeres i C++.

Klassen defineres i headerfilen **KlasseC.h**:

```
class KlasseC
{
public:
    returntype operationC1(type1 param1, type2 param2);
private:
    datatype attributC1;
};
```

Klassen implementeres i kodefilen **KlasseC.cpp**:

```
returntype KlasseC::operationC1(type1 param1, type2 param2)
{
    // her skrives selve koden for operationC1
}
```

Dertil kommer de specielle operationer, der sørger for hhv. oprettelse af et objekt (*constructor* operationen) og den tilhørende nedlukningsoperation (*destructoren* operationen). Disse operationer medtages først på klassediagrammerne ved den detaljerede design af klasserne.

Regel:

Ved kodning af en klasse, der har defineret attributter, tilføjes der altid en *constructor* operation, der kaldes automatisk ved oprettelsen af objektet og sørger for, at attributternes dataværdier bliver passende initialiseret. En *constructor* operation har altid samme navn som klassen.

På samme måde kan man ved at definere en *destructorn* operation, der kaldes automatisk når objektet nedlægges og sørger for, at der automatisk bliver ryddet op.

En given klasse kan således starte ud med, i de indledende analysefaser, kun at være repræsenteret ved sit navn f.eks. KlasseC og vist som en kasse, der kun indeholder navnefeltet. Senere i analysen tilføjes der nogle attributter, der repræsenteres ved deres navn f.eks. hastighed og gear for klassen Bil. Senere hen tilføjes der operationer, der først optræder som navngivne operationer uden parametre for til slut at have fået tilføjet både parametre, returntype og markering af synlighed. Er der oprettelsesoperationer (*constructors*) med væsentlige parametre kan man også medtage disse på klassediagrammet.

Sensor
<ul style="list-style-type: none"> -sensorVaerdi: int -kalibreringsKst: unsigned char -portAddr: unsigned int
Sensor(portAdresse: unsigned int, kalibreringsKonstant: unsigned char) +laesVaerdi(): int

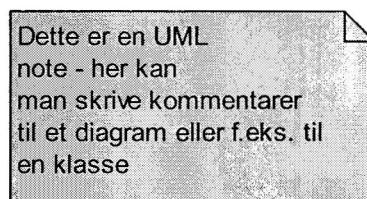
Figur 7. Eksempel på en detaljeret specifikation af en klasse

3 UML-Light diagrammer

I dette afsnit beskrives de basale UML diagramtyper, der er medtaget i UML-Light. Afsnittet indledes med at beskrive UML notationen for en kommentar (en note), da denne kan anvendes på alle UML diagramtyper.

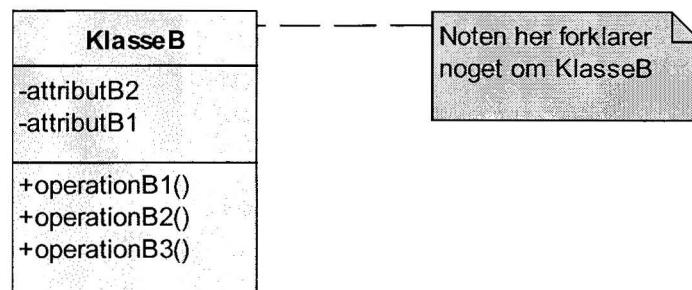
3.1 UML-Light note

En meget nyttig ting i UML-Light og UML er note faciliteten. Noter anvendes til at angive supplerende kommentarer for at forklare og supplere et diagram eller et diagramelement. Noter kan anvendes på alle UMLs diagramtyper. På ethvert diagram er det meget nyttigt at tilføje en note, der angiver dato for den sidste modifikation sammen med et evt. versionsnummer for diagrammet.



Figur 8. UML notation for en note

Ved hjælp af en stiplet linje kan noten hæftes på et diagramelement. På Figur 9 hører noten til en klasse.

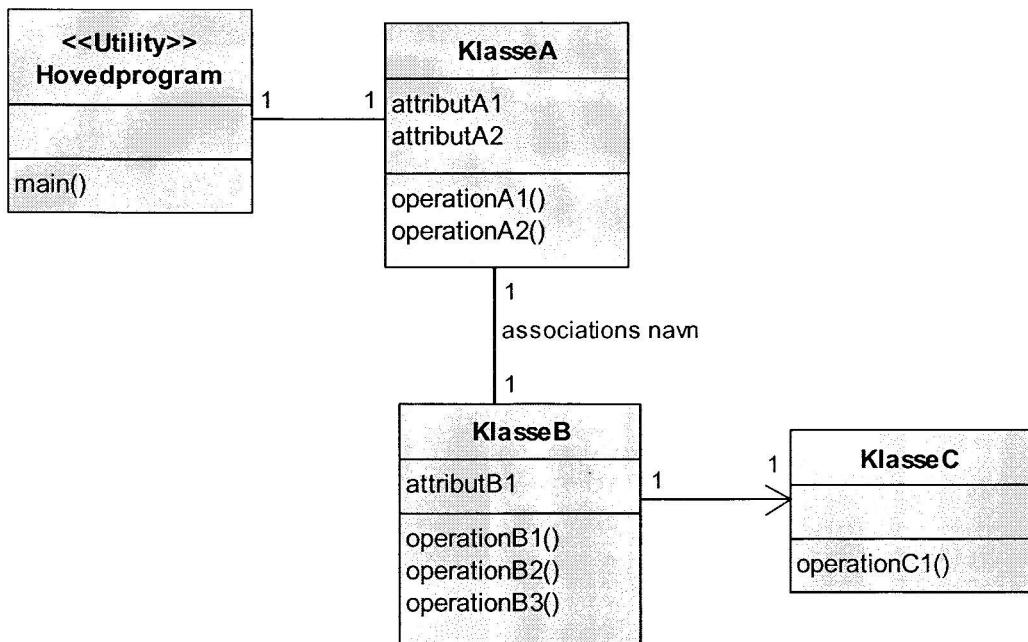


Figur 9. UML notation for en note tilknyttet en klasse

3.2 UML-Light klassediagram

Et klassediagram anvendes til at beskrive den *statiske* struktur af et system og benyttes således som notation ved designarbejdet, hvor systemet struktureres vha. klassebegrebet. Klassediagrammet er det vigtigste af UML-Light, UML-Light++ og UMLs diagramtyper.

Ud over at vise de klasser, der indgår i et givet system eller delsystem, viser klassediagrammet også deres indbyrdes sammenhæng, der i UML kaldes for associationer (*associations*).



Figur 10. Notation for et UML-Light klassediagram (klasser og associationer)

Regel:

For hver klasse på klassediagrammet beskriver man klassens ansvar. En klasses ansvar skal kort og præcist opsummere de operationer man kan udføre på klassen. Hvis man har problemer med at beskrive ansvaret for en given klasse kan det ofte skyldes at klassen er designet forkert og f.eks. skal splittes op i flere klasser.

En **association** angiver muligheden for, at objekter af de givne klasser kan kommunikere via en sådan forbindelse. Dette betyder, at et objekt af KlasseA på Figur 10 kan kommunikere med et KlasseB objekt ved, at én af KlasseA's operationer kalder enten operationB1(), operationB2() eller operationB3(). Omvendt kan KlasseB's operationer kalde en af KlasseA's operationer. Derimod kan man ikke fra en KlasseA operation direkte kalde KlasseC's operationC1(), da der ingen association er mellem kasserne. De viste associationer på Figur 10 er **tovejs** associationer dvs. de giver muligheder for kald i begge retninger. Hvis der kun kræves kald i den ene retning, er associationen **envejs**, hvilket indikeres ved at sætte en pil på associationslinien som angivet ved KlasseC.

Tallene på Figur 10 kaldes for **multipliciteter** (*multiplicity*) og anvendes til at angive det mulige antal af objekter, der kan/skal instantieres ud fra klassediagrammet. I det viste eksempel hører der præcis ét objekt til hver klasse.

Der er generelt i UML følgende muligheder for multipliciteter: 0..1, 0..*, *, 1..*, n, n..m, hvor * indikerer mange dvs. et ubegrænset antal af objekter.

En association kan som en option have et **associationsnavn**, der beskriver associationen. Navnet anbringes midt mellem kasserne. Associationen beskrives ofte vha. et udsagnsord, således at klasse1 + associationsnavn + klasse2 kan læses som en sætning f.eks. *KontrolPanel styrer LogikAnalysator*.

Figur 10 viser et eksempel på en facilitet i UML, der kaldes for en **stereotype**. Denne facilitet, der i UML markeres med '«navn »', anvendes til at tilføje supplerende information om et UML element, her en klasse. På Figur 10 er hovedfunktionen *main()* anbragt i klassen "Hovedprogram", der har fået tilføjet stereotypen «Utility». Stereotypen Utility anvendes til at diagrammere f.eks. C operationer, der ikke er defineret som en del af en normal klasse. I eksemplet her anvendes den til, at symbolisere hovedprogrammet med C operationen *main()*. For større systemer vil man normalt ikke medtage *main()* operationen og dens tilhørende "Utility" klasse på et klassediagram, da et godt designet system normalt har meget lidt funktionalitet i selve *main()* funktionen.

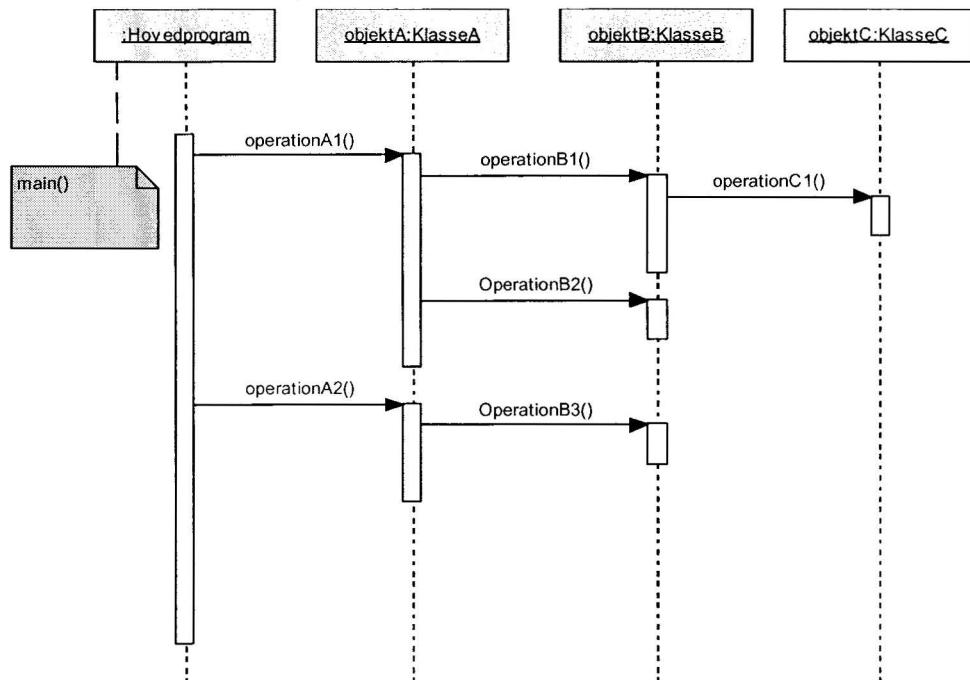
3.3 UML-Light sekvensdiagram

Sekvensdiagrammer anvendes til at beskrive de *dynamiske* sammenhænge i et system. Det er ved hjælp af disse diagrammer man viser, hvorledes de forskellige objekter i et system kommunikerer, for at udføre en af systemets funktionaliteter.

Det er vigtigt at gøre sig klart, at sekvensdiagrammer altid kun beskriver ét bestemt forløb, der kaldes for et **scenario**. Et sekvensdiagram viser et tidsforløb, hvor tiden vokser nedad.

Regel:

Man bør altid have mindst et sekvensdiagram, der beskriver et normalt scenario. Hvis der er vigtige afvigelsesforløb, bør man også medtage sekvensdiagrammer, der viser disse scenarier. Scenariet bør kort navngives vha. en note, der beskriver scenariet og versionsdato.



Figur 11. Notation for et UML-Light sekvensdiagram

Et sekvensdiagram viser et antal objekter og hvordan disse vha. meddelelser (operationskald) kommunikerer med hinanden for at udføre en af systemets funktioner. I Figur 11 er objekterne

angivet vha. objektnavn:Klassenavn. Den stiplede linie under objektet angiver objektets livslinje. De firkantede blokke på livslinjen angiver aktiveringstiden for en given operation, dvs. at operationen er udført, når blokken ender. I eksemplet kalder hovedprogrammet operationerne: operationA1() og operationA2() efter hinanden. Figuren viser også, hvorledes f.eks. operationA1() udføres som et samspil mellem objekterne: objektA, objektB og objektC.

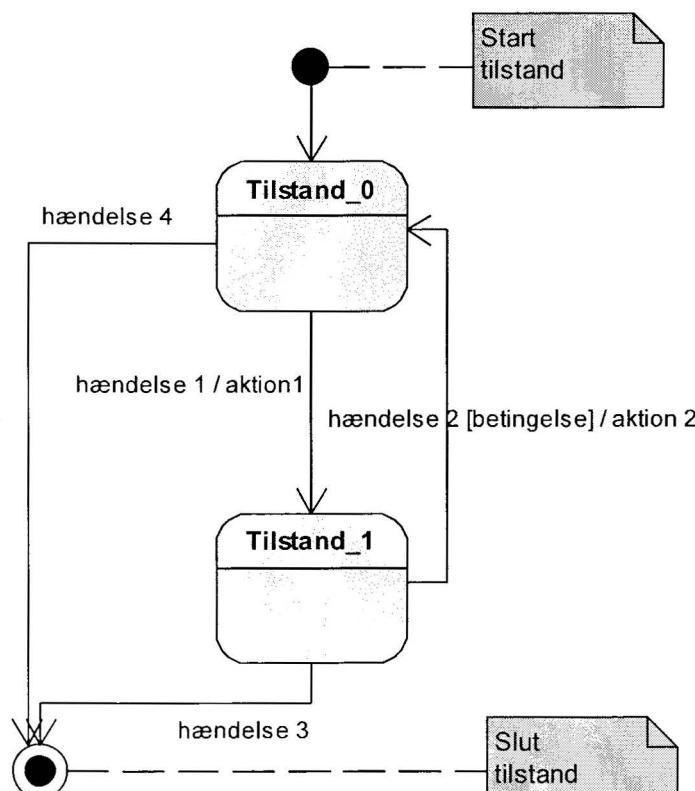
Det kan undertiden være nyttigt at angive en **betinget udførelse** af en operation. Dette er muligt ved at tilføje en betingelse i firkantede parenteser foran en meddeelse f.eks. [strøm på batteriet]start, hvor *start* kun sendes, såfremt der er strøm på batteriet.

3.4 UML-Light tilstandsdiagram

Tilstandsdiagrammer anvendes til at designe og dokumentere tilstandsmaskiner. En tilstandsmaskine kan anvendes til at beskrive en tilstandsafhængig opførsel af såvel en hardware- som en softwareenhed. Anvendelse af tilstandsmaskiner er meget nyttig og ofte anvendt både som modellerings- og implementeringsredskab ved udvikling af tekniske systemer og dette gælder både for software- og hardwareudvikling.

UML-Light omfatter kun den helt basale notation for tilstandsdiagrammer, hvorimod UML har et noget større sæt af notationer og begreber. Tilstandsdiagrammer kaldes i UML for *State Diagrams*.

Et tilstandsdiagram hører i UML altid til en enkelt klasse og beskriver dennes tilstandsafhængige opførsel. Figur 12 viser et eksempel på notationen for et UML-Light tilstandsdiagram.



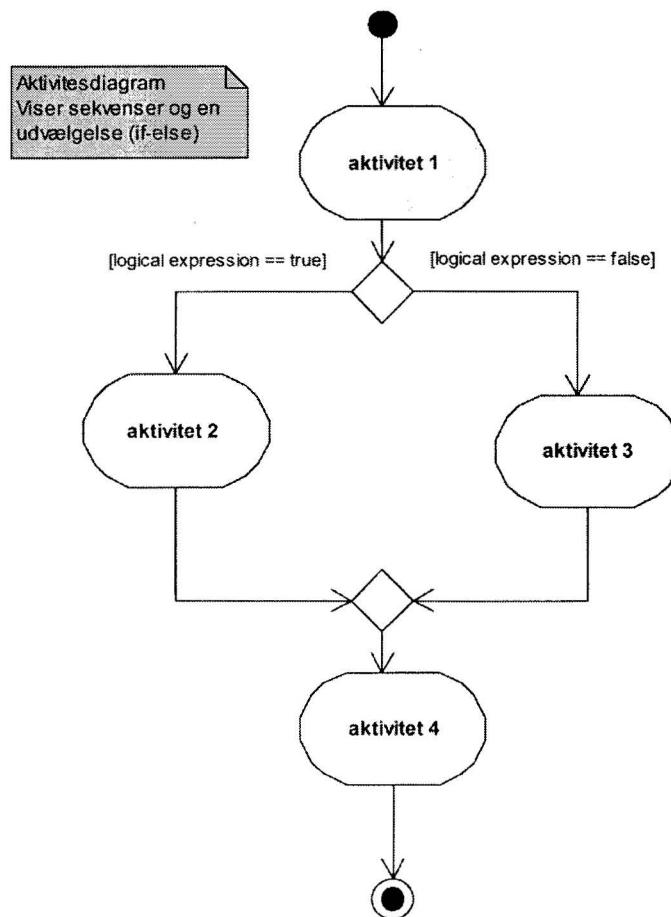
Figur 12. Notation for et UML-Light tilstandsdiagram

En **tilstand (state)** beskriver en veldefineret og statisk tilstand af et system, der kan være af kortere eller længere varighed. Systemet bliver i en given tilstand indtil, der modtages en **hændelse (event)**, der svarer til et af navnene på de udadgående pile. Pilene mellem to tilstade angiver en **tilstandsovergang (transition)**. Sammen med angivelsen af hændelsen kan der også angives en eller flere aktioner (*actions*), der skal udføres ved tilstandsovergangen. En **aktion** angiver en korterevarende operation f.eks. på et objekt. Tilstandsdiagrammet viser også hhv. en starttilstand og en sluttillstand. Starttilstanden skal altid angives, hvorimod sluttillstanden er optional.

Et tilstandsdiagram kan også have en **betinget tilstandsovergang**, der angives som en betingelse (*guard*) i firkantede parenteser sammen med hændelsen. En **betingelse** skal være et logisk udtryk, der har værdien sand eller falsk. På figuren er tilstandsovergangen mellem Tilstand_1 og Tilstand_2 således afhængig af, at hændelse 2 indtræffer og at den tilhørende betingelse er sand.

3.5 UML-Light aktivitetsdiagram

UMLs aktivitetsdiagrammer kan generelt anvendes på flere niveauer ved udvikling af et system eller et produkt. I UML-Light anvendes aktivitetsdiagrammer **kun** til detaljeret design af en procedure eller en funktion, der i UML termer kaldes for en operation (*operation*). Aktivitetsdiagrammer kan i denne udgave anvendes som et rutediagram (*flowchart*).



Figur 13. Notation for et UML-Light aktivitetsdiagram (sekvens + udvælgelse)

Figur 13 viser, hvorledes man vha. af et aktivitetsdiagram kan diagrammere en udvælgelse (if / else), hvor enten aktivitet 2 eller aktivitet 3 udføres afhængigt af om det logiske udtryk, der er

angivet i de firkantede parenteser, er hhv. sandt eller falsk. Ovalerne på figuren beskriver diagrammets aktiviteter. En **aktivitet** (*activity*) beskriver udførelse af en funktionalitet, der ophører efter en given tid. Når aktiviteten er udført, skiftes der til den efterfølgende aktivitet, hvortil der er en udadgående pil. Ud over aktiviteter kan man vha. en rhombe angive en **udvælgelse** (*choice*), hvor man på de udadgående pile efter rhomben angiver de betingelser, der er for at en given vej vælges. Betingelserne angives i firkantede parenteser og skal indeholde et logisk udtryk. Et aktivitetsdiagram er i UML sammenhæng en afart af et tilstandsdiagram, dog uden hændelser og aktioner.

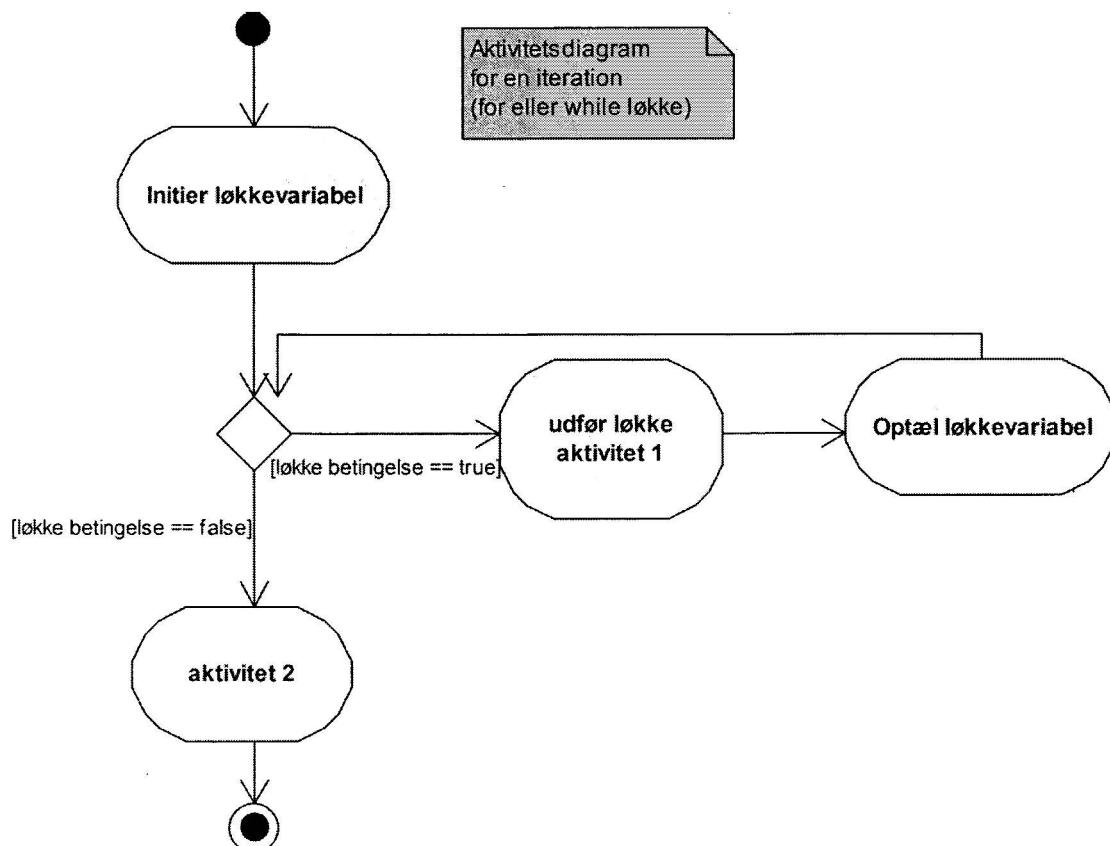
I stedet for at anvende et aktivitetsdiagram til detaljeret design af en operation, kan man ofte med fordel udtrykke dette vha. pseudokode, da denne beskrivelsesform er lettere at vedligeholde. Pseudokode for eksemplet på Figur 13:

```

aktivitet 1
if (logical expression == true)
    aktivitet 2
else
    aktivitet 3
aktivitet 4

```

Figur 14 viser, hvorledes et aktivitetsdiagram kan anvendes til at diagrammere en løkke (en iteration) svarende til en *for* eller en *while* løkke i C++.



Figur 14. Notation for et UML-Light aktivitetsdiagram (iteration)

3.6 Sammenhæng mellem UML-Light diagrammer

3.6.1 Sammenhæng mellem klassediagram og sekvensdiagram

Et sekvensdiagram viser altid et sæt af instantierede objekter, der er oprettet ud fra de muligheder, der er vist på klassediagrammet.

Sekvensdiagrammet viser, hvorledes objekterne kommunikerer ved at kalde de til klassen hørende operationer. Et sekvensdiagram viser normalt et bestemt scenario. Derfor kan der til et bestemt klassediagram høre et sæt af sekvensdiagrammer, der beskriver de forskellige mulige scenarier f.eks. normalscenariet suppleret med et antal undtagelsesscenarier.

Verifikation:

Hvis der er et kald mellem to objekter på sekvensdiagrammet, så skal der også være en tilsvarende association mellem de tilhørende klasser på klassediagrammet. Kalder objektA operationB1() på objektB, så skal den tilhørende klasse KlasseB på klassediagrammet have en tilsvarende operation med samme navn og parametre og returntype.

3.6.2 Sammenhæng mellem klassediagram og tilstandsdiagram

Et tilstandsdiagram beskriver altid opførslen af én og kun én klasse, dvs. at tilstandsdiagrammet hører til en bestemt klasse. Omvendt så er det ofte kun nogle få af et systems klasser, der har en tilstandsafhængig opførsel og derfor er beskrevet vha. en tilstandsmaskine.

Hændelserne på tilstandsdiagrammet kan modelleres som operationer på den tilhørende klasse. Aktionerne modelleres også som operationer enten på den samme klasse eller på en af de associerede klasser.

Verifikation:

At såvel hændelser som aktioner optræder som operationer på den tilhørende klasse eller at aktionerne optræder som operationer på en eller flere af de associerede klasser.

3.6.3 Sammenhæng mellem klassediagram og aktivitetsdiagram

Aktivitetsdiagrammet beskriver den interne virkemåde af én af én klasses operationer. Aktiviteterne på diagrammet kan være simple udtryk, der ændrer attributternes værdi, kald af private operationer i klassen eller kald af operationer på associerede objekter.

3.7 Diagrammering af Hardwareenheder

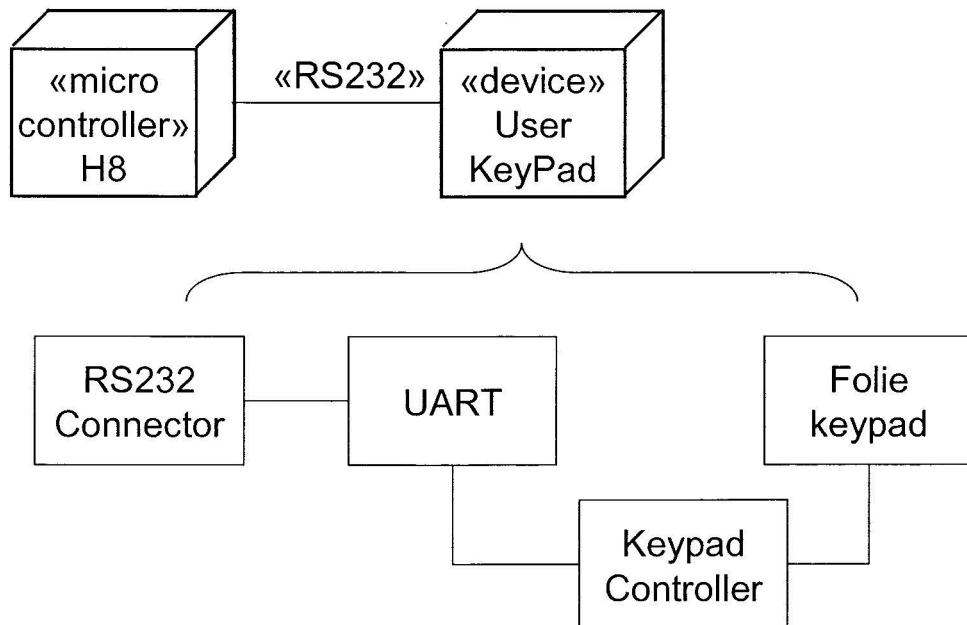
Hardwareenheder kan diagrammeres vha. et UML *Deployment* diagram. Disse diagrammer er meget nyttige, da man ved hjælp af disse kan sammenkoble design og dokumentation af såvel hardware- som softwaredelen af et system.



Figur 15. Deployment diagram

En hardwareenhed diagrammeres i UML som en *node* vha. kassenotationen som vist på Figur 15. Ved hjælp af en stereotype («navn») kan man angive typen af noden, det kan f.eks. være en «processor» eller en «device». På associationen mellem de to nodes kan man ligeledes vha. en stereotype angive typen af kommunikationsmedie eller bus f.eks. «RS232» eller «PC104».

Opbygningen af en hardwareenhed vises som et blokdiagram, der viser funktionsblokke og deres forbindelser. Dette blokdiagram kan udformes som et UML klassediagram vha. UMLs klassesymbol i den basale form dvs. uden attribut- og operationsfelt. Associationer anvendes til at vise forbindelserne mellem funktionsblokkene, ligesom man kan tilknytte noter til diagrammet og funktionsblokkene – se eksempel på Figur 16.

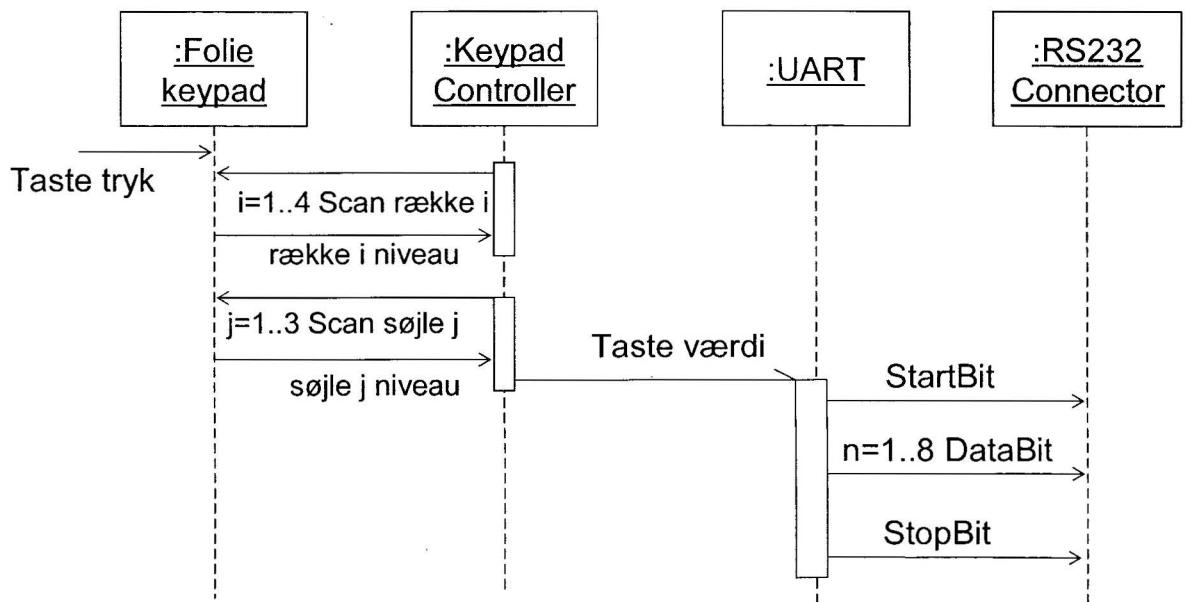


Figur 16. Hardware blokdiagram eksempel

Disse oversigtsdiagrammer suppleres med de normale kredsløbsdiagrammer, der viser det detaljerede design af den digitale og/eller analoge hardware.

Ved udvikling af digital hardware, der er baseret på tilstandsmaskiner vil det være oplagt at anvende UML-Light notationen for tilstandsdiagrammer, da man derved kan anvende de samme værktøjer ved hardware- og softwareudviklingen.

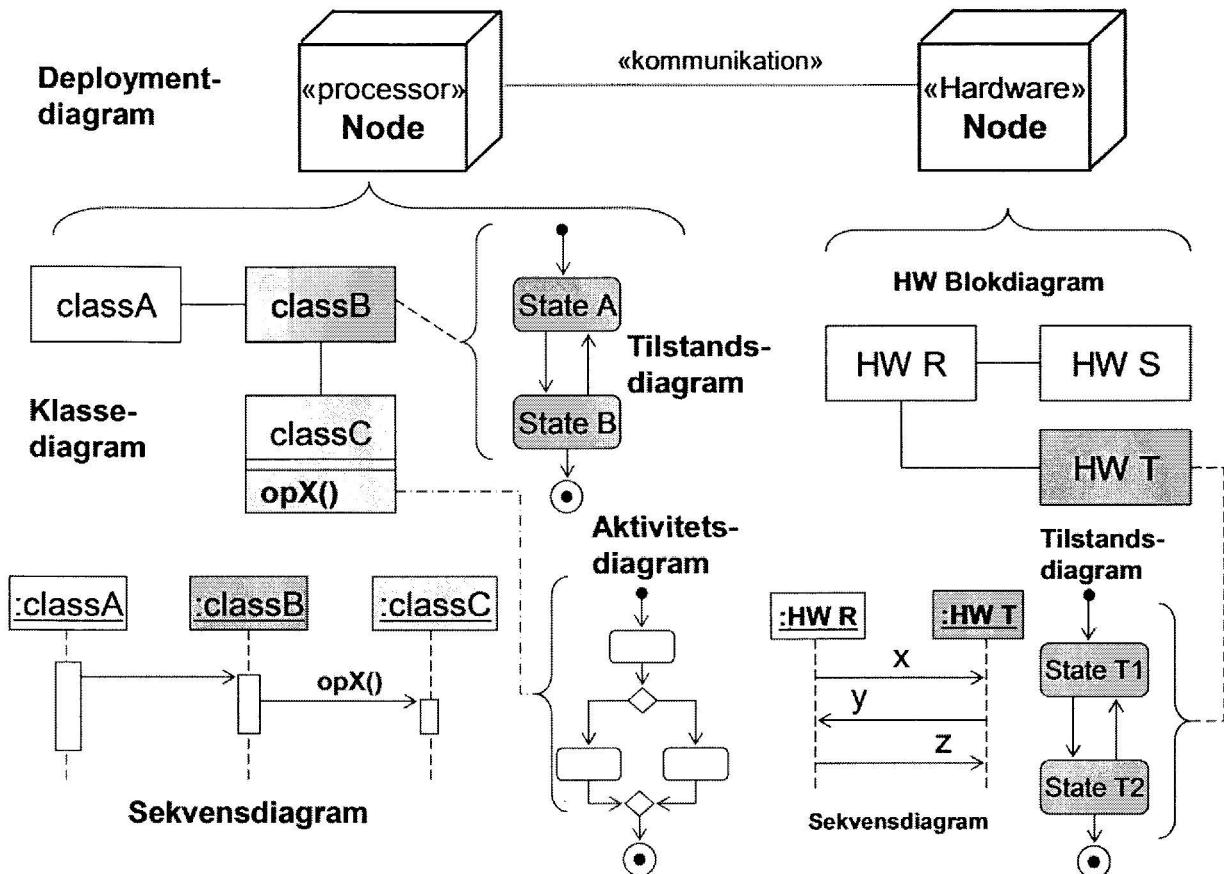
En tredje type af diagrammer der kan anvendes ved hardwareudvikling er UML-Lights sekvensdiagrammer, der anvendes til at vise kommunikationen mellem hardwarefunktionsblokkene dvs. hvilke signaler der anvendes og deres timing.



Figur 17. Hardware sekvensdiagram eksempel

3.8 UML-Light oversigtsdiagram

Oversigtsdiagrammet på Figur 18 viser de forskellige UML-Light diagramtyper samt deres anvendelse ved udvikling og dokumentation af hhv. Software- og Hardwaredelen af et system eller produkt.



Figur 18. Oversigtsdiagram over UML-Lights diagramtyper

Deploymentdiagrammet viser den fysiske hardware, hvor der er mulighed for at vise såvel enheder (nodes), der har en processor og dermed SW tilknyttet og de enheder, der er implementeret i hardware.

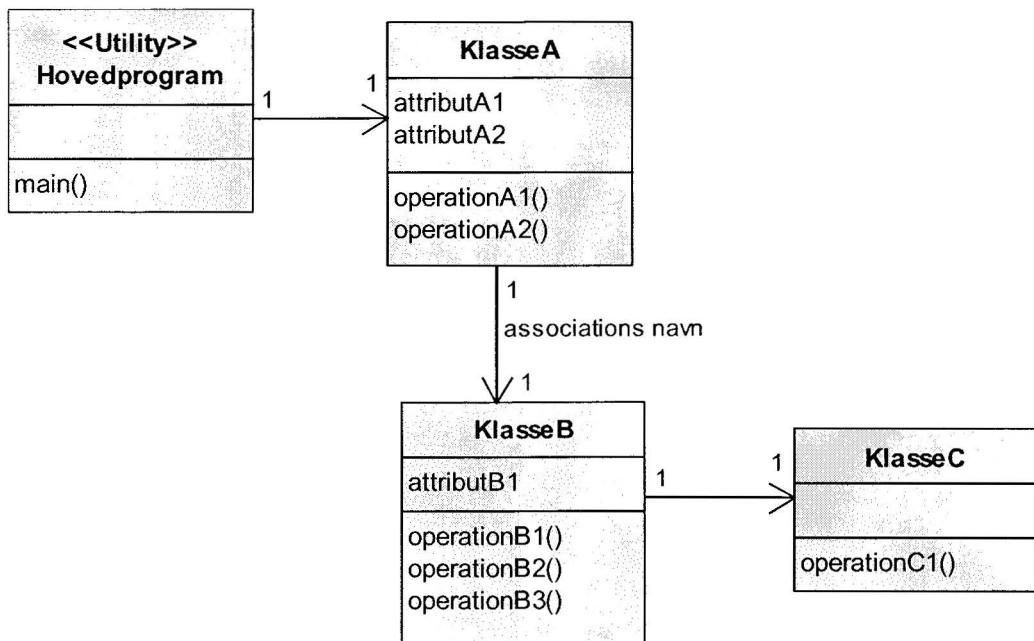
For processorenheden er der vist et klassediagram, der giver et overblik over de klasser, som softwaren er opdelt i og dermed også et overblik over de tilhørende kode- og headerfiler. For de klasser, der har en tilstandsafhængig opførsel, er denne opførsel beskrevet vha. et tilstandsdiagram (her for klassen classB). Samspillet mellem objekterne i det kørende program er vist på et sekvensdiagram, der viser hvilke operationer, der kaldes på objekterne. Endelig kan man vha. et aktivitetsdiagram vise, hvorledes en enkelt operation er detaildesignet. Et alternativ til at anvende et aktivitetsdiagram er som tidligere nævnt, at beskrive operationen vha. pseudokode.

For hardwareenheden kan man vise dens opbygning vha. et blokdiagram (et simplificeret UML klassediagram eller alternativt et SysML blok definition diagram) evt. suppleret med et tilstandsdiagram, hvis der er anvendt tilstandsmaskiner i designet. Samspillet kan vises vha. et eller flere sekvensdiagrammer som vist på Figur 17.

4 Sammenhæng mellem klassediagram og kode

4.1 Klassediagram og *include*-filer

Her vises hvorledes klassediagrammet på Figur 19 hænger sammen med kodning i C++.



Figur 19. Klassediagram og kode

Klassen hovedprogram indeholder selve *main()* funktionen, der er en C-funktion.

C-funktioner kan generelt medtages i UML-Light ved at angive stereotypen «Utility» over klas senavnet.

For hver klasse på diagrammet, undtagen Hovedprogram Utility-filen, udarbejdes der hhv. en headerfil (*.h) og en implementeringsfil (*.cpp).

Klassediagrammet på Figur 19 kan direkte mappes over på følgende filer:

```

Hovedprogram.cpp
KlasseA.cpp, KlasseA.h
KlasseB.cpp, KlasseB.h
KlasseC.cpp, KlasseC.h
  
```

Regel 1:

Hver klasse skal i sin implementeringsfil (*.cpp) inkludere sin egen headerfil, således at compileren kan checke, at definitionerne i headerfilen er implementeret korrekt i implementeringsfilen.

Regel 2:

Hver klasse, der har en association til en anden klasse, skal inkludere dennes headerfil (for envejs associationer inkluderer den klasse, der har en envejs association til en anden klasse, kun den klasse som envejs associationen peger på).

Anvendelse af disse to regler på det viste klassediagrameksempel, giver følgende resultat:

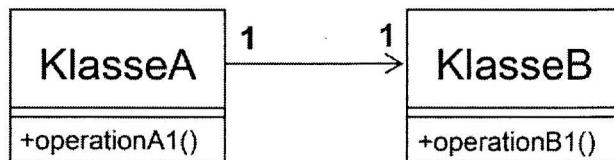
Hovedprogram.cpp:	<code>#include "KlasseA.h"</code>
KlasseA.cpp:	<code>#include "KlasseA.h"</code> <code>#include "KlasseB.h"</code>
KlasseB.cpp:	<code>#include "KlasseB.h"</code> <code>#include "KlasseC.h"</code>
KlasseC.cpp:	<code>#include "KlasseC.h"</code>

4.2 Implementering af associationer

Her vises eksempler på, hvorledes forskellige typer af et klassediagrams associationer kan implementeres i C++.

Envejs 1-1 association

Figur 20 viser et eksempel på en envejs association, der i dette tilfælde altid vil forbinde et KlasseA objekt med præcist ét KlasseB objekt.



Figur 20. Envejs 1-1 association

Da denne association er en envejs 1-1 association, kan den implementeres vha. en KlasseB pointer i KlasseA.

```

class KlasseA
{
public:
    KlasseA(KlasseB* pB);      // Constructor
    void operationA1();
private:
    KlasseB* pKlasseB; // implementerer associationen
}
  
```

For at sikre at associationen bliver korrekt initialiseret, tilføjes der en parameter til *constructor* operationen, således at dette sker automatisk ved oprettelse af objektet (det sørger compileren for at checke). Dette medfører samtidigt, at man skal oprette KlasseB objektet, før man kan oprette KlasseA objektet.

Constructor operationen bliver således:

```
KlasseA::KlasseA(KlasseB* pB)
{
    pKlasseB= pB;

    // her indsættes den øvrige initialiseringskode
    // for klassens øvrige attributter
}
```

Som et eksempel på hvordan objekter af KlasseA og KlasseB oprettes og initialiseres, vises her et simpelt *main()* program.

```
int main()
{
    KlasseB objektAfKlasseB;
    KlasseA objektAfKlasseA(&objektAfKlasseB);

    // nu har vi dannet forbindelsen således at én af
    // operationerne i KlasseA objektet kan kalde
    // operationB1() i det KlasseB objekt vi har oprettet

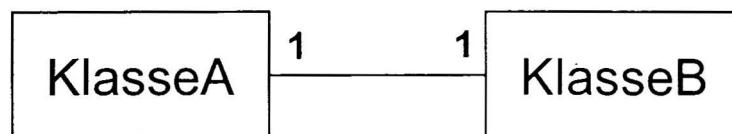
    objektAfKlasseA.operationA1();
    return (0);
}
```

Her vises et eksempel på, hvordan en KlasseA operation kalder operationB1 på et KlasseB objekt:

```
KlasseA::operationA1()
{
    pKlasseB->operationB1();
}
```

Tovejs 1-1 association

Ved en tovejs 1-1 associationen tilføjes der på tilsvarende måde en pointer til den anden klasse og denne skal også initialiseres, hvilket også kan forgå i klassens *constructor* operation.



Figur 21. Tovejs 1-1 association

I eksemplet på Figur 21 vil det se ud på følgende måde:

```
class KlasseA
{
public:
    KlasseA(KlasseB* pB);      // Constructor
private:
    KlasseB* pKlasseB; // implementerer associationen
}
```

```
class KlasseB
{
public:
    KlasseB();      // Constructor
    initAssociation(KlasseA* pA);
private:
    KlasseA* pKlasseA; // implementerer associationen
}
```

Denne viste løsning kræver, at et objekt af KlasseB oprettes før et objekt af KlasseA. Der er her tilføjet operationen *initAssociation()* til KlasseB, som kan initialisere pointeren til KlasseA.

KlasseA *constructoren* kommer til at se ud på følgende måde:

```
KlasseA::KlasseA(KlasseB* pB)
{
    pKlasseB= pB;
    pKlasseB->initAssociation(this);
}
```

Her initialiseres associationen fra KlasseB til KlasseA ved at kalde *initAssociation* operationen med KlasseA objektet som parameter via *this* pointeren.

Anvendelsen af disse klasser ser således ud i *main()*:

```
int main()
{
    KlasseB objektAfKlasseB;          // skal oprettes først
    KlasseA objektAfKlasseA(&objektAfKlasseB);

    // nu har vi dannet en tovejsforbindelse således at én af
    // operationerne i KlasseA objektet kan kalde en af
    // operationerne i KlasseB objektet og omvendt

    objektAfKlasseA.operationA1();    // kalder f.eks. operationB1()
    objektAfKlasseB.operationB2();    // kalder f.eks. operationA2()

    return (0);
}
```

0-N association

I de tilfælde, hvor man har en 0-N association, hvor N er en konstant, kan man i stedet for at anvende én pointer, anvende et array af pointere f.eks. KlasseC *pKlasseC[n].

Disse skal så på tilsvarende måde initialiseres enten i en *constructor* operation eller ved at tilføje en *initAssociation* operation.

0-* association

Dette er det helt generelle tilfælde, hvor der kan være et ubegrænset (i teorien) antal objekter forbundet til et andet objekt. Her vil man anvende en dynamisk datastruktur (f.eks. en linket liste) ved implementeringen, men dette vil ikke blive yderligere omtalt i denne note.

5 Eksempel 1: Minutur designet med UML-Light

5.1 Beskrivelse af eksemplet

Dette eksempel viser en model for et simpelt minutur. Minuturet kan startes, stoppes og resettes ved hjælp af tre knapper, der er anbragt på et knappanel. Knappanelet er forbundet til en inputport på en microcontroller. Når minuturet er startet, optæller det tiden og viser den i minutter og sekunder på et display i formatet minutter:sekunder. Displayet er forbundet til en outputport på microcontrolleren. Til at styre tiden anvendes en i microcontrolleren indbygget timerkreds, der kan tælle i millisekunder. For at simplificere eksemplet mest muligt vil hovedprogrammet (main) her *polle* (periodisk aflæse) både knappanel og timer.



Figur 22. Skitse af brugergrænseflade for Minutur

5.2 UML-Light model

Som det ses af klassediagrammet på Figur 23, består programmet af fire objekter, der er instantieret ud fra klasserne Ur, Timer, Display og KnapPanel. Hovedprogrammet er implementeret vha. funktionen *main()*.

Det er som tidligere nævnt vigtigt at supplere et klassediagram med en beskrivelse af hver klassens ansvar.

Klasserne på Figur 23 har følgende ansvar:

Hovedprogram:

Denne klasse af typen "utility" repræsenterer selve hovedprogrammet, der er repræsenteret ved C-funktionen *main()*. Hovedprogrammet poller hhv. KnapPanel og Timer objektet for hændelserne *start*, *stop*, *reset* og *timeout*, der sendes til Ur objektet for behandling.

Ur:

Denne klasse har ansvaret for at implementere selve minuturfunktionaliteten. Klassen er beskrevet vha. en tilstandsmaskine, der er vist på tilstandsdiagrammet på Figur 24.

Timer:

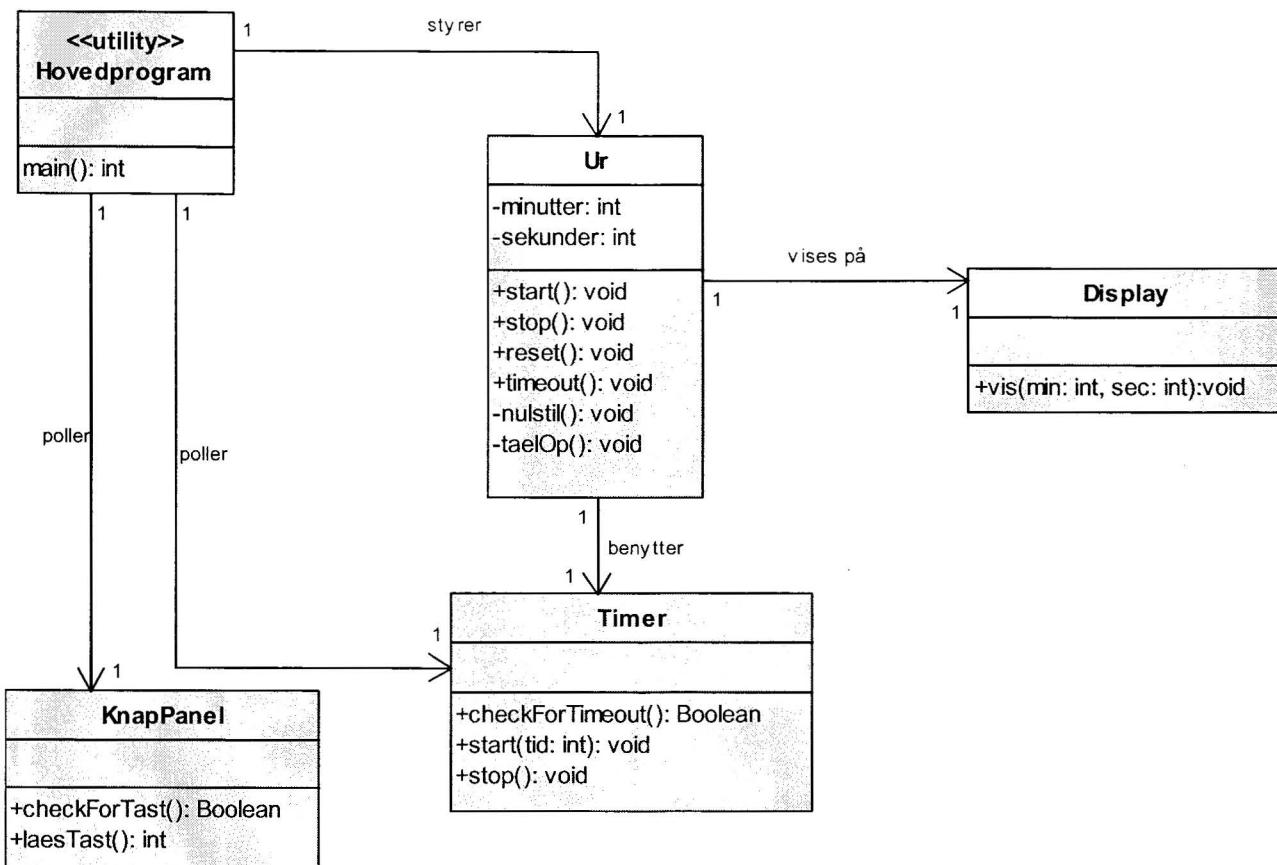
Denne klasse har ansvaret for at implementere SW grænsefladen til en hardwaretimer. Hardwaretimeren kan startes med et tælletal i millisekunder, hvorefter den tæller ned til 0.

KnapPanel:

Denne klasse har ansvaret for at teste om en knap er aktiveret og i givet fald at kunne aflæse den aktuelle knaps værdi. Klassen håndterer et KnapPanel med knapperne *Start*, *Stop* og *Reset*.

Display:

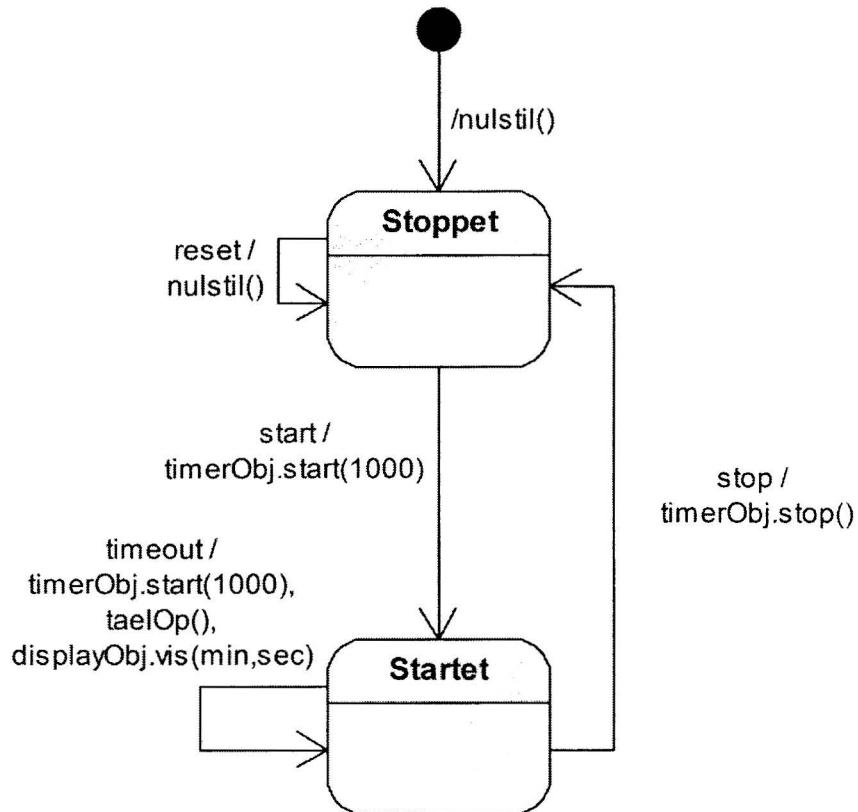
Denne klasse har ansvaret for at vise tiden i minutter og sekunder på et display med fire cifre og et kolon (format mm:ss).



Figur 23. Klassediagram for Minutur

Operationerne på klassediagrammet er fundet ved at udarbejde de to følgende diagrammer.

Klassen **Ur** er designet vha. en tilstandsmaskine, der er vist på tilstandsdiagrammet på Figur 24.

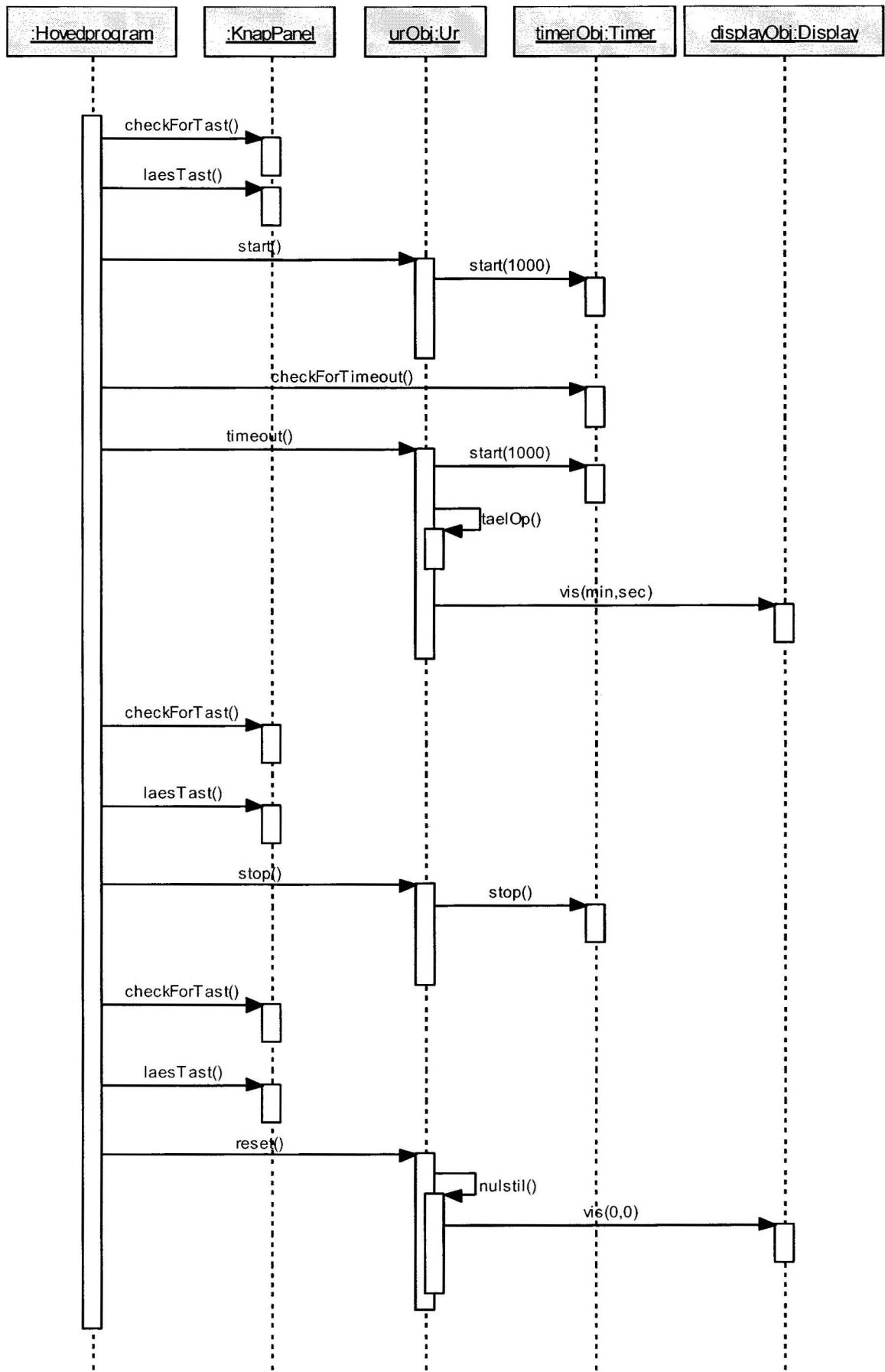


Figur 24. Tilstandsdiagram for Minuturet

Tilstandsdiagrammet har to tilstande *stoppet* og *startet*. I tilstanden *stoppet* har man mulighed for at nulstille minuturet vha. hændelsen ”reset”, der kommer, når man aktiverer *reset* tasten. Når brugeren aktiverer *start* tasten, vil der blive sendt en ”start” hændelse til tilstandsmaskinen, hvorefter der skiftes tilstand samtidig med, at der sendes en ”start” meddelelse til timerobjektet *timerObj*. Når tiden er gået (1000ms), modtages hændelsen ”timeout”, der bevirker at operationen *taelOp()* og operationen *vis(min,sec)* kaldes samtidig med, at der sendes en ny ”start” meddelelse til timerobjektet (*timerObj*).

Nu mangler vi kun at beskrive sammenspiellet mellem hovedprogrammet og de fire objekter, der instantieres ud fra klassediagrammet. Dette sammenspiel beskrives vha. UML-Light sekvensdiagrammet på Figur 25. Der er her kun vist et enkelt forløb af hhv. et *start*, et *timeout*, et *stop* og et *reset* scenario.

I de følgende to afsnit vises implementering af minuturet i henholdsvis C++ og i C.



Figur 25. Sekvensdiagram for Minutur

5.3 Minutur implementeret i C++

I dette afsnit vises, hvorledes de vigtigste af klasserne på klassediagrammet på Figur 23 kan implementeres vha. C++ kode.

Prøv at sammenligne koden med klassediagrammet og se den fine sammenhæng, det er muligt at opnå mellem et UML klassediagram (=design) og den tilhørende C++ kode (=implementering).

Utilityklassen *Hovedprogram* implementeres i filen ***Hovedprogram.cpp***:

```
#include "Ur.h"
#include "KnapPanel.h"          // definerer datatypen KnapHaendelse
#include "Timer.h"
#include "Display.h"

int main()
{
    KnapPanel knapPanelObj; // her oprettes de fire objekter som anvendes
    Timer timerObj;
    Display displayObj;
                // ur objektet bliver her initialiseret med
                // pointere til et timer- og et displayobjekt
    Ur urObj(&timerObj,&displayObj);

    KnapHaendelse knapHaendelse;

    while ( true )           // Uendelig hovedløkke i programmet
    {
        // polling af knappanel
        if (knapPanelObj.checkForTast())
        {
            // task aktiveret
            knapHaendelse= (KnapHaendelse) knapPanelObj.laesTast();
            if (knapHaendelse == START)
                urObj.start();
            else if (knapHaendelse == STOP)
                urObj.stop();
            else if (knapHaendelse == RESET)
                urObj.reset();
        }
        // polling af timer for timeout
        if (timerObj.checkForTimeout())
            urObj.timeout();
    }
    return 0;
} // end main
```

I C++ anvendes *enum* til at definere en såkaldt enumeration type, der definerer en datatype, der kun kan antage et bestemt antal værdier. I eksemplet definerer filen Knappanel.h datatypen *KnapHaendelse* vha. følgende erklæring:

```
enum KnapHaendelse {START=1, STOP=2, RESET=3};
```

En variabel af typen *KnapHaendelse* kan kun antage værdien START, STOP eller RESET, der ved defineringen også har fået tildelt de tilhørende konstantværdier. Anvendelsen af enumeration types gør programmet lettere at læse og anvendelsen sikrer samtidig også at konstanterne ikke overlapper i værdi.

Klassen *Ur* specificeres i filen ***Ur.h***:

```
#include "Timer.h"
#include "Display.h"

class Ur
{
public:
    Ur(Timer*, Display*);      // constructor operation
    void start();
    void stop();
    void reset();
    void timeout();
private:
    void nulstil();           // privat da den kaldes via tilstandsmaskinen
    void taelOp();             // privat da den kaldes via tilstandsmaskinen

    int minutter;
    int sekunder;
    Display* mitDisplay;
    Timer* minTimer;
    enum TILSTAND {STARTET, STOPPET} tilstand;
    const static int TIME1000MS= 1000;
};
```

Klassen *Ur* implementeres i filen ***Ur.cpp***, der implementerer tilstandsmaskinen. Her er tilstandsmaskinen direkte implementeret i kodden på den simplest mulige måde, der er ok for dette enkle eksempel.

```
#include "Ur.h"
#include "Timer.h"
#include "Display.h"

Ur::Ur(Timer* timerPtr, Display* displayPtr)
{                               // constructor operation
    minTimer= timerPtr;
    mitDisplay= displayPtr;
    tilstand = STOPPET;
    nulstil();
}

void Ur::start()           // håndterer hændelsen: start
{
    if (tilstand == STOPPET)
    {
        tilstand= STARTET;
        minTimer->start(TIME1000MS);
    }
}

void Ur::stop()            // håndterer hændelsen: stop
{
    if (tilstand == STARTET)
    {
        tilstand= STOPPET;
        minTimer->stop();
    }
}
```

```
void Ur::reset()          // håndterer hændelsen: reset
{
    if (tilstand == STOPPET)
    {
        nulstil();
    }
}

void Ur::timeout()         // håndterer hændelsen: timeout
{
    if (tilstand == STARTET)
    {
        minTimer->start(TIME1000MS);
        taelOp();
        mitDisplay->vis(minutter, sekunder);
    }
}

void Ur::nulstil()          // privat nulstil operation
{
    minutter=0;
    sekunder=0;
    mitDisplay->vis(minutter, sekunder);
}

void Ur::taelOp()           // privat taelOp operation
{
    sekunder++;
    if (sekunder >= 60)
    {
        sekunder=0;
        minutter++;
        if (minutter >= 60)
            minutter=0;
    }
}
```

Klasserne KnapPanel, Timer og Display er alle grænsefladeklasser til hardwaren. For hver af disse er der på tilsvarende måde en headerfil og en implementeringsfil. Disse klasser implementeres således, at deres *constructor* operation initialiserer hardwaren.

5.4 Minutur implementeret i C

Her skitseres, hvorledes det objektbaserede design af minuturet kan implementeres i programmeringssproget C.

5.4.1 C implementering af maksimum ét objekt pr. klasse

Først vises den simpleste og mest effektive implementering, der kan anvendes i de tilfælde, hvor der kun er ét objekt af hver klasse.

De private attributter og operationer implementeres her som *static* medlemmer, der i C betyder, at de kun kendes i den fil, hvori de er defineret. I modsætning til C++ skal man i C definere disse private attributter og operationer vha. *static* i kodenfilen (.c) og ikke i headerfilen (.h).

Associationerne er her implementeret implicit ved, at en "C-klasse" kun inkluderer de headerfiler, som den har associationer til på klassediagrammet. Der anvendes således ikke pointere til de andre objekter i denne simple men effektive implementering.

Mange C-compilere understøtter kun filnavne med 8 karakterer, hvorfor det kan være nødvendigt at anvende en forkortet version af klassenavnene fra klassediagrammet til filnavnene.

Utilityklassen *Hovedprogram* implementeres i filen ***HovedPrg.c***:

```
#include "Define.h"
#include "Ur.h"
#include "KnapP.h"      // definerer datatypen KnapHaendelse
#include "Timer.h"
#include "Display.h"

int main()
{
    enum KnapHaendelse knapHaendelse;
    // initiering af de fire "objekter"
    Display_init();
    Timer_init();
    KnapPanel_init();
    Ur_init();

    while ( true )    // Uendelig hovedløkke i programmet
    {
        // polling af knappanel
        if (KnapPanel_checkForTast())
            {                                // task aktiveret
                knapHaendelse= KnapPanel_laesTast();
                if (knapHaendelse == START)
                    Ur_start();
                else if (knapHaendelse == STOP)
                    Ur_stop();
                else if (knapHaendelse == RESET)
                    Ur_reset();
            }
        // polling af timer for timeout
        if (Timer_checkForTimeout())
            Ur_timeout();
    }
    return 0;
} // end main
```

I filen *Define.h* kan man f.eks. definere de globale konstanter og brugerdefinerede datatyper, der anvendes i projektet, hvorfor denne fil inkluderes som den første fil i alle øvrige filer. Her kan man f.eks. vha. *enum* definere en datatype *bool*, der kan antage værdien false eller true. (*enum bool {false=0, true=1};*).

Datatypen KnapHaendelse kan på tilsvarende måde som i C++ eksemplet defineres i filen *KnapP.h* vha. følgende erklæring:

```
enum KnapHaendelse {START=1, STOP=2, RESET=3};
```

Klassen *Ur* specificeres i filen *Ur.h*:

```
// class Ur
// private:
#define TIME1000MS 1000

// public:
extern void Ur_init(); // Ur_init er constructor operationen
extern void Ur_start();
extern void Ur_stop();
extern void Ur_reset();
extern void Ur_timeout();
// end class Ur
```

Bemærk at alle public operationer navngives med klassenavnet efterfulgt af operationsnavnet.

Klassen *Ur* implementeres i filen *Ur.c*, der implementerer tilstandsmaskinen.

```
#include "Ur.h"
// inkludering af de "objekter" hvis operationer kaldes fra Ur
#include "Timer.h"
#include "Display.h"

//***** private (static) attributter *****
static enum TILSTAND {STARTET, STOPPET} tilstand;
static int minutter;
static int sekunder;

//***** private (static) operationer *****
static void nulstil() // privat nulstil operation
{
    minutter=0;
    sekunder=0;
    Display_vis(minutter,sekunder);
}

static void taelOp() // privat taelOp operation
{
    sekunder++;
    if (sekunder >= 60)
    {
        sekunder=0;
        minutter++;
        if (minutter >= 60)
            minutter=0;
    }
}

//***** public operationer *****
void Ur_init() // "constructor" operation
{
    tilstand = STOPPET;
    nulstil();
}
```

```
void Ur_start()           // håndterer hændelsen: start
{
    if (tilstand == STOPPET)
    {
        tilstand= STARTET;
        Timer_start(TIME1000MS);
    }
}

void Ur_stop()            // håndterer hændelsen: stop
{
    if (tilstand == STARTET)
    {
        tilstand= STOPPET;
        Timer_stop();
    }
}

void Ur_reset()           // håndterer hændelsen: reset
{
    if (tilstand == STOPPET)
    {
        nulstil();
    }
}

void Ur_timeout() // håndterer hændelsen: timeout
{
    if (tilstand == STARTET)
    {
        Timer_start(TIME1000MS);
        taelOp();
        Display_vis(minutter,sekunder);
    }
}

//*****
```

5.4.2 C implementering af flere objekter pr. klasse

For nogle klasser vil man også i et C-baseret projekt have brug for, at kunne oprette flere objekter af en given klasse. Princippet for dette vises her med udgangspunkt i klassen Ur.

Objektets datatype defineres vha. en *typedefinition* (*typedef*) og en *C struct* – denne nye type benævnes med samme navn som den tilhørende klasse på klassesdiagrammet.

```
typedef struct Ur_type
{
    int minutter;
    int sekunder;
    enum TILSTAND {STARTET, STOPPET} tilstand;
} Ur;
```

Herefter kan datatypen Ur anvendes til at skabe ”objekter” ud fra som det fremgår af følgende *main()* program.

Hovedprogrammet *main()* i filen **Main.c**, hvor der oprettes to Ur objekter.

```
int main()
{
    enum KnapHaendelse knapHaendelse;

    Ur mitUrNr1, mitUrNr2;      // her oprettes to Ur "objekter"

    Ur_init(&mitUrNr1);         // her initialiseres objekterne
    Ur_init(&mitUrNr2);
    Display_init();
    Timer_init();
    KnapPanel_init();

    Ur_start(&mitUrNr1);        // her kaldes operationen start()
    Ur_stop(&mitUrNr1);
    Ur_reset(&mitUrNr1);
    // etc.
    return 0;
}
```

Klassen *Ur* specificeres i filen **Ur.h**:

```
// class Ur
// private:
#define TIME1000MS 1000

typedef struct Ur_type
{
    int minutter;
    int sekunder;
    enum TILSTAND {STARTET, STOPPET} tilstand;
} Ur;

// public:
extern void Ur_init(Ur* const this); // Ur_init er constructoren
extern void Ur_start(Ur* const this);
extern void Ur_stop(Ur* const this);
extern void Ur_reset(Ur* const this);
extern void Ur_timeout(Ur* const this);
// end class Ur
```

Her vises et par eksempler på implementering af operationerne i filen **Ur.c**:

```
***** static operationer *****

static void nulstil(Ur* const this) // privat nulstil operation
{
    this->minutter= 0;
    this->sekunder= 0;
    Display_vis(this->minutter, this->sekunder);
}
```

```
static void taelOp(Ur* const this) // privat taelOp operation
{
    this->sekunder++;
    if (this->sekunder >= 60)
    {
        this->sekunder=0;
        this->minutter++;
        if (this->minutter >= 60)
            this->minutter=0;
    }
}

//***** public operationer *****
void Ur_init(Ur* const this)           // "constructor" operation
{
    this->tilstand = STOPPET;
    nulstil(this);
}

void Ur_timeout(Ur* const this) // håndterer hændelsen: timeout
{
    if (this->tilstand == STARTET)
    {
        Timer_start(TIME1000MS);
        taelOp(this);
        Display_vis(minutter, sekunder);
    }
}
```

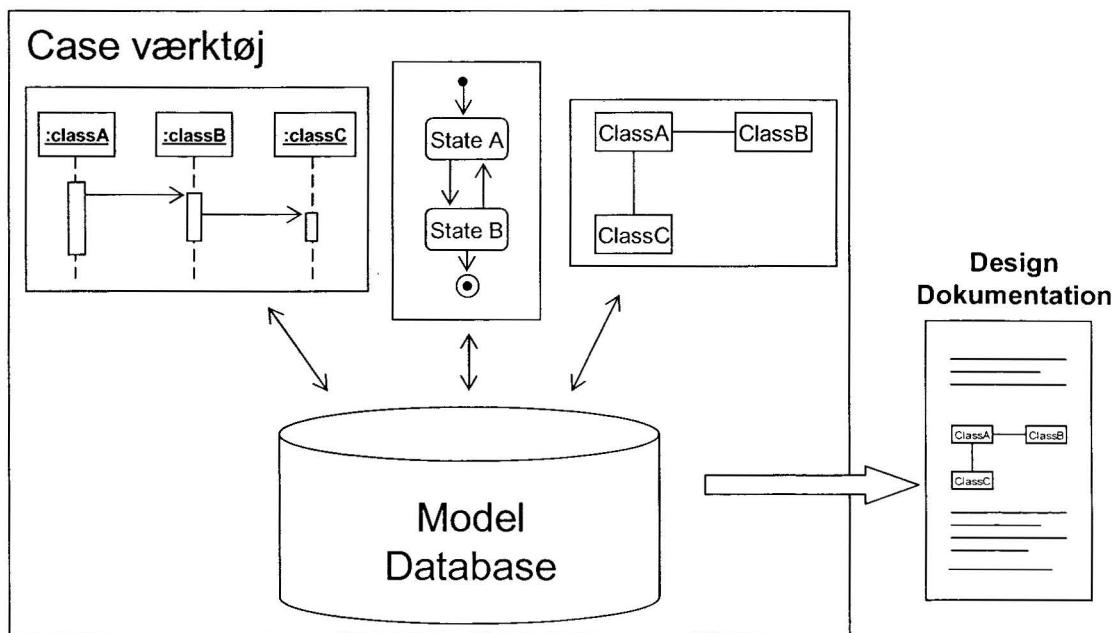
Af disse eksempler ses det, at hver operation i klassen Ur, som første parameter har en pointer, der identificerer det objekt, som operationerne skal udføres på. Dette gælder dog ikke for Display og Timer klasserne, hvor der her kun kan være ét objekt af hver. ”Objektpointeren” kaldes her for *this* som den hedder i C++. Som det ses af f.eks. operationen *taelOp()* så refereres objekts variable vha. denne *this* pointer (*this->sekunder++;*)

6 Værktøjer til UML-Light, UML-Light++ og UML

Værktøjer til UML-Light, UML-Light++ og UML kan groft opdeles i følgende to kategorier hhv. tegneværktøjer og CASE (*Computer Aided Software Engineering*) værktøjer. CASE værktøjer er beregnet til videregående og avanceret UML modellering, hvorfor det til UML-Light kan anbefales at starte med at anvende et simpelt tegneværktøj.

UML-Light diagrammer kan udarbejdes med et hvilket som helst tegneværktøj, f.eks. kan diagrammerne let udarbejdes med Microsofts *Powerpoint* program, hvor man kan linke de enkelte diagramelementer sammen med konnektorer. Mere avancerede tegneprogrammer som f.eks. Microsofts *Visio* program indeholder stencils med de forskellige UML symboler, hvorfor Visio også er meget anvendt til UML diagrammering.

Figur 26 viser, at et CASE værktøj er baseret på en modeldatabase, hvor alle informationer gemmes. Det gælder de UML diagrammer man tegner og de informationer, man definerer for de forskellige diagramelementer. Figuren viser også, hvordan man ud fra et CASE værktøj kan få genereret f.eks. en designdokumentation for sit system.



Figur 26. CASE Værktøj og dokumentation

Ideogramic UML er et dansk udviklet CASE værktøj fra firmaet *Ideogramic*, der er udviklet til brug i forbindelse med UML modellering ved en interaktiv grafisk tavle. Programmet er meget let at anvende og kan også anvendes på en PC. Værktøjet *Ideogramic* egner sig specielt godt til det indledende analyse- og designarbejde med UML-Light, hvor flere arbejder sammen ved en tavle. Eksempler på avancerede og dermed forholdsvis dyre CASE værktøjer er f.eks. *Rhapsody* fra *IBM* og *Artisan Studio* fra *Atego*. Dette er begge værktøjer, der er beregnet specielt modellering af større systemer. Disse værktøjer kræver en del indlæring for at opnå fuld udbytte og anbefales derfor til den komplette UML, hvor de kan tilbyde såvel kodegenerering som simulering af systemet ud fra UML modellerne.

7 UML-Light++ til objekt-orienteret udvikling

UML-Light++ angiver en udvidelse af UML-Light med Use Cases og med notation for generalisering/specialisering og udvidede begreber for associationer. UML-Light++ kan således anvendes til modellering af simple objektorienterede systemer, hvor der anvendes *objektorienteret udvikling* og implementering i f.eks. C++, Java eller C#.

7.1 UML-Light++ og kravspecifikation vha. Use Cases

Use Case teknikken er en teknik og notation, der er integreret i UML og anvendes som teknik ved udarbejdelse af kravspecifikationer. Use Case teknikken anvendes udelukkende til at specificere de såkaldte funktionelle krav i kravspecifikationen. De øvrige, ikke funktionelle krav specificeres på normal måde vha. almindelig tekst. Use Case teknikken kan desuden anvendes i forbindelse med projektstyring samt som udgangspunkt for udarbejdelsen af analyse- og designmødeller og som udgangspunkt for at specificere en accepttest. Use Case teknikken har vundet stor international anerkendelse og er også i Danmark allerede med succes anvendt som specifikationsmetode i et stort antal projekter.

Use Case teknikken kan opdeles i en basal Use Case notation og i en udvidet og mere avanceret notation, der her præsenteres i hvert sit underafsnit.

Anbefalingen er her, at man starter med at anvende den basale Use Case notation og beskriver de funktionelle krav til systemet ved hjælp af denne notation. Derefter kan man overveje, om den udvidede notation vil give en bedre model af kravene og derfor kun anvende denne, hvis dette er tilfældet.

7.1.1 Basal Use Case notation

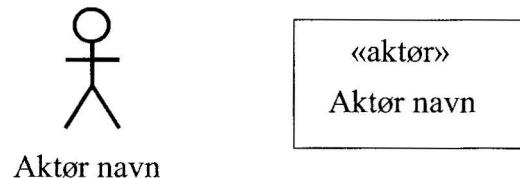
Centrale begreber i Use Case teknikken er begreberne *aktør* og *Use Case*.

Definition:

En **aktør** repræsenterer enten en person, et andet system eller en hardwareenhed.
En aktør er pr. definition udenfor det system, der skal udvikles, men er i samspil med systemet. En aktør beskriver således en grænseflade til det system, man udvikler.
En aktør navngives altid med et navneord i ental.

Til at navngive en personaktør anvendes de roller, personen har overfor systemet. Har en given person flere roller, så optræder hver rolle som en aktør. Dette har stor betydning når man skal finde de tilhørende Use Cases.

En aktør vises som en tændstiksfigur med aktørens navn påført under figuren. Alternativt kan man vælge at vise aktøren som en firkant (en klasse) med stereotypen «aktør».



Figur 27. To alternative notationer for en aktør

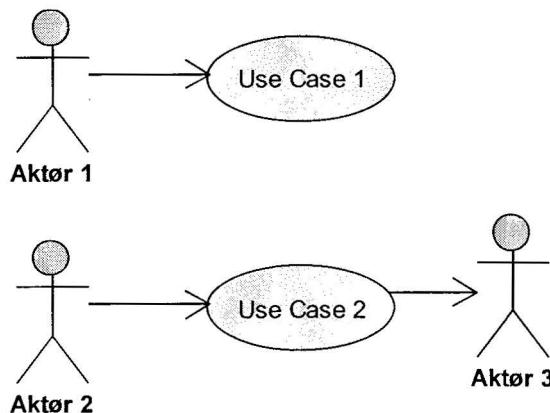
Definition:

En **Use Case** beskriver kravene til en funktionalitet, der udføres af systemet i forhold til en given aktør. En god Use Case skal opfylde et mål for en given aktør eller for systemets kunde. En anden måde at udtrykke dette på er, at kunden, der køber systemet, vil betale for den funktionalitet, som Use Casen beskriver. En Use Case skal beskrive en samlet og afsluttet funktionalitet i forhold til en af systemets aktører.

En Use Case navngives med et handlingsorienteret udsagnsord, der virker på et navneord.

En Use Case vises på et Use Case diagram som en oval med navnet på Use Casen enten i eller under ovalen. Hver Use Case er forbundet til mindst én aktør.

Eksempler på Use Cases for en pengeautomat, er ”*Hæv penge*”, ”*Indsæt penge*” og ”*Se saldo*”, der alle kan udføres af aktøren ”*kunde*”.



Figur 28. Use Case diagram

For hver Use Case udarbejdes en specifikation, der præcist specificerer Use Casens funktionalitet med en beskrivelse af normalforløbet og alle de undtagelser, der kan forekomme. I nogle tilfælde kan beskrivelsen være suppleret med forskellige diagrammer (f.eks. sekvens- eller tilstandsdigrammer). Til at beskrive de enkelte Use Cases anvendes der ofte en skabelon, der består af et antal standardpunkter. Afhængigt af den udviklingssituation som Use Casene anvendes i, anvender man i nogle projekter en meget simpel skabelon, hvor man i andre projekter anvender en mere avanceret skabelon.

Hvordan man specificerer de enkelte Use Cases er ikke beskrevet i UML standarden, men der er i industrien dannet en slags de facto standard for hvordan man beskriver disse. I dette UML-

Light++ afsnit vises der på Figur 29 et eksempel på en simpel og dermed minimal Use Case skabelon.

Use Case navn:

Her vælges et navn der består af et handlingsorienteret udsagnsord, der virker på et navneord. Use Case navnet skal kort opsummere det mål man opnår når Use Casen udføres med succes f.eks. "Hæv Penge".

Mål:

Her uddybes Use Case navnet således at formålet med Use Casen er lydende klart og således virker som en introduktion til Use Casen. Eksempel: Denne Use Case skal håndtere kunders hævning af penge i en pengeautomat.

Normalt scenario:

Her beskrives det normale forløb (scenario) for Use Casen – dvs. der hvor alt udføres uden fejl fra systemets side og hvor en evt. aktør f.eks. en bruger også opfører sig normalt. Scenariet beskrives vha. et antal trin, der trin for trin fører til slutmålet.

Eksempel:

1. Kunden isætter sit dankort
2. Automaten validerer dankortet
[dankortet kan ikke læses]
[spærret dankort]
3. Kunden indtaster sin pinkode
4. Automaten validerer pinkoden
[forkert pinkode]
5. Kunden vælger det beløb der skal hæves
6. Automaten udbetaler beløbet

Undtagelser:

Her beskrives de undtagelser og afvigelser der kan forekomme i forhold til det normale scenario og hvordan systemet skal håndtere disse. Undtagelserne kan f.eks. nummereres således at de henviser til et trin i normalforløbet.

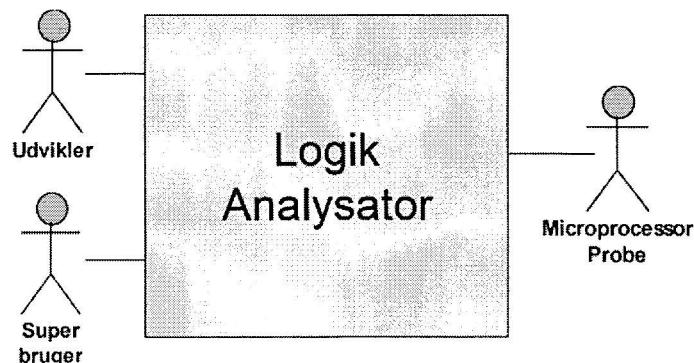
Eksempel:

- 2.A *dankortet kan ikke læses*:
 automaten udskyder kortet og giver besked til kunden om at kortet ikke kan læses.
- 2.B *spærret dankort*:
 automaten udskyder kortet og giver besked til kunden om at kortet er spærret.
4. *forkert pinkode*:
 besked om forkert kode – efter tre forsøg spærres kortet.

Figur 29. Eksempel på en minimal Use Case skabelon

Anvendelse af Use Case teknikken starter med, at man fastlægger konteksten for systemet ved at definere aktørerne i forhold til det system, man er ved at udvikle. Aktørerne og systemet vises på et aktør-kontekstdiagram, der viser systemet i samspil med dets aktører. Figur 30 viser et eksempel bestående af hhv. to personaktører og én hardwareaktør. Ved at navngive og definere de aktører som systemet er i samspil med, får man defineret systemets afgrænsning i forhold til omgi-

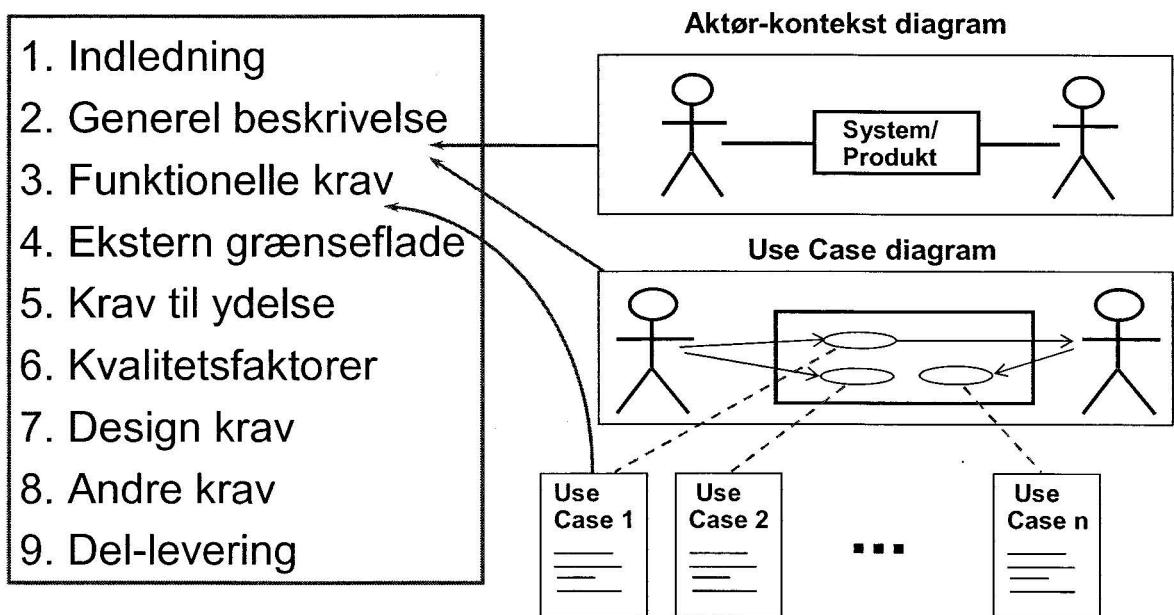
velserne. Når dette diagram er på plads, kan man begynde på at finde de Use Cases, som de enkelte aktører er involveret i.



Figur 30. Eksempel på et aktør-kontekstdiagram (for eksempel 2).

Figur 31 viser, hvorledes aktør-kontekstdiagrammet og Use Case diagrammerne og de tilhørende specifikationer indgår i en standard indholdsfortegnelse for en kravspecifikation [SPU88].

En Use Case baseret kravspecifikation består ligesom den traditionelle kravspecifikation hovedsageligt af tekst, der beskriver de funktionelle krav (kap.3). Det der gør den store forskel, er den teknik og den Use Case drevne synsvinkel, der er anvendt til at finde Use Casene i den Use Case baserede kravspecifikation.

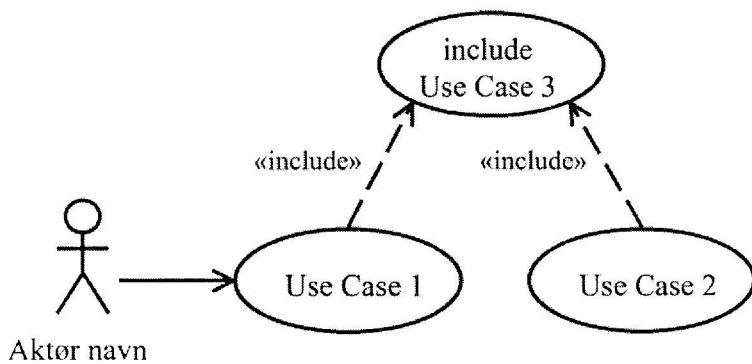


Figur 31. Use Case i relation til et kravspecifikationsdokument

Use Case teknikken kan i øvrigt også anvendes, selv om man ikke efterfølgende vil anvende objektbaserede eller objektorienterede udviklingsmetoder. I dette tilfælde vil man ikke få glæde af de metoder der findes til at omsætte en Use Case baseret specifikation til objektorienterede analyse- og designmodeller.

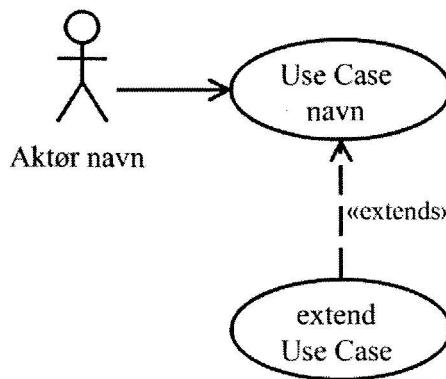
7.1.2 Udvidet Use Case notation

Er der fælles funktionalitet, der indgår i flere Use Cases, beskrives denne funktionalitet for sig selv som en *"include"* Use Case, der er en slags "hjælpe" Use Case. Anvendelsen af *"include"* har den store fordel, at man undgår gentagelser i specifikationen. Notationen for en *"include"* Use Case fremgår af Figur 32.



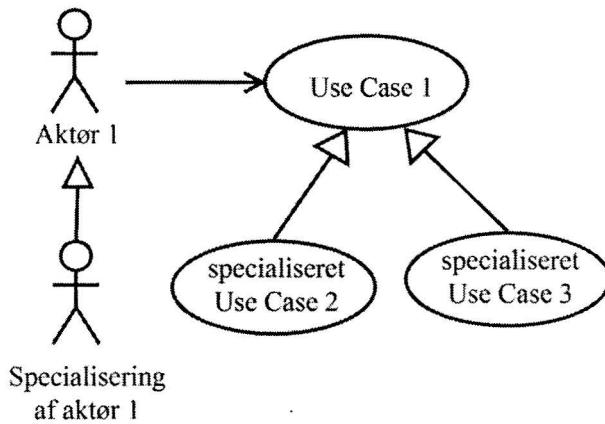
Figur 32. Use Case diagram med *include*

En anden udvidelse til den basale notation er en *"extend"* Use Case, der er en udvidelse til en eksisterende selvstændig Use Case. En *extend* Use Case kan beskrive enten komplekse fejlforløb eller optionale udvidelser, som alle kunder ikke ønsker, eller beskrive funktionalitet, der først ønskes i senere versioner af systemet. Notationen for en *"extend"* Use Case fremgår af Figur 33.



Figur 33. Use Case diagram med *extend*

En tredje udvidelse til den basale notation er en **specialisering** af en Use Case, der beskriver en mere specifik og ofte udvidet funktionalitet i forhold til den Use Case, den er en specialisering af. En sidste udvidelse består i, at også aktører kan specialiseres. Disse udvidelser kan begge ses på Figur 34.



Figur 34. Use Case diagram med specialisering

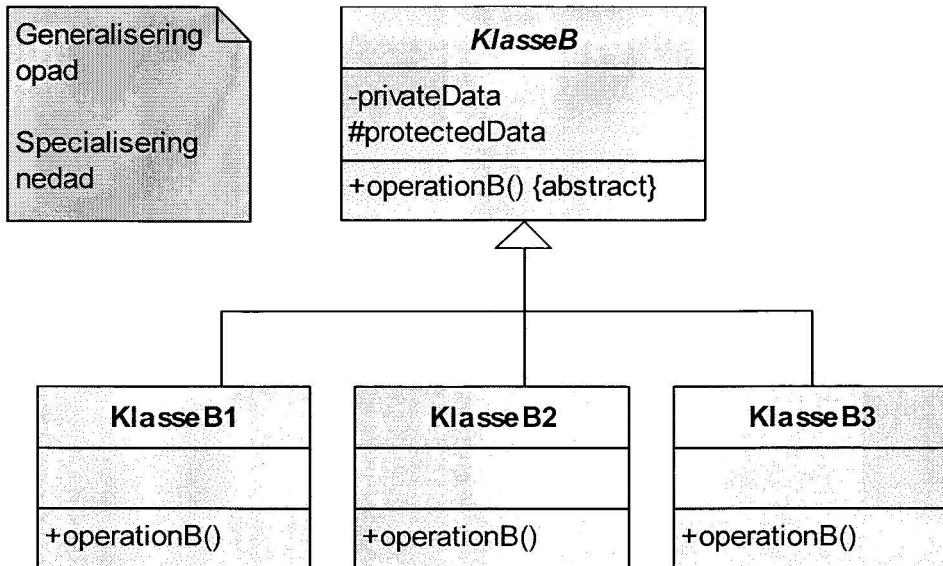
Om Use Case teknikken:

Use Case teknikken er udviklet af svenskeren Ivar Jacobson og først beskrevet i hans bog "Object-Oriented Software Engineering – A Use Case driven approach", Addison-Wesley 1992.

Use Case teknikken er baseret på en simpel grafisk notation. Denne notation indgår i industristandarden UML (Unified Modeling Language), der er en standardnotation for objektorienteret modellering.

7.2 UML-Light++ klassediagrammer med generalisering/specialisering

De væsentligste udvidelser fra objektbaseret udvikling til objektorienteret udvikling er muligheden for at kunne beskrive generalisering og specialisering af klasserne samt polymorfe operationer (=C++ *virtual*). Specialisering implementeres i et objektorienteret programmeringssprog implementeres vha. nedarvning (*inheritance*).



Figur 35. Klassediagram med generalisering/specialisering

Generalisering/specialisering vises på et klassediagram vha. en åben trekant, som det ses af Figur 35. Sammen med dette hører også muligheden for at angive polymorfi, der kan vises enten ved at angive *operationB()* i kursiv og/eller alternativt ved at angive denne som *{abstract}*. Samtidig med dette medtages operationen også på de specialiserede klasser for at vise, at den findes implementeret forskelligt i de forskellige subklasser. Superklassen KlasseB kan også være abstrakt, hvilket vises ved at navnet er i kursiv eller ved at man tilføjer *{abstract}* efter navnet. En klasse der er abstrakt kan ikke instantieres til et objekt, men kan udelukkende anvendes til at nedarve fra. I forbindelse med specialisering optræder der en tredje type af synlighed (*visibility*) nemlig beskyttet (*protected*), der angives med tegnet '#'. Den kan være beskyttede data og beskyttede operationer. Det specielle ved beskyttede data eller operationer er, at de kan tilgås direkte i en subklasse, hvorimod f.eks. private data kun kan ændres af de operationer, der er defineret på den samme klasse.

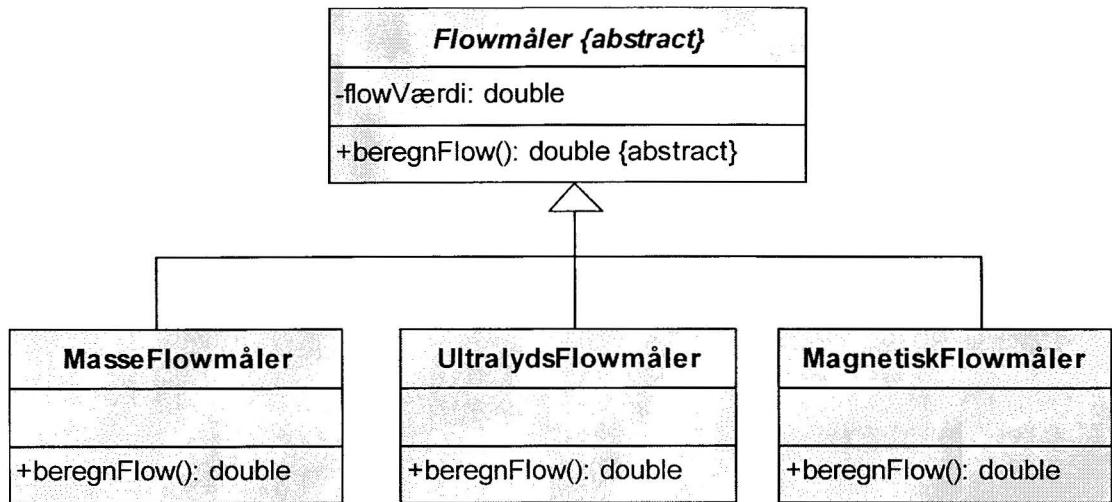
Her vises den tilhørende erklæring af hhv. KlasseB og KlasseB1 i C++.

```
class KlasseB
{
public:
    KlasseB(); // constructor
    virtual void operationB() = 0; // pure virtual
protected:
    int protectedData;

private:
    int privateData;
}
```

```
// her vises subklassen KlasseB1

class KlasseB1: public KlasseB // KlasseB1 nedarver public fra KlasseB
{
public:
    KlasseB1(); // constructor
    void operationB(); // KlasseB1 implementering af operationB
}
```



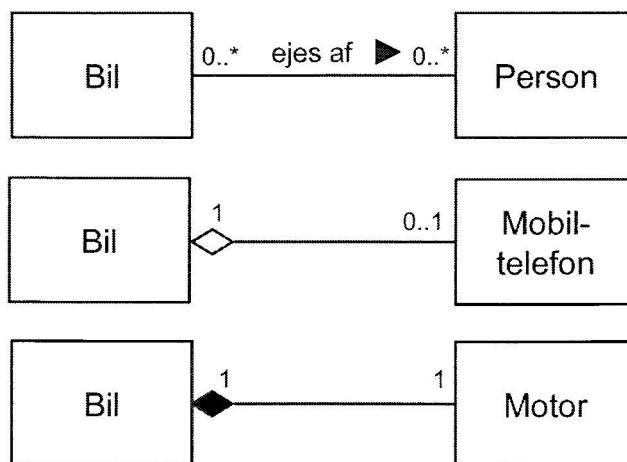
Figur 36. Eksempel på specialiseringer af en flowmåler

7.3 UML-Light++ klassediagrammer med udvidede associationsbegreber

UML-Light introducerede associationer mellem klasser. I *UML-Light++* tilføjes der to nye typer af associationer, der kaldes for aggregering og komposition.

En **aggregering** (*aggregation*) beskriver et forhold mellem klasser, hvor den ene klasse er overordnet i forhold til den anden. En aggregering beskriver således en stærkere binding mellem to klasser end en almindelig association gør. En aggregering benævnes også som en ”har en” (*has a*) relation.

I eksemplet på Figur 37 har en bil en mobiltelefon, dvs. at mobiltelefonen er underlagt bilens opførsel og kan f.eks. få strøm fra denne. Der er her indikeret en stærkere binding mellem disse to klasser end der er mellem en bil og en person.



Figur 37. De tre associations typer

Er der tale om en endnu stærkere binding mellem to klasser, skal man anvende en **komposition** (*composition*), der beskriver den situation, hvor en klasse indgår i realiseringen af en anden. I dette tilfælde lever og dør det indlejrede objekt med det objekt, hvori det indgår. I eksemplet indgår en motor altid i en bil, hvorfor der er et stærkt forhold mellem disse klasser og de tilhørende objekter. Oprettes der et bilobjekt, vil man automatisk også få oprettet et objekt af klassen Motor og tilsvarende vil motorobjektet forsvinde, hvis bilobjektet nedlægges.

Regel:

Husk at det alle er associationer, hvor bindingen mellem objekterne vokser nedad i forhold til Figur 37.

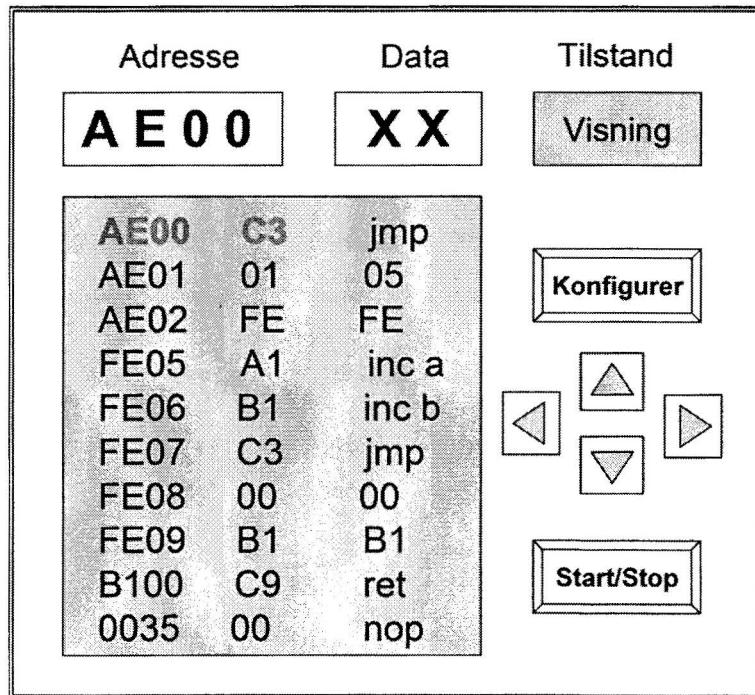
Er man i tvivl om en given associationstype – så gå et niveau op.

8 Eksempel 2: Logikanalysator designet med UML-Light++

8.1 Beskrivelse af eksemplet

Som eksempel 2 vises her modelleringen af softwaren til en logikanalysator. En logikanalysator anvendes til at undersøge eksekveringen af et logisk kredsløb. Her er logikanalysatoren anvendt til at analysere afviklingen af et program på en mikroprocessor. Logikanalysatoren indsættes via et kabel og et specialstik i soklen til microprocessoren, der monteres ovenpå specialstikket. Logikanalysatoren sampler signalerne på hhv. adressebussen og databussen. Logikanalysatoren implementeres som et program på en PC, der får signalerne fra et dedikeret indstikskort. Kortet har et påmonteret kabel med et stik, der passer i den mikroprocessorsokkel, som logikanalysatoren skal teste.

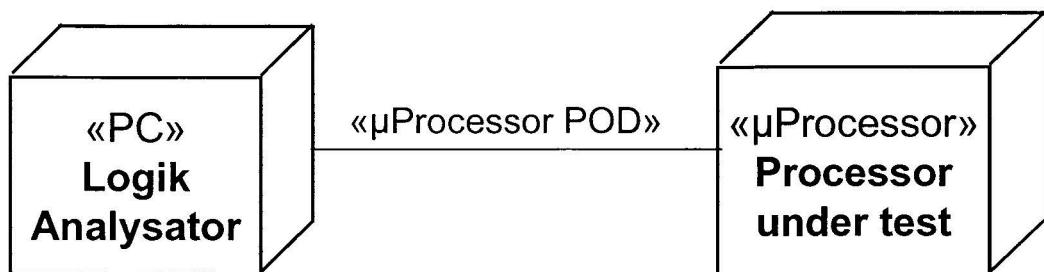
Figur 38 viser en skitse af brugergrænsefladen for logikanalysatoren, der består af to felter, hvor man kan indstille hhv. en trigger-adresse og/eller et trigger-dataord. Derudover er der to kontrolltaster *Konfigurer* og *Start/stop* tasterne samt fire piletasterne.



Figur 38. Skitse af brugergrænsefladen for logikanalysatoren

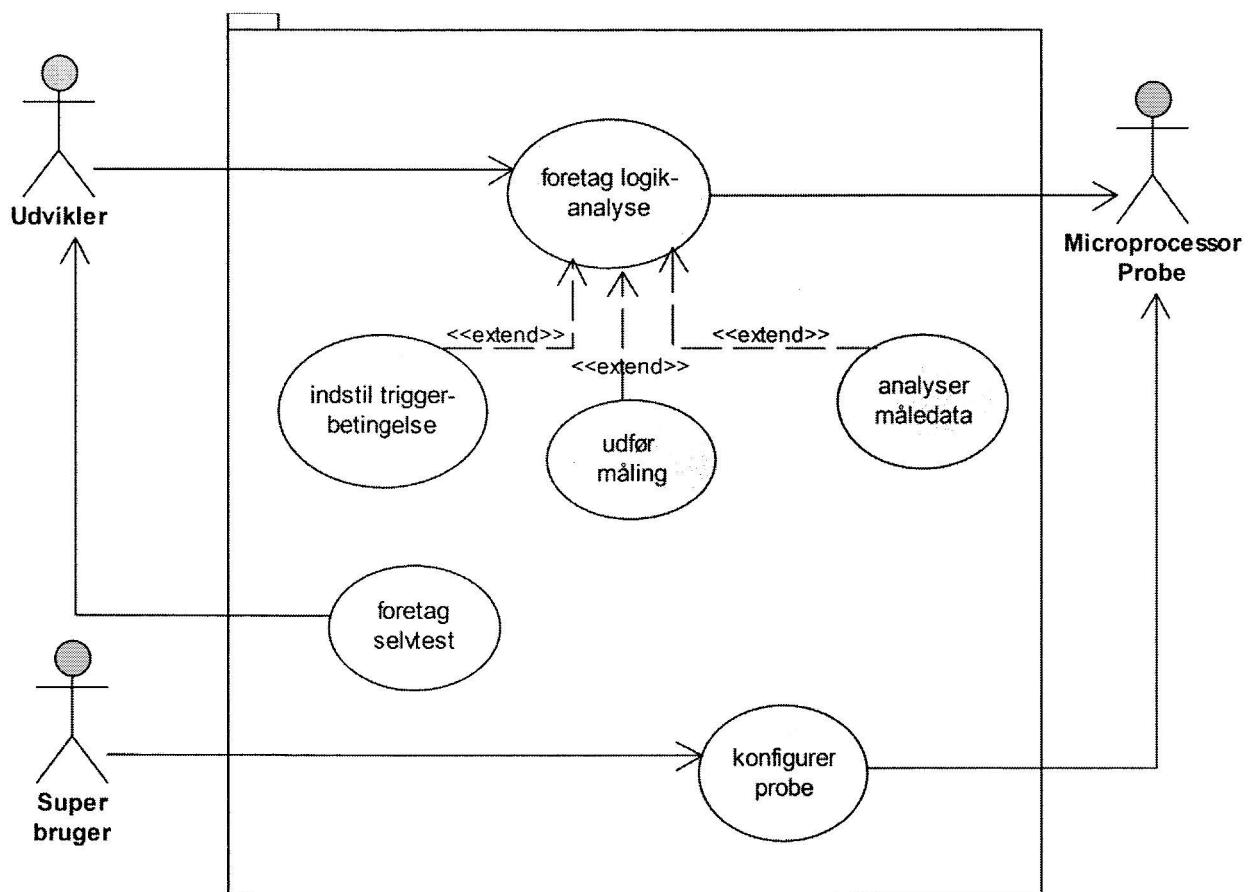
Brugeren af logikanalysatoren kan, efter at have trykket på tasten *Konfigurer*, opsætte en bestemt adresse og/eller et bestemt dataord i hexadecimal notation. Dette gøres vha. piletasterne, hvor højre/venstre piltasterne skifter til det næste felt og op/ned piltasterne scroller mellem 0-F og X, der angiver *don't care*. Når man har opsat adresse- og datafeltet aktiveres tasten *Start/Stop*, hvorefter logikanalysatoren vil begynde at kigge efter adresse og data, der matcher den opsatte trigger betingelse. Hvis denne findes under kørslen af programmet, samples processorens adresse- og databus og indlæses i lageret. Når lageret er fyldt, går logikanalysatoren over i visningstilstanden, hvor brugeren vha. op/ned piltasterne kan scroll frem og tilbage i de opsamlede data. På displayet vises afviklingen af programmet (i det blå felt) først med adressefeltet, dernæst med instruktionerne i hhv. maskinkode og i assemblernotation.

8.2 UML-Light++ model



Figur 39. Hardwarekonfigurationen vist vha. et UML *Deployment*-diagram

De funktionelle krav til logikanalysatoren er beskrevet vha. Use Case teknikken og fremgår af Use Case diagrammet på Figur 40.



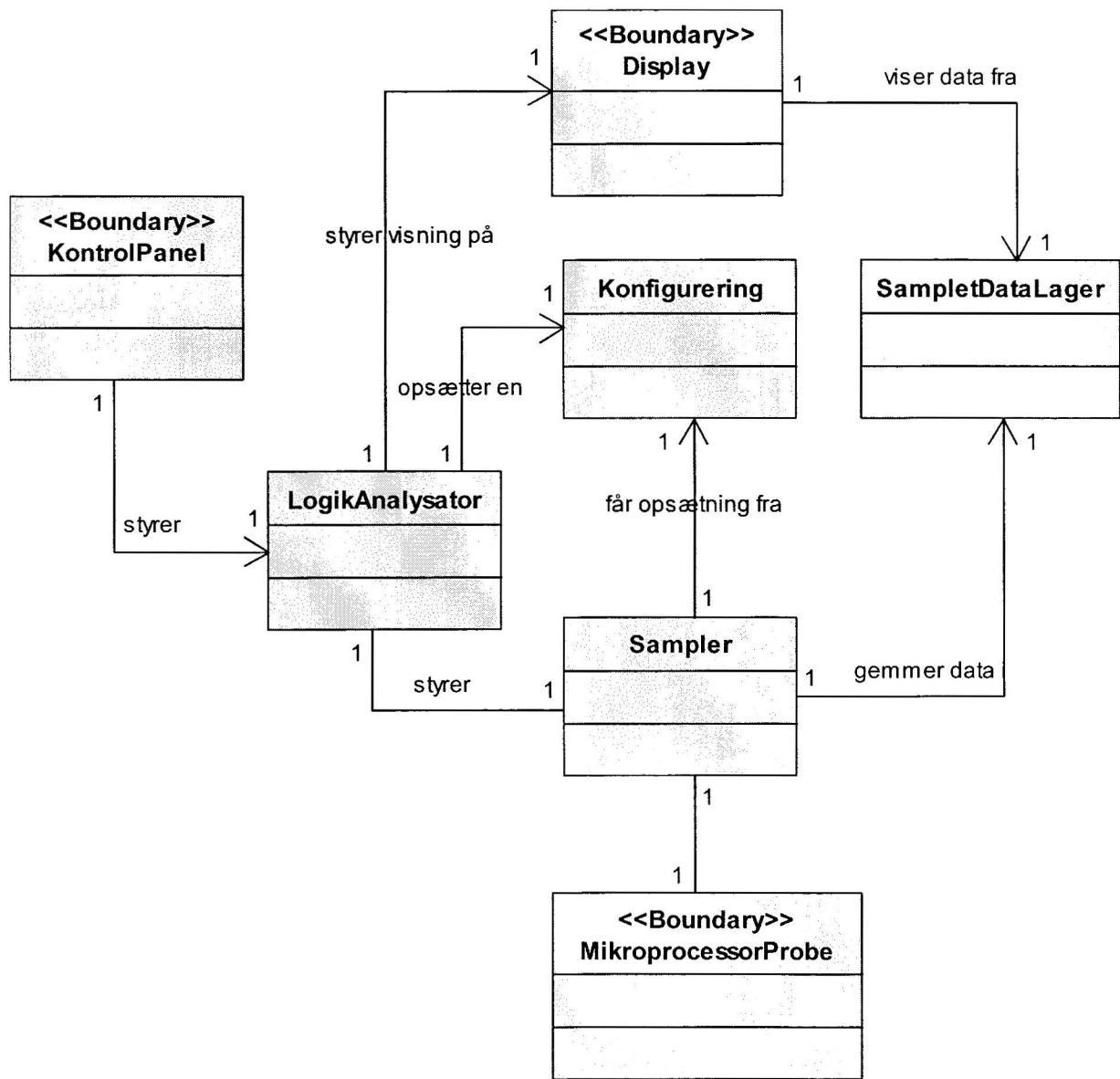
Figur 40. Use Case diagram for logikanalysatoren

Den vigtigste Use Case er her Use Casen "foretag logikanalyse", hvis funktionalitet benyttes af en udvikler ved den normale anvendelse af logikanalysatoren. Derudover indeholder diagrammet Use Casen "foretag selvtest", der tester apparatet ved opstart og melder evt. fejl til brugeren, der her er udvikleren. Derudover viser diagrammet også opsætnings use casen "konfigurer probe", der her benyttes af en aktør med navnet "super bruger". Denne Use Case anvendes kun når logikanalysatoren skal sættes op til at kunne anvendes sammen med en ny type hardware f.eks. en ny type processor.

Med udgangspunkt i Use Case diagrammet kan man efterfølgende udarbejde et eller flere klassediagrammer. Dette gøres ved at udvælge en eller flere af Use Casene og ud fra disses specifikationer tegnes det første udkast til et klassediagram.

I eksemplet vil det være oplagt at tage udgangspunkt i Use Casen ”foretag logikanalyse” og dens extend use cases. Med udgangspunkt i disse Use Cases er klassediagrammet Figur 41 udarbejdet. Diagrammet beskriver designet af den vigtigste del af den software, der kører på LogikAnalysator PC'en. Softwaren tænkes implementeret i C++, hvor der for hver klasse findes hhv. en specifikationsfil (*.h fil) og en kodefil (*.cpp fil).

Stereotypen «Boundary» anvendes her til at angive de klasser, der er grænseflader til aktørerne.



Figur 41. Klassediagram for logikanalysator (uden attributter og operationer)

Klasserne på Figur 41. Klassediagram for logikanalysator (uden attributter og operationer)

Figur 41 har følgende ansvar:

KontrolPanel:

Denne klasse har ansvaret for at håndtere brugerinput til logikanalysatoren.

LogikAnalysator:

Denne klasse har til ansvar at styre logikanalysatorens virkemåde, der er beskrevet vha. en tilknyttet tilstandsmaskine.

Display:

Denne klasse er ansvarlig for at vise informationer til brugeren, det være sig måledata og logikanalysatorens tilstand. Klassen foretager også en disassemblering af processorens maskinkode, således at maskinkoden vises i assemblerkode.

Konfigurering:

Denne klasse er ansvarlig for indtastning af trigger-adresse og trigger-dataord og gemmer denne opsætning under et måleforløb.

SampletDatalager:

Denne klasse er ansvarlig for at lagre de opsamlede data, der måles af logikanalysatorens sampler objekt.

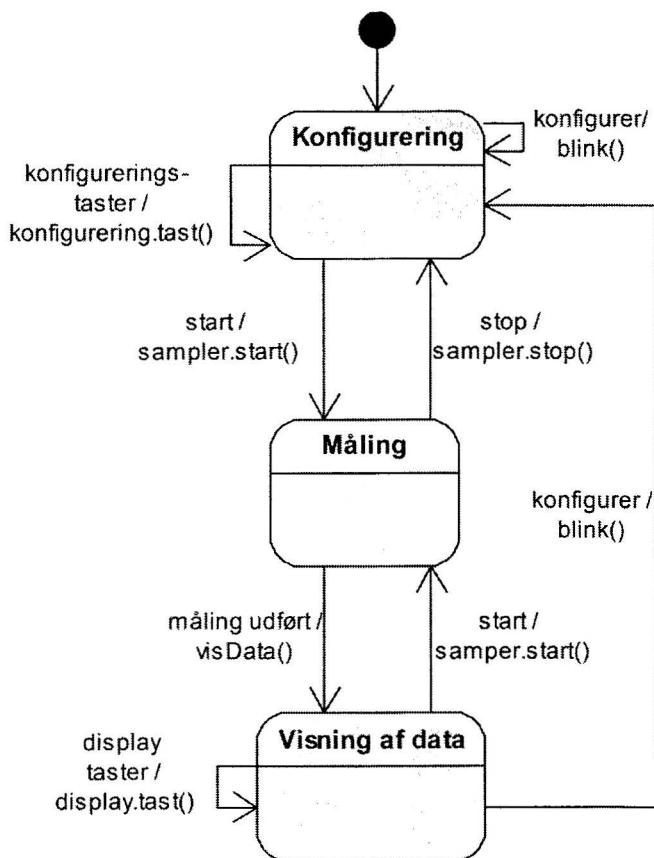
Sampler:

Denne klasse foretager den egentlige opsamling af måledata ud fra den trigger-opsætning, der er opsat via klassen Konfigurering.

MicroprocessorProbe:

Denne klasse repræsenterer den tilkoblede hardwareprobe og har til ansvar at måle de fysiske signaler på den mikroprocessor, der analyseres.

Logikanalysatorprogrammet er et oplagt eksempel på anvendelsen af en tilstandsmaskine til at beskrive den overordnede virkemåde af analysatoren. Tilstandsmaskinen er beskrevet vha. tilstandsdiagrammet på Figur 42, der hører til Logikanalysatorklassen og beskriver denne klasses virkemåde. For at holde diagrammet på et overskueligt niveau er der indført begrebet konfiguringstaster, der her er en samlet betegnelse for tasterne *pilOp*, *pilNed*, *pilHøjre* og *pilVenstre*. Disse taster sendes videre til objektet konfigurering, hvor de håndteres. Tilsvarende er displaytaster en kortere angivelse af tasterne *pilOp* og *pilNed*. Disse taster sendes videre til objektet display, hvor de bevirket, at displayet scrolles hhv. op og ned.



Figur 42. Tilstandsdiagram for logikanalysator

Tilstandsdiagrammet på Figur 42 har følgende tre tilstande, der styrer logikanalysatorens virkemåde.

Konfigurerering:

I denne tilstand kan brugeren opsætte og konfigurere den følgende måling, der udføres af sampleren. Det hexadecimale ciffer, der aktuelt kan ændres, angives ved at det blinker. Der skiftes til næste ciffer vha. enten højre eller venstre piletasterne.

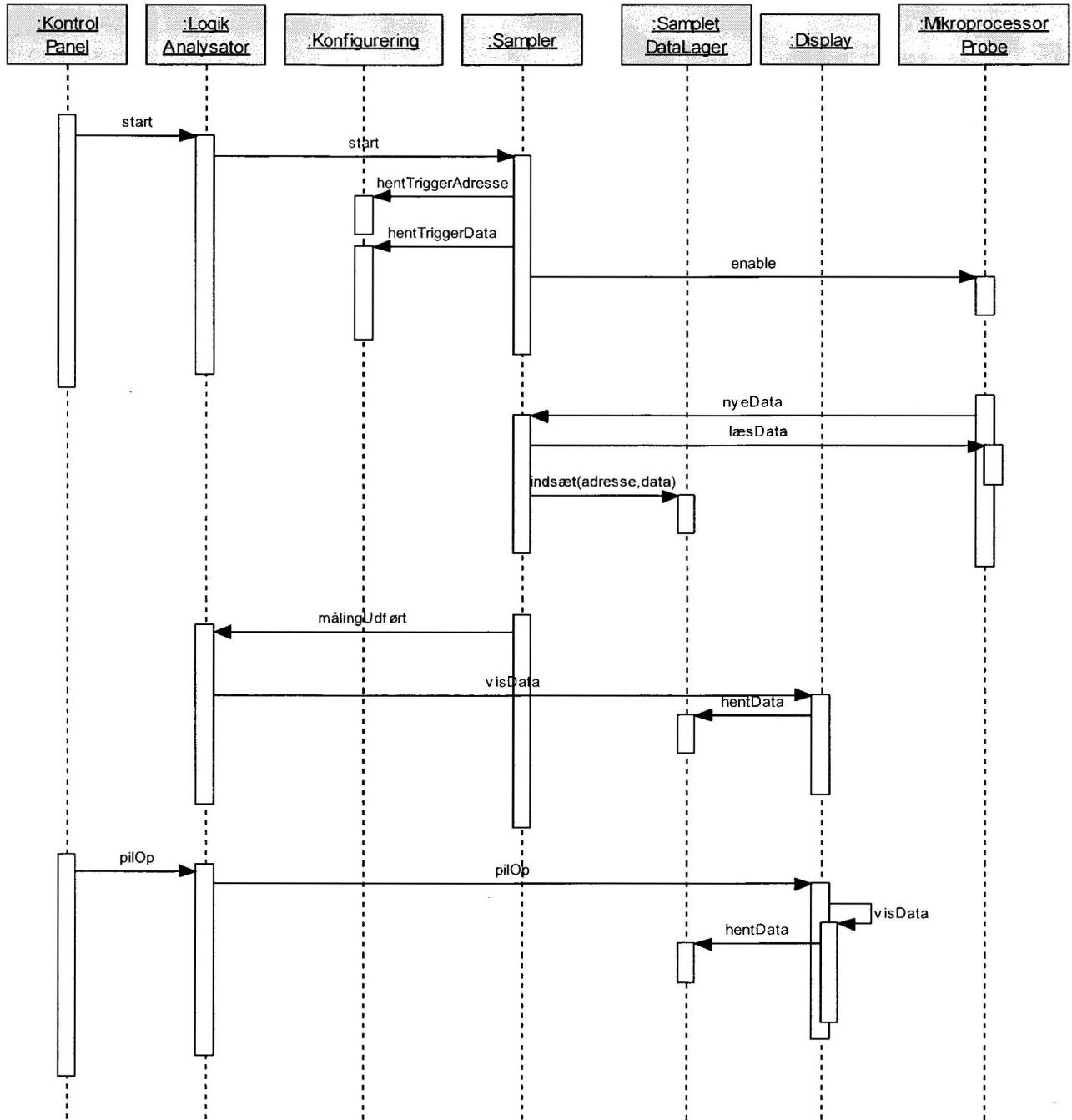
Måling:

I denne tilstand vil logikanalysatoren måle på mikroprocessorens adresse og databus indtil de betingelser, som konfigureringen har defineret, er opfyldt.

Visning af data:

I denne tilstand kan brugeren analysere de data, der er opsamlet. Dette gøres ved at scrolle frem og tilbage i måledataene.

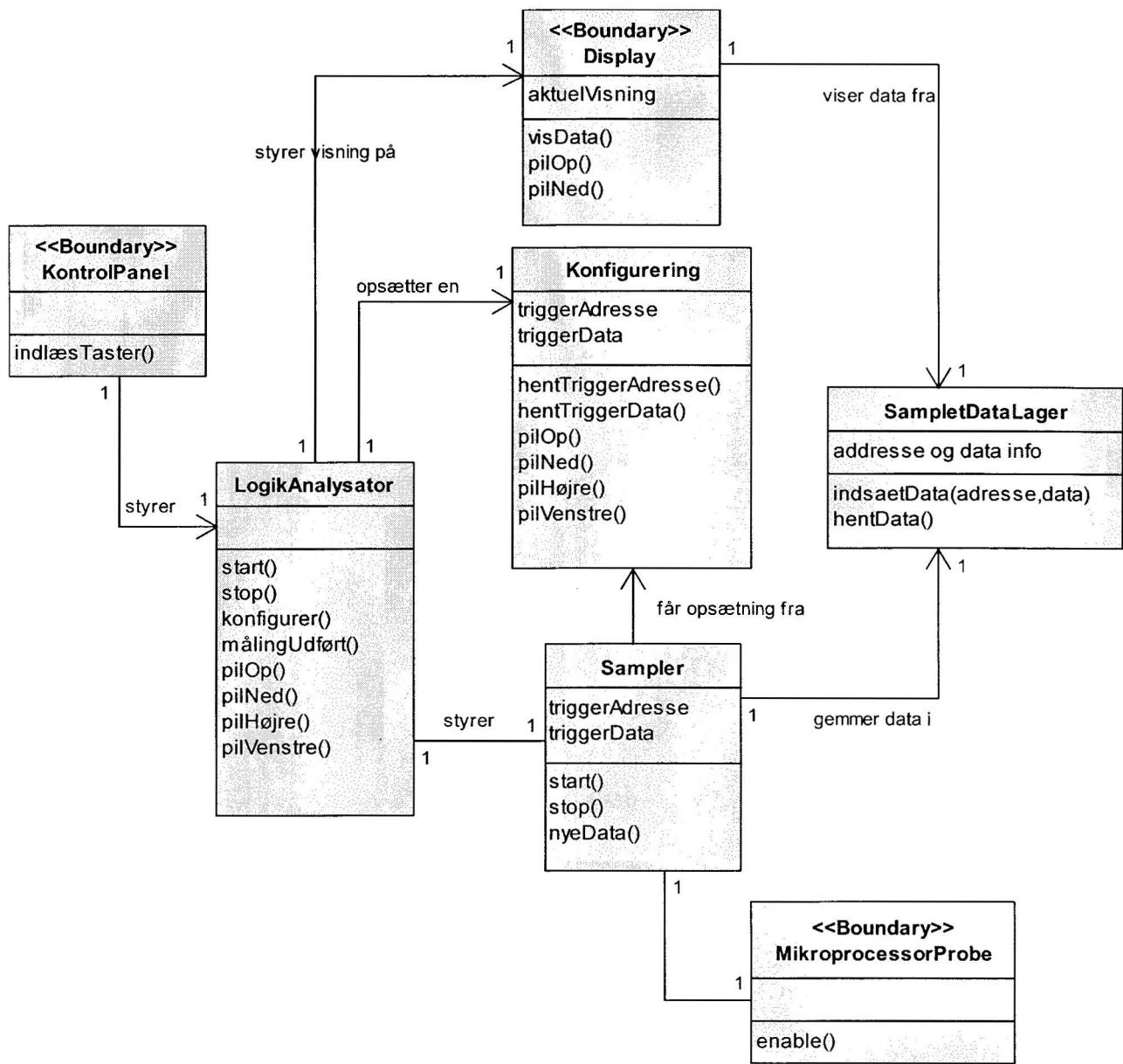
Samspillet mellem de objekter, der instantieres ud fra klassediagrammet, beskrives vha. et UML-Light sekvensdiagram, som vist på Figur 43.



Figur 43. Sekvensdiagram for logikanalysator med flere scenarier

Sekvensdiagrammet viser målescenariet og et enkelt scenario i visningstilstanden, hvor tasten *pilOp* aktiveres. Meddelelsen ”*nyeData*” vil generere et interrupt, hvorefter Sampler objektet indlæser data fra adresse- og databussen og gemmer disse i objektet *:SampletDataLager*. Når lagret er fyldt, genererer Sampler objektet meddelelsen ”*måling udført*”, der sendes til LogikAnalysator objektet, der skifter fra tilstanden ”*måling*” til tilstanden ”*visning*”. Her kan brugeren ved hjælp af tasterne *pilOp* og *pilNed* scrollle i visning af de opsamlede data fra objektet *:SampletData-Lager*.

Efter at have udarbejdet tilstandsdiagram og et eller flere sekvensdiagrammer for logikanalysatoren, kan man nu tilføje de fundne operationer på klassediagrammet fra Figur 41.



Figur 44 Klassediagram for logikanalysator (med attributter og operationer)

9 Referencer

[Fowler2003] "UML Distilled", Third Edition,

Martin Fowler, Addison-Wesley 2003.

Martin Fowlers "UML Distilled" regnes for at være den bedste kortfattede introduktion til UML.

[Jacobson92] "Object-Oriented Software Engineering",

Ivar Jacobson, Addison-Wesley 1992.

Den første egentlige beskrivelse af Use Case teknikken. Bogen er samtidig også en Software Engineering bog, med en grundig indføring i en OO metode.

[Parnas72] "On the Criteria to be used in Decomposing Systems into Modules"

D.L. Parnas, Communication of the ACM, dec. 1972 pp. 1053-1058.

[OMG] "Object Management Group", home page: www.omg.org

OMG står bag UML standarden, CORBA m.fl. Tilgået juni 2013.

[SPU88] "Håndbog i Struktureret Programudvikling",

Stephen-Biering Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen
Teknisk Forlag 1988.

[SPU-UML2003] "SPU - UML note, Systematisk Program-Udvikling" (note i PDF format)

Finn Overgaard Hansen, Ingeniørhøjskolen i Århus, august 2003.

<http://staff.ih.a.dk/foh/Foredrag/SPU-UML.pdf>

Opdateringsnote til "Håndbog i Struktureret ProgramUdvikling (SPU)" med fokus på anvendelse af UML i forbindelse med objektorienteret softwareudvikling.

[SysML2012] "OMG System Modeling Language (SysML)". <http://www.omg.org/spec/SysML/>

Nyeste version er ver. 1.3 (Juni 2012). Tilgået juni 2013.

[UML2011] "Unified Modeling Language", se www.uml.org eller www.omg.org/uml/

Nyeste version af UML er UML 2.4.1 (August 2011). Tilgået juni 2013.

Tools:

Artisan Studio: UML og SysML based CASE værktøj fra firmaet Atego

<http://www.atego.com/products/artisan-studio/>, Tilgået juni 2013.

Rhapsody: UML og SysML based CASE værktøj fra IBM, Tilgået juni 2013.

<http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/>

Ideogramic UML: CASE værktøj fra firmaet Ideogramic (NB! Firmaet eksisterer ikke mere).

Appendix:

Anvendelse af UML-Light, UML-Light++ og UML på Ingenørhøjskolen Aarhus Universitet

På 1. semester introduceres og anvendes **UML-Light** dels i forbindelse med programmeringsundervisningen og dels i forbindelse med semesterprojektet.

UML-Light er den delmængde af UML, der er beskrevet i kapitel 1-5 af denne note.

UML-Light omfatter derfor *ikke* følgende UML begreber og diagramtyper:

- nedarvning og polymorfi
- mange multiplicitet
- aggregering og komposition
- UMLs pakkebegreb
- UMLs hierarkiske tilstandsdiagrammer og tilhørende avancerede notationer
- UMLs komponentdiagrammer
- UMLs Use case diagrammer

På 2. semester anvendes **UML-Light++**, dvs. at klassediagrammerne udvides med notation til at vise generalisering/specialisering (nedarvning) og polymorfi. UMLs Use Case begreb anvendes i forbindelse med udarbejdelse af kravspecifikationer og introduceres som teknik i forbindelse med kurset introduktion til SysML og semesterprojektet på andet semester. Anvendelsen af UML følges op på 3. semester i de software relaterede fag og ved softwareudviklingen i semesterprojekterne.

På 4. semester introduceres alle elementer i **UML** i kurset software design (IKT) samtidig med, at der undervises i en metodisk fremgangsmåde dvs. en **udviklingsproces** for objektorienteret software udvikling. Her vil bogen [Fowler2003] være et godt supplement for den komplette UML notation og anvendelsen af de objektorienterede begreber sammen med metodeovervejelser og anvendelse af designmønstre.

Efter dette forløb er der skabt et solidt fundament således, at man senere kan udvide sin UML erfaring. Dette kan f.eks. foregå i forbindelse med semesterprojektet på fjerde semester, tilvalgskurser på de efterfølgende semestre, i praktikperioden og ved bachelorprojektet. Endvidere anvendes UML og OO også i forbindelse med undervisningen og udvikling af konkrete indlejrede og distribuerede realtidssystemer på civilingeniøruddannelsen i teknisk IT.

Der er på Ingenørhøjskolen mulighed for at anvende følgende UML baserede værktøjer (se kapitel 6.):

Ideogramic, Visio og Rhapsody.

Da *Rhapsody* er et avanceret CASE værktøj, anbefales dette først anvendt i forbindelse med semesterprojektet på 4. semester og fremefter.

Rhapsodys mulighed for kodegenerering og simulering på UML modelniveau anbefales først anvendt senere i studiet, dvs. efter at man på 4. semester har lært at implementere UML modellerne manuelt i et objektorienteret programmeringssprog som f.eks. C++.