



Objektorienteret analyse og design med UML og Unified Process

Hanne Sommer 2020



Indhold

1	Introduktion til systemudvikling.....	5
2	Analyse	10
2.1	Krav og use cases	10
2.1.1	Krav.....	10
2.1.2	Use cases	13
2.1.3	Use case diagram	14
2.1.4	Use case beskrivelser	16
2.1.5	Krydstabel	19
2.2	Klasser.....	20
2.2.1	Finde klasser	20
2.3	Sammenhænge mellem klasser	21
2.3.1	Statiske sammenhænge	21
2.3.2	Dynamiske sammenhænge.....	23
2.3.3	Analyseklassediagram.....	26
2.3.4	Opgaver.....	27
2.4	Analysemønstre	28
2.4.1	Rollemønstret	29
2.4.2	Genstand-beskrivelsesmønstret	31
2.4.3	Relateringsmønstret.....	32
2.4.4	Hierarkimønstret	34
2.4.5	Samlingsmønstret.....	36
2.5	Klassers adfærd	38
2.6	Sekvensdiagrammer	40
3	Design	43
3.1	Designkriterier.....	43
3.2	Arkitektur	44
3.2.1	GUI (Graphical User Interface)-laget	45
3.2.2	Applikationslaget	46
3.2.3	Storagelaget.....	47
3.3	Designklasser	47
3.4	Designsekvensdiagrammer.....	53



3.5	Opgaver	56
4	Unified Process	57
4.1	Faserne	62
4.1.1	Mål for Inception	63
4.1.2	Mål for Elaboration	63
4.1.3	Mål for Construction	63
4.1.4	Mål for Transition.....	64
4.2	Disciplinerne	64
4.2.1	Business Modeling.....	64
4.2.2	Requirements	65
4.2.3	Analysis & Design	66
4.2.4	Implementation	67
4.2.5	Test	67
4.2.6	Deployment.....	68
4.3	Anvendelse af Unified Process.....	69
4.3.1	Overordnet projektplan	69
4.3.2	Planlægning af iterationerne – detaljeret projektplan	70
4.3.3	Planer og værktøjer	76
5	Test.....	79
5.1	Indledning og formål med test	79
5.2	Test generelt	79
5.3	Planlægning af test	81
5.4	Testcases som centralt element i en test.....	84
5.4.1	Testteknikker.....	84
5.5	Testniveauer	93
5.5.1	Test på unittest niveau	93
5.5.2	Test på integrationstest niveau.....	93
5.5.3	Test på systemtestniveau.....	95
5.5.4	Brugertest.....	98
5.5.5	Opsummering mht. testniveauer	99
5.6	Test og OO	100
6	Begrebsliste.....	101



7	Referencer og supplerende litteratur	106
---	--	-----

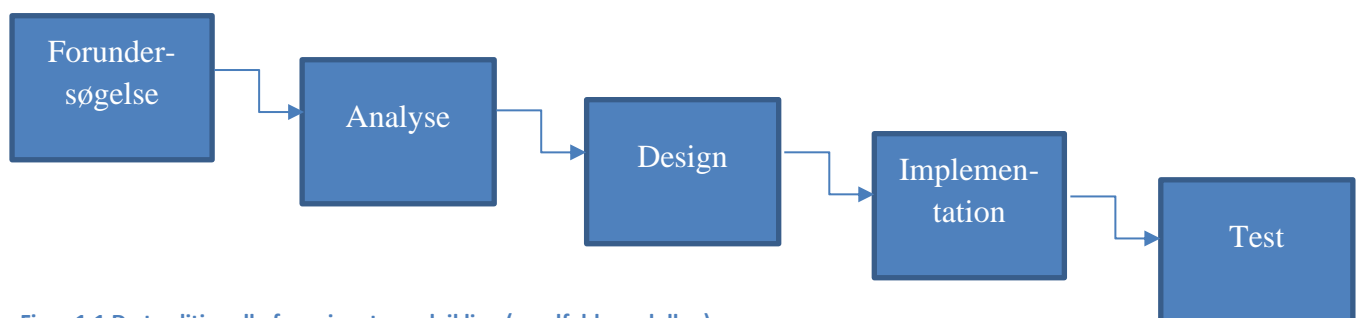
1 Introduktion til systemudvikling

Udvikling af computerbaserede informationssystemer, også kaldet IT-systemer, er en udfordrende opgave med en række komplekse aktiviteter. Et computerbaseret informationssystem kan betragtes som en mængde af elementer og sammenhænge mellem disse. Elementerne kan være programmet, materiellet, menneskene, databasen, dokumentationen, arbejdsgangene, uddannelsen osv.

Både udvikleren og brugerne udfordres ofte med hensyn til at forstå og udvikle hinandens verdener. Ved udvikling af systemer skal vi være opmærksomme på, at vi ofte også er involveret i en organisatorisk forandring, som kan have store konsekvenser for brugernes arbejds-situation. Der er altså nogle menneskelige faktorer, der ikke må glemmes, når vi sammen udvikler systemer. Vi er som regel ikke i stand til fra dag ét at forstå alle problemer og lægge os fast på alle de rigtige løsninger. Forståelsen og de rigtige løsninger kommer typisk løbende, efterhånden som vi i samarbejde med brugerne arbejder os igennem aktiviteterne i udviklingsprocesserne.

Systemudvikleren er en nøgleperson, der analyserer organisationen, identificerer problemer og muligheder for forbedring. Desuden designer og implementerer systemudvikleren IT-systemer til realisering af idéerne til forbedringer. Systemudvikleren laver systemudvikling ved at gennemføre en række udviklingsaktiviteter. Udviklingsaktiviteterne er typisk pakket ind i en form for faser. Det er dog forskelligt fra metode til metode, om det hedder faser, hvordan udviklingsaktiviteterne er pakket ind i faserne osv. I den oprindelige måde at udvikle systemer på gennemløb man udviklingsaktiviteterne en fase ad gangen jf.

Figur 1.1 nedenfor, hvor ideen var, at man afsluttede én fase, før man gik i gang med den næste fase.



Figur 1.1 De traditionelle faser i systemudvikling (vandfaldsmodellen)

Denne systemudviklingsmåde betegnes som **vandfaldsmodellen**, og den kunne fint bruges til de relativt små systemer, der blev udviklet i 1960'erne og frem til 1980'erne. Men efterhånden blev de systemer, der skulle udvikles, så omfattende og komplicerede, at det i praksis var umuligt at forudsætte, at én fase var fuldstændig færdig, inden man begyndte på den næste.

Derfor er man i stedet gået over til at variere denne model, så de forskellige faser gennemføres flere gange eller mere eller mindre parallelt. Dette vil vi gå i dybden med, når vi gennemgår systemudviklingsmetoden Unified Process i et senere kapitel.

Den nævnte faser er nyttige til at forstå noget om de aktiviteter, der skal udføres, og herunder præsenteres de fem faser.

- **I forundersøgelsen** undersøger man et område af virkeligheden, hvor der er oplevet en eller anden form for problemer, eller hvor man har en idé om, at man kan forbedre situationen ved at anvende en (ny/forbedret) IT-løsning. I forundersøgelsen analyserer man først den nuværende situation med henblik på at finde frem til og forstå de aktuelle problemstillinger eller muligheder. Derpå opstiller man en eller flere visioner for, hvordan man ønsker, den fremtidige situation skal være. Endelig skal det besluttes, hvilken vision man vil have virkeliggjort, og hvis den valgte løsning medfører IT-systemudvikling, er man klar til næste fase.¹
- **Analysen** handler om at undersøge den del af virkeligheden, som den valgte vision er baseret på. Det er den del af virkeligheden, hvori problemerne er konstateret, eller hvorom forbedringerne handler, og den kaldes **problemområdet** eller domænet. I analysen er formålet at finde frem til, **hvad** systemet skal indeholde, dvs. hvilke data, der skal registreres og **hvad** anvenderen af systemet skal kunne bruge systemet til. Analysen resulterer i en række beskrivelser, herunder en detaljeret beskrivelse af kravene til det kommende system. Beskrivelserne er basis for designet.
- **Design** handler om, at man på baggrund af analysen finder frem til, **hvordan** systemet skal implementeres. Det drejer sig om både den interne opbygning af systemet og om systemets yder, specielt brugergrænsefladen. For den interne opbygning beslutter man, hvordan systemet mest hensigtsmæssigt opdeles. Systemer opdeles ofte i et antal komponenter, som har hver deres ansvar, fx en komponent, der realiserer grænsefladen. Dette gør selve implementationen mere overskuelig.
- **Implementation** handler om at få programmeret systemet. Programmeringen foretages med udgangspunkt i de beskrivelser, der er lavet i design-fasen.
- **Test** handler om at få verificeret, om systemet lever op til de krav og beskrivelser, der er udarbejdet i analyse og design.

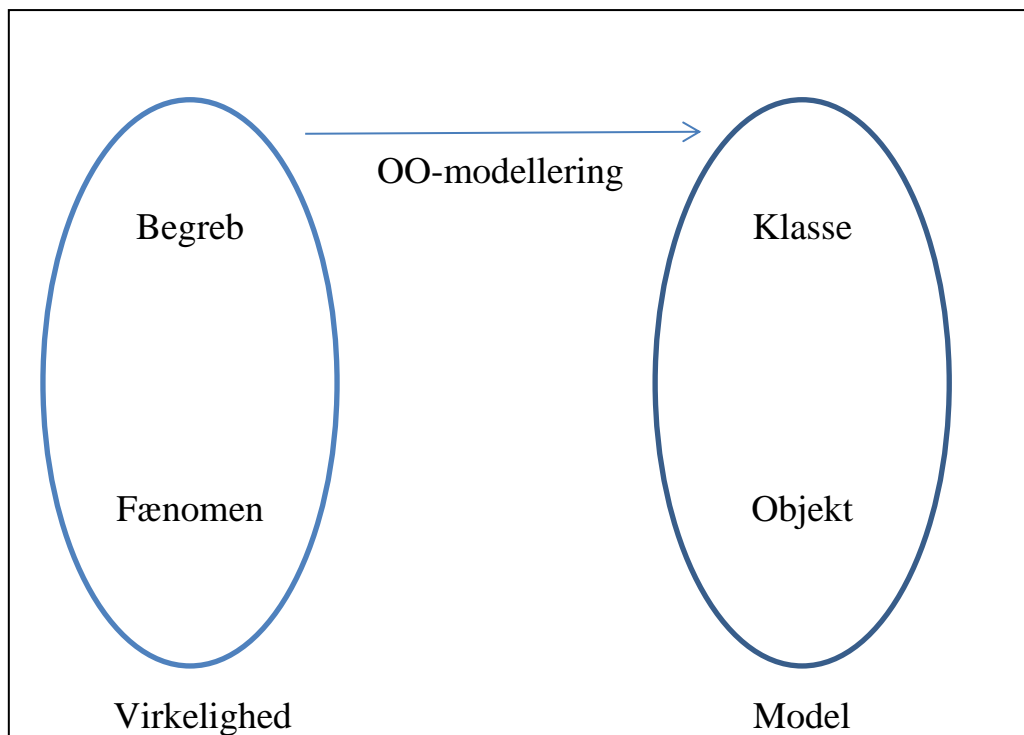
Dette dokument fokuserer på analyse- og designfaserne. Hvordan man udfører forundersøgelse, implementation og test ligger uden for dokumentets rammer, og det gør også tværgående aktiviteter, der ikke er nævnt herover (fx projektstyring). I første del af dokumentet koncentrerer vi os om at beskrive, hvilke aktiviteter der gennemføres samt hvilke produkter, der udarbejdes i analyse- og design-fasen. Der vil være fokus på enkelte teknikker og modeller,

¹ Det kan forekomme, at en vision ikke omfatter udvikling af en IT-løsning, men fx går på en omorganisering af arbejdsgange. I så fald bliver der naturligvis ikke brug for de næste faser.

der udarbejdes, i mindre grad sammenhængen mellem dem. Det overordnede forløb i et systemudviklingsprojekt beskrives i Figur 1.1.

I dette dokument vil vi præsentere analyse og design med den objektorienterede tilgang. Den objektorienterede tilgang til systemudvikling ser på informationssystemer som en samling af interagerende objekter, som arbejder sammen om at gennemføre en opgave. Objektorienteret analyse handler derfor om at finde identificere og beskrive objekter i problemområdet. Gennem analysen af problemområdet beslutter man sig for de fænomener og begreber i virkeligheden, som man ønsker at registrere i systemet. Fænomenerne og begreberne beskrives som objekter. Derudover analyserer man sig frem til, hvad man skal kunne med systemet og objekterne for at gennemføre en opgave. De fænomener og begreber, der ønskes med i systemet, samt egenskaber ved disse udarbejdes der en objektorienteret model af. I objektorienteret design omsætter man den objektorienterede analysemodel til en computer-rettet beskrivelse, og der tilføjes en lang række objekter, der skal til for at implementere systemet. Det er centralt at få beskrevet, hvordan alle objekterne interagerer med hinanden, med brugerne eller andre systemer for at gennemføre opgaverne.

Modellen i Figur 1.2 siger netop en **del** om, hvad objektorienteret systemudvikling går ud på, nemlig at identificere og udvælge fænomener og begreber fra den virkelige verden og lave en objektorienteret model af disse.



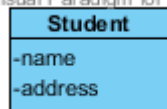
Figur 1.2 Model for objektorienteret modellering

For at forstå modellen er det nødvendigt at forstå nøglebegreberne i objektorienteret systemudvikling. Vi har tidligere skrevet, at man med den objektorienterede tilgang ser på informa-

tionssystemer som en samling af interagerende objekter, som arbejder sammen om at gennemføre en opgave. Et objekt i et computerbaseret system modsvarer et fænomen i den virkelige verden. I den virkelige verden er der ofte mange fænomener af samme type. Hvis problemområdet f.eks. er administration på en videregående uddannelse, er der mange konkrete studerende. Det er som regel muligt at finde et begreb, som dækker over de mange fænomener af samme type. I computersystemet laver man en beskrivelse af objekterne af samme type, dvs. beskrivelsen modsvarer begrebet i den virkelige verden. Beskrivelsen beskriver objektets egenskaber og kaldes en klasse. De mange konkrete studerende i problemområdet bliver i computersystemet til mange studerende-objekter, som er skabt på baggrund af klassen.

En **klasse** er en model af et begreb fra den virkelige verden og tegnes i Unified Modeling Language (UML), som det ses nedenfor. UML er et internationalt standard diagrammeringsteknik til at visualisere resultaterne af analyse og design.

Visual Paradigm for UML

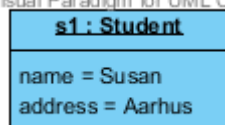


En klasse defineres på følgende måde:

- En klasse er **en beskrivelse af en samling af objekter**, der har de samme egenskaber/karakteristika
- En klasse definerer hvilke **egenskaber**, som hører til hvert eneste objekt af klassen. Egenskaberne beskrives ved hjælp af en række **attributter**, her f.eks. navn, adresse.
- En klasse **definerer adfærden** af de beskrevne objekter i form af **metoder** eller **operationer** (disse er dog her ikke tegnet med i ovenstående UML-modellen af klassen)

Et **objekt** er en model af et konkret fænomen fra den virkelige verden og tegnes i UML på følgende måde:

Visual Paradigm for UML Com



Et objekt defineres på følgende måde:

- Et objekt har en **identitet**, dvs. en **egenskab**, der adskiller objektet fra alle andre objekter. I virkelighedens verden er det typisk nemt at skelne to objekter fra hinanden – i mange tilfælde kan man se dem. Er der tale om et computerobjekt er man nødt til at bruge værdien af en af attributterne, f.eks. et navn, en ID eller lignende.

- Et objekt har en **tilstand**, som er værdien af attributterne på et givet tidspunkt
- Et objekt har en konkret **adfærd**, hvilket er det forløb af anvendelsen af **metoderne** eller **operationerne**, som netop det objekt gennemløber. Metoderne eller operationerne vil enten anvende eller ændre på objektets tilstand (værdierne af attributterne).

2 Analyse

Analyse handler som nævnt ovenfor om at undersøge den del af virkeligheden, der kaldes problemområdet (eller domænet) i forhold til den vision der er valgt. Man detaljerer visionen, hvorved man blandt andet finder frem til en afgrænsning af systemet, dvs. hvad skal med i systemet, og hvad skal ikke med i systemet. Dette kan beskrives på to måder: Hvad skal brugerne **kunne** med systemet, og hvad skal systemet **indeholde**.

Først ser vi på, hvad man skal kunne med systemet. Dette beskrives overordnet set ved hjælp af krav, som udviklerne skal aftale med systemets rekvirent (typisk med kommende brugere). Alternativt beskriver udviklerne det samspil, der skal være mellem brugere og systemet, i form af såkaldte use cases.

Når man i den objektorienterede analyse skal finde ud af, hvad systemet skal indeholde, beskriver man som nævnt ovenfor dette ved hjælp af klasser og objekter samt egenskaber ved disse. Klasserne findes direkte i det problemområde, man arbejder med. En meget vigtig opgave er også at beskrive sammenhængen mellem klasserne, hvilket gøres i et klassesdiagram.

I dette kapitel ser vi nærmere på disse emner og præsenterer teknikker og modeller til at beskrive resultatet af analysen med.

2.1 Krav og use cases

I dette afsnit beskæftiger vi os med teknikker til at modellere, hvad man skal kunne med systemet. Der er grundlæggende to teknikker, der kan anvendes hertil, og de baserer sig på henholdsvis **krav** til systemet og **use cases**. Vores anbefaling er, at man bruger begge teknikker, da de giver to forskellige synsvinkler på brugen, hvorved der er større chance for, at man får det hele med. Det skal dog understreges, at det kun yderst sjældent på én gang kan lade sig gøre at få beskrevet *alt* det, som systemet i den sidste ende skal kunne. Derfor kan man blive nødt til at vende tilbage til henholdsvis krav og use cases.

2.1.1 Krav

Når vi ovenfor har nævnt, at kravene beskriver, hvad man skal kunne gøre med systemet, er det kun en del af sandheden. Den type krav er de funktionelle krav. Der findes også en anden type krav, som primært handler om kvalitetsegenskaber ved det system, der skal udvikles. De kaldes ikke-funktionelle krav eller kvalitetskrav.

- **Funktionelle** krav er udtryk for det, man skal kunne gøre med systemet, dvs. funktionalitet. Man taler ofte om, at systemet indeholder en række funktioner.
- De **ikke-funktionelle** krav er kvalitetsegenskaber ved systemet. Det kan f.eks. være krav til
 - teknik (f.eks. hvilken platform/database eller lignende systemet skal kunne køre på)
 - systemets performance (svartider, datamængdekapacitet osv.)
 - brugbarhed

- pålidelighed
- sikkerhed (både mod uvedkommendes adgang og sikring mod at miste data)

Identifikationen af kravene kan ske på forskellige måder. For eksempel kan udviklerne lave interview med nogle af de kommende brugere, hvor de forsøger at spørge ind til emner som:

- Hvad er det du gør, når ...?
- Hvordan gøres det ...?
- Hvilken information er der behov for, når du gør ...?
- Osv.

Man kan også afdække krav ved at arrangere workshops, hvor både brugere og udviklere er til stede og diskuterer krav til systemet. Man kan som tidligere nævnt også søge i forskellige tekstbeskrivelser af virkeligheden, lave observation osv. for at blive klogere på virkeligheden og kravene til systemet.

Det er vigtigt at have fokus på at få afdækket begge typer krav. Kravene dokumenteres i en kravliste. Herunder ses et udsnit af en sådan kravliste.

Nr	Beskrivelse	Prioritet
	Funktionelle:	
K1	Med skoleadministrationssystemet skal administrationen kunne optage en studerende på en bestemt uddannelse.	M
K2	Med skoleadministrationssystemet skal administrationen eller den studerende selv kunne tilmelde den studerende til et kursus udbudt på uddannelsen.	M
K3	Med skoleadministrationssystemet skal administrationen eller læreren kunne registrere at en studerende har bestået en eksamen i et kursus og opnået en bestemt karakter.	M
K4	Med skoleadministrationssystemet skal administrationen hvert semester kunne oprette et nyt kursus i et bestemt fag på en uddannelse	M
K5	Med skoleadministrationssystemet skal der på nye kurser kunne tilknyttes lærere, der har de rette kompetence til at undervise på det pågældende kursus.	M
K6	Med skoleadministrationssystemet skal administrationen kunne beregne et karaktergennemsnit for en bestemt studerende.	S
	Ikke-funktionelle:	
K7	Skoleadministrationssystemet skal være hurtigt at anvende i det daglige arbejde, max 2 min. for oprettelse af et nyt kursus for en erfaren sekretær.	S
K8	Skoleadministrationssystemet skal være et pålideligt system med en opetid på 98 %.	M

Denne kravliste omfatter følgende

- **Funktionelle** og **ikke-funktionelle** krav
- En **entydig nummerering** af hvert krav (gør det nemmere at referere til kravene)
- En **beskrivelse** af hvert krav, eksempelvis efter følgende skabelon:

Nr – beskrivelse – prioritet

F.eks.: K1 – Med <System x> skal man kunne <funktion> – Must have

Prioriteringen af kravene kan f.eks. ske efter **MoSCoW** princippet:

- **Must have** – Fundamentalt/obligatorisk
- **Should have** – Vigtigt men kan udelades
- **Could have** – Kan vente
- **Would like to have but not right now** – Ønske, kan vente til senere levering

Der kan også prioriteres efter andre principper, f.eks. ”1,2,3 osv.”, ”høj, middel, lav osv.”

Det er vigtigt, når man beskriver kravene, at det sker så konkret og målbart som muligt. Et konkret og målbart beskrevet krav siger præcis, hvad der skal laves og med hvilken præcision. Med et konkret og målbart beskrevet krav er det lettere at afgøre, om det er gennemført i det færdige system. Det er også vigtigt, at de enkelte krav er forholdsvis simple, så man ikke får beskrevet **for** mange ting i ét krav, for så kan det være svært at planlægge med, uddelegere og få markeret som gennemført før meget sent i udviklingsforløbet. Hvis de enkelte krav indeholder for meget, så kommer udviklerne let til at arbejde flere på samme krav, arbejde for længe på samme krav og arbejde på mange reelle krav på én gang. I så fald kan det være svært at få et overblik over, hvor man egentlig er i forløbet. Indeholder det enkelte krav kun få konkrete ting, så er det lettere løbende at markere dem som gennemført, og derved er det lettere at gennemskue, hvor langt man er i projektet, og hvad der arbejdes på lige nu.

De funktionelle krav kan knyttes til use cases, som vil blive beskrevet i næste afsnit. De ikke-funktionelle krav knyttes **ikke** direkte til use cases, idet de er tværgående egenskaber ved systemet. Håndteringen af ikke-funktionelle krav vil ikke blive behandlet i dette dokument.

2.1.2 Use cases

Use cases er en anden måde at beskrive funktionalitet i systemet på, end det at finde funktionelle krav

- Hvad skal man kunne gøre med systemet
- Hvordan skal interaktionen mellem aktøren og systemet være

Definition på use case:

”Tidsmæssig afgrænset sekvens af interaktioner med systemet, der giver et resultat af værdi for aktøren”

Definition på aktør:

”En rolle omfattende brugere eller andre systemer, der interagerer direkte med systemet”

Use cases kan beskrives på flere niveauer

- Overbliksniveau (outlined): **Use case diagram**
- Overbliksniveau: **Use case diagram med strukturer**
- Detaljeret niveau: **Use case beskrivelse**
 - Kort udgave
 - Mellemstor udgave
 - Fuld udgave
- Meget detaljeret niveau: **Sekvensdiagrammer**
 - Systemsekvensdiagrammer
 - Designsekvensdiagrammer

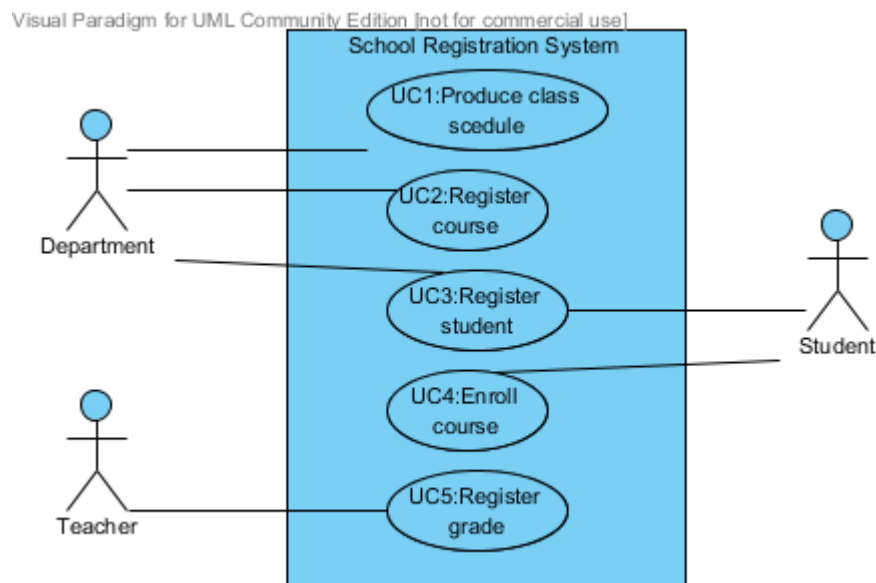
Use casen fortæller en **historie** om **aktørens interaktion** med systemet. Den illustrerer et **funktionelt krav** gennem den historie, den fortæller om interaktionen, og den er supplerende i forhold til kravlisten. Historien om aktørens interaktion med systemet beskrives i use case beskrivelsen.

Man identificerer use cases ved at fastlægge **afgrænsningen** af systemet for derefter at identificere **aktørerne**. For hver aktør (rolle) spørger man, **hvilken funktionalitet** denne stiller krav om for at kunne nå sine mål. Man spørger også om, **hvilke data** systemet skal kunne lagre og udtrække. Svaret på disse spørgsmål giver en række use cases, som kan illustreres i use case diagrammet.

2.1.3 Use case diagram

I Figur 2.1 herunder ses et uddrag af et use case diagram for et skoleadministrationssystem. Hovedelementerne er følgende

- IT-systemets afgrænsning vises med det store rektangel. Øverst angiver man typisk navnet på systemet.
- Aktørerne står uden for systemet og er angivet som tændstikmennesker. Bemærk, at aktører godt kan være andre systemer, men vises alligevel som tændstikmennesker. Man vil normalt kunne se af betegnelsen under figuren, om det er et menneske eller et system.
- Inde i systemrektangleret ses en række ovaler, der hver især dækker over en use case.
- Streger mellem en aktør og en use case angiver, at aktøren interagerer med systemet for at opnå et bestemt mål. Er der forbindelse mellem mere end én aktør og en use case, kan det betyde to ting: Enten at begge aktører indgår i en interaktion samtidig, eller at hver af aktørerne uafhængigt af hinanden kan gennemføre interaktionen. Det første ses relativt sjældent; et eksempel kan være almindelig betaling i et supermarked, hvor butiksassistenten og kunden begge indgår i interaktionen – kunden ved betaling med kort. UC3 herunder er et eksempel på den anden betydning, at enten afdelingen eller den studerende kan registrere en studerende.



Figur 2.1 Uddrag af use case diagrammet for skoleadministrationssystemet

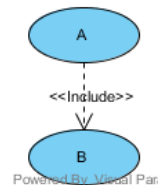
Der er nogle anbefalinger med hensyn til valg af use cases:

- **Use case navnet** skal helst være på formen <verbum>-<navneord>. Verbet bør endda være et "stærkt" **verbum**, dvs. et verbum der tydeligt beskriver resultatet af interaktionen, f.eks. "Enroll course", "Register grade", i modsætning til f.eks. "Do class schedule" (hvad skal man gøre med skemaet?).

- Der vil mange gange være nogle simple, sekundære opgaver, der skal løses i systemet. I skoleadministrationssystemet kan der være nogle opgaver knyttet til en lærer: "Opret ny lærer", "Fjern lærer", "Ret lærerens data", "Vis lærer". Den type use cases vil man finde for flere andre klasser, og man omtaler dem samlet som "**CRUD**" (Create, Read, Update, Delete). Sådanne use cases kan kombineres i **en** use case, som man f.eks. kan kalde "Vedligehold <X>" eller "CRUD <X>", hvor X er klassen (lærer i eksemplet). Brug dog kun denne løsning efter en grundig vurdering af use casene, hvor man kommer frem til, at der er tale om **simple** objekter og simple handlinger derpå.
- Alle systemer har en **Opstart** og **Nedluk** use case. Disse kan dog betragtes som trivielle og behøver ikke at blive medtaget i use case diagrammet.

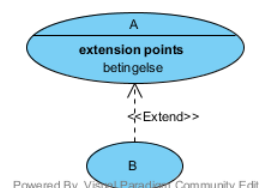
I det mere detaljerede use case diagram kan der anvendes en række strukturer til at beskrive sammenhænge mellem enkelte use cases. Følgende strukturer anvendes typisk (se brugen i diagrammet nedenfor):

- **<<Include>>** – viser at en use case B er indeholdt i en anden use case A (vises med en pil fra A til B). Det opfattes sådan, at use case B er en del af use case A og udføres hver gang (eller næsten hver gang), at use case A udføres. En tommelfingerregel er, at man kan bruge <<include>>, hvis interaktionen beskrevet i use case B både er en del af A og også kan foretages alene med en forbindelse til en aktør uden for systemet, eller hvis interaktionen i B både er en del af A og af en eller flere andre use cases. Man bruger <<include>> for at undgå at beskrive B's interaktion flere gange. Der er altså tale om genbrug af en klump interaktion.



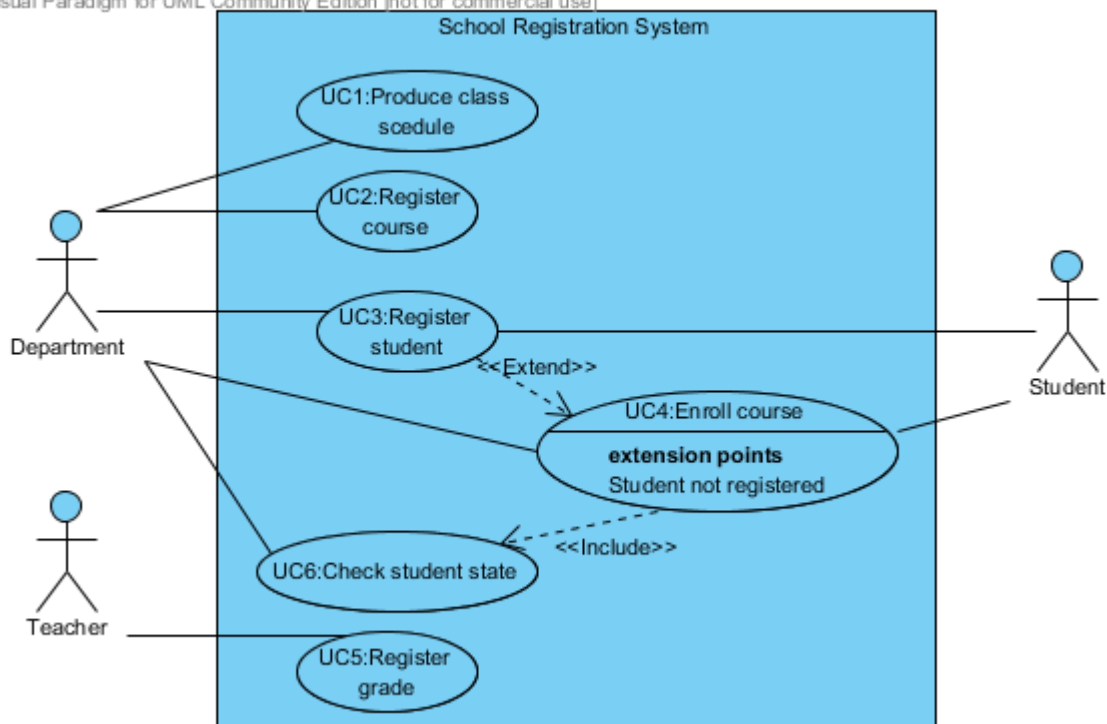
Bemærk, at man *ikke* bruger <<include>> til at beskrive dele af den enkelte use case, hvis delene *kun* indgår i denne use case. De enkelte dele af interaktionen beskrives i use case beskrivelserne (se nedenfor).

- **<<Extend>>** – udvider en use case, dvs. beskriver en speciel interaktion, der kun udføres en gang imellem, nemlig når en bestemt betingelse er opfyldt. Hvis der er en <<extend>>-pil fra B til A, vil man typisk i A angive, hvilken betingelse der skal være opfyldt, for at B udføres. Man kalder dette for et **extension point** (se figuren herunder).



På Figur 2.2 er der vist et eksempel på et use case diagram, hvor der er brugt strukturer. For eksempel er use casen UC6 "Check student state" både en <<include>> i UC4 "Enroll course" og en selvstændig use case, og UC3 "Register student" er en <<extend>> til UC4 "Enroll course", idet der er en interaktion, der kun sker sammen med UC4 i den specielle situation, hvor studenten ikke findes i forvejen.

Visual Paradigm for UML Community Edition [not for commercial use]



Figur 2.2 Use case diagram med strukturer for skoleadministrationssystemet

2.1.4 Use case beskrivelser

Et velvalgt navn på en use case giver udvikleren en god ide om den interaktion, der skal foregå mellem aktøren og systemet. Men som regel er denne overskrift ikke tilstrækkelig som grundlag for det videre arbejde. I så fald laver man use case beskrivelser, der detaljerer den ønskede interaktion mellem aktøren og systemet i forhold til det fremtidige system.

Use case beskrivelser kan laves på forskellige niveauer. Her beskrives tre niveauer.

En **kort use case beskrivelse** indeholder blot en kort tekstlig udvidelse af overskriften. Den korte beskrivelse kan man vælge, hvis det er klart, at interaktionen er ret oplagt. Følgende er et eksempel på en kort beskrivelse:

UC5: Register grade

Following an exam, the teacher registers the grade for one or more student.

En **mellemlang use case beskrivelse** beskriver mere formelt de trin, som interaktionen består af. Den kan også beskrive enkelte variationer i form af exceptions. Dette niveau kan man vælge, når interaktionen er lidt mere kompliceret. Her følger et eksempel:

UC3: Register student
Main flow: <ol style="list-style-type: none"> 1. The actor enters the cpr number for the student 2. The system verifies that the student is not already registered 3. The actor enters other personal data about the student (name, address, phone number etc.) 4. The actor chooses the education that the student applies for 5. The actor enters exam data 6. The system verifies that the exam data fulfills the requirements for the chosen education 7. The system accepts the student
Exceptional conditions: <ol style="list-style-type: none"> 6. Condition: The student's exam data does not fulfil the requirements <ol style="list-style-type: none"> a. The system prints a rejection letter for the student

Den fulde use case beskrivelse giver en god oversigt over interaktionen i systemet samt en række andre informationer i relation til use casen. Den fulde beskrivelse kan altid bruges og anbefales, hvis man på nogen måde er i tvivl, om de to kortere udgaver er tilstrækkelige til at arbejde videre på. Her følger først et eksempel på en fuld use case beskrivelse; bagefter følger forklaringen på de enkelte dele af den.

Use Case Name:	UC4: Enroll course	
Trigger Event:	Student wish to take a course	
Brief Description:	When the student choose to take a new course of a specific education, he or she register the enrollment to the course in the system	
Actors:	Student or Department	
Related use cases:	UC3: Register student, UC6: Check student state	
Stakeholders:	Student, School	
Precondition:	Course is registered	
Postcondition:	Student is enrolled course	
Flow of events:	Actor <ol style="list-style-type: none"> 1. The actor enters name or cpr number of the student 2. If the system did not find the student in the system go to <<extend>> UC3: Register student 3. The actor asks for possible educations 4. The actor chooses the wanted education and the correct semester 5. The actor chooses a course and confirms the choice 	System <ol style="list-style-type: none"> 1.1 The system displays the information about the student 3.1 The system displays available educations 4.1 The system displays the courses of the chosen education and semester 5.1 The system checks the student state <<include>> UC6: Check student state and creates an enrollment of the student to the course
Exceptional Flows:	<ol style="list-style-type: none"> 4. Condition: No course is suited for the student <ol style="list-style-type: none"> a. The actor interrupts the use case 5.1. Condition: The student state is not valid <ol style="list-style-type: none"> a. The actor is forced to interrupt the use case 	

Den fulde use case indledes med en række generelle informationer om use casen:

- Et **id** og **navnet** på use casen fra use case diagrammet



- Hvilken **hændelse** fra virkelighed, der starter use casen
- En kort **beskrivelse** af use casen (svarer typisk til niveauet fra den korte use case beskrivelse)
- Hvilke **aktører**, der er involveret
- **Relaterede use cases**, dvs. hvilke andre use cases, der har <<include>> eller <<extend>> til denne use case
- Hvem der er **interessenter** i forhold til denne use case. Interessenter er personer eller grupper, som udførelsen af en use case har betydning for. Aktøren behøver ikke at være nævnt, idet denne automatisk opfattes som interessent.
- En **præ-betingelse**, dvs. en betingelse der skal være opfyldt *inden indledningen af use casens flow*, for at use casen kan gennemføres. Det kan være bestemte data, der skal være registreret osv.
- En **post-betingelse**, dvs. en betingelse er opfyldt, når use casen er gennemført. Også her kan der være tale om bestemte data, der er registreret. Bemærk, at post-betingelsen typisk ikke er opfyldt, hvis man slutter i exceptional flow.

Beskrivelsen af den **typisk** forekommende interaktion mellem aktøren og systemet er beskrevet under "**Flow of events**". Det aktøren gør er beskrevet under "**Actor**", og det systemet gør er beskrevet under "**System**". Rækkefølgen på events kan aflæses gennem nummereringen, f.eks. først 1, så 1.1, så 2, så 3, så 4, så 4.1 osv. Det er i flow-beskrivelsen vigtigt at fokusere på, hvad aktøren leverer til eller gør med systemet, samt på hvad systemet giver tilbage til aktøren (svar).

Nogle gange sker der noget i interaktionen, så den beskrevne typiske interaktion mellem aktøren og systemet ikke gennemføres helt til enden. Der sker noget alternativt, så gennemførelsen af den beskrevne interaktion må afbrydes før tid. Den interaktion, der forekommer i de situationer, kan beskrives under "**Exceptional Flows**".

2.1.5 Krydstabel

Hvis man i forbindelse med at finde frem til, hvad man skal kunne med systemet, både benytter sig af krav og use cases, kan man for at tjekke sammenhæng mellem disse lave en krydstabel. Faktisk laves der så et kvalitetstjek af, om de funktionelle krav bliver realiseret gennem de valgte use cases.

	UC1	UC2	UC3	UC4	UC5	UC6	Osv.
K1			x				
K2				x			
K3					x		
K4		x					
K5		x					
Osv.							

Er der tomme rækker eller tomme søjler i tabellen, bør man lige gennemgå sine funktionelle krav og use cases for, om der mangler et eller flere krav eller use cases, eller om der er nogle for meget et sted. Nogle use cases, som det ses for UC2, kan realisere flere krav, og et krav kan også realiseres af flere use cases. Der kan altså godt være flere krydser i en søjle eller en række. Bemærk at de ikke-funktionelle krav ikke tages med i denne tabel, de ikke-funktionelle krav udtrykker ønskede egenskaber ved hele systemet og knytter sig ikke til en specifik use case.

2.2 Klasser

De sidste afsnit har handlet om gennem use cases at beskrive, hvad man skal kunne med systemet. Nu vil vi vende fokus mod at få beskrevet, hvad systemet skal indeholde. Hvad systemet skal indeholde, beskrives med den objektorienterede tilgang ved hjælp af klasser og objekter samt egenskaber ved disse. Klasserne og sammenhængene mellem disse findes direkte i det problemområde, man arbejder med.

2.2.1 Finde klasser

For at finde frem til hvilke klasser og attributter, der skal med i modellen, tages en af de kendte teknikker i brug. Den teknik handler om at kigge eller lytte efter navneord i beskrivelser af virkeligheden. En virkelighed kan bl.a. beskrives af en person, som vi stiller nogle spørgsmål. Den kan også være beskrevet i div. dokumenter f.eks. arbejdsgangsbeskrivelser eller kan opstå gennem observation af arbejde gennemført af brugerne osv. Der er mange måder at blive klogere på virkeligheden på. Et navneord kan blive til en klasse eller en attribut i vores model. Et navneord kan også bare være et synonym for et andet navneord eller det kan være noget andet, som bare ikke skal med i vores model. I processen med at spotte navneord, laves en liste med navneord, som er kandidater til klasser og attributter. Denne liste kalder vi en kandidatliste. Flg. er et eksempel på en kandidatliste til skoleadministrationssystemet

Kandidatord	Beskrivelse af hvad kandidatordet er og evt. hvad der skal gemmes
Lærer	En person der underviser på en uddannelse
Fag	Et fag der udbydes på en uddannelse
Kursus	Evt. en afholdelse af et fag i et bestemt semester
Navn	Egenskab ved f.eks. en lærer eller studerende
Uddannelse	Den bestemte uddannelse
Underviser	Evt. et synonym for lærer, der bør vælges mellem de to begreber
Studerende	En person der er tilmeldt en uddannelse
Karakter	Registreres når en studerende har været til prøve på et bestemt kursus
Beskrivelse	En tekst der registreres på et fag, som beskriver faget
Elev	Evt. et synonym for studerende, der bør vælges mellem de to begreber
osv.	

Det er også muligt, som supplement til teknikken med navneord, at finde frem til kandidater til klasser ved at tjekke tjeklister over potentielle klasser. Tjeklisten kan indeholde eksempler på typiske klasser i nogle typiske kategorier f.eks.



- fysiske objekter (bog, bil, hotel, vare, dokument osv.)
- personer og roller (person, studerende, ansat, kunde, rådgiver, kassemedarbejder osv.)
- organisationer (division, afdeling, sektion osv.)
- begreber (ordre, tilmelding, registrering osv.)
- osv.

Når kandidatlisten er ”færdig” anvendes en række vurderingskriterier til at afgøre, hvilke af kandidaterne der reelt skal blive til klasser og attributter. Vurderingskriterierne er flg.:

- Er det en unik ting, der er behov for at registrere noget omkring
- Er det indenfor scope af systemet
- Er der mere end en forekomst af den
- Er det blot et synonym for en anden ting, der allerede er identificeret
- Er det blot noget output produceret af anden information, der allerede er identificeret
- Er det blot en attribut ved en anden ting, der allerede er identificeret
- Osv.

Efter at have brugt vurderingskriterierne i forhold til kandidatlisten, skulle der gerne nu være dannet et godt billede af, hvilke klasser og attributter der skal med i modellen af systemet. Der kan dog komme ændringer hen ad vejen, efterhånden som udviklerne bliver klogere på virkeligheden.

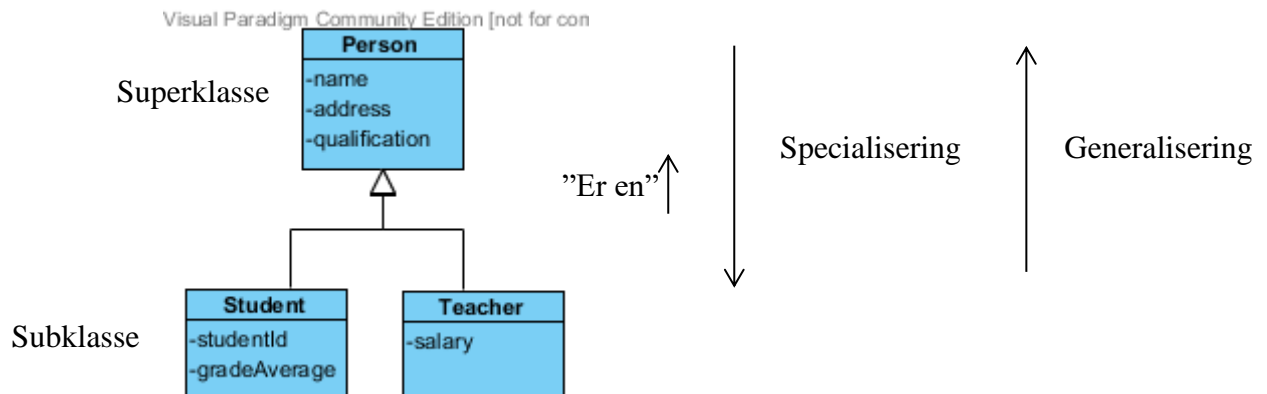
2.3 Sammenhænge mellem klasser

Efter at have fundet frem til hvilke klasser, der skal med i modellen, skal vi nu vurdere, hvilke sammenhænge, der er eller skal være mellem klasserne. Nogle sammenhænge er statiske sammenhænge mellem klasser, mens andre sammenhænge er dynamiske sammenhænge mellem objekter. Det gælder dog for alle sammenhængene at de i UML tegnes som sammenhænge mellem klasser.

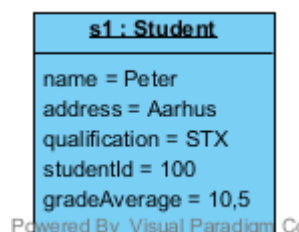
2.3.1 Statiske sammenhænge

Generalisering/specialisering

Der modelleres med specialisering/generalisering, når der i virkeligheden findes fænomener, der har nogle egenskaber til fælles men også har nogle egenskaber, der er specielle for det enkelte fænomen.



I eksemplet ovenfor er der lavet en model af en virkelighed, hvor fænomenerne eksempelvis er nogle studerende og nogle lærere. Det gælder for disse fænomener at de har nogle fælles egenskaber, som vi vil registrere og derfor have med i vores model. De studerende og lærerne har egenskaberne navn, adresse og kvalifikation til fælles. Derudover har de studerende den særlige egenskab karaktergennemsnit og lærerne den særlige egenskab løn. De fælles egenskaber modelleres i en **superklasse** *Person*, de specielle egenskaber for studenten modelleres i **subklassen** *Student* og de specielle egenskaber for læreren modelleres i **subklassen** *Teacher*. Afhængig af hvordan virkeligheden er, så kan der laves objekter af alle tre klasser eller af kun de to. Hvis der i virkeligheden er fænomener, hvor man **kun** har behov for at registrere navn, adresse og kvalifikation, så skabes objekterne af superklassen *Person*. De fænomener hvorpå der også skal registreres et karaktergennemsnit, skabes der et objekt af subklassen *Student*, og de fænomener, hvorpå der skal registreres en løn, skabes der et objekt af subklassen *Teacher*. Hvis der i virkeligheden kun findes studerende og lærere, dvs. ingen fænomener hvorpå der **kun** skal registreres navn, adresse og kvalifikation, så gøres superklassen *Person* **abstrakt** (navnet på klassen er i kursiv i UML). At superklassen er **abstrakt** betyder, at der ikke kan laves objekter af den. Når der skabes et objekt af subklassen *Student*, så **nedarves** egenskaberne fra superklassen. Det vil sige at et objekt af *Student* får egenskaberne fra både klassen *Person* og klassen *Student*. Et eksempel på et objekt af *Student* kan ses i objekt diagrammet nedenfor.

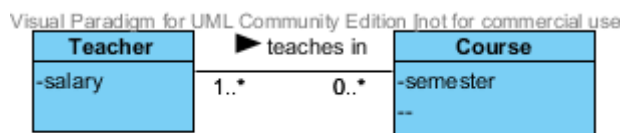


Det statiske i strukturen generalisering/specialisering ligger i, at et objekt bliver skabt som enten en *Person*, en *Student* eller en *Teacher* og det kan ikke ændres. Et objekt kan altså ikke både være en *Student* og en *Teacher* eller gå fra at være det ene til at være det andet. Har man brug for, at en person i systemet både kan være en *Student* og en *Teacher*, så skal det modelleres på en anden og mere dynamisk måde.

2.3.2 Dynamiske sammenhænge

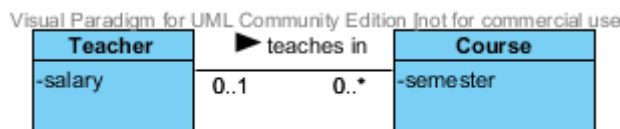
Associering

Associeringen er en dynamisk sammenhæng, altså en sammenhæng mellem objekter, der kan komme og gå i systemet. En associering mellem 2 klasser kan være en naturlig eller en nødvendig sammenhæng. Et eksempel på en naturlig sammenhæng er sammenhængen i eksemplet nedenfor, der viser, at en lærer underviser på et kursus.

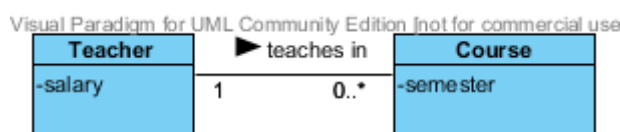


Multipliciteterne, som er de tegnede tal og stjerner på associeringen, er udtryk for, hvordan de dynamiske sammenhænge mellem objekter kan være. I eksemplet herover udtrykker multipliciteten ved *Course* at et objekt af *Teacher* kan tilknyttes 0 til mange objekter af *Course*. Dvs. en lærer kan undervise i 0 til mange kurser. Multipliciteten ved *Teacher* udtrykker, at et *Course* kan tilknyttes 1 til mange objekter af *Teacher*. Dvs. der kan være 1 til mange undervisere, der kan undervise i det fag. Pilen over associeringen kalder vi en læse-retningspil, og den kan sammen med en tekst være god til at forklare associeringen. I det her eksempel viser pilen, at der skal læses fra venstre mod højre, dvs. at en "teacher" "teaches in" et "course".

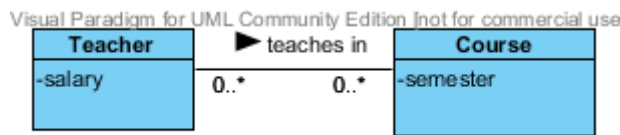
Nedenfor er vist eksempler på mulige multipliciteter på en associering.



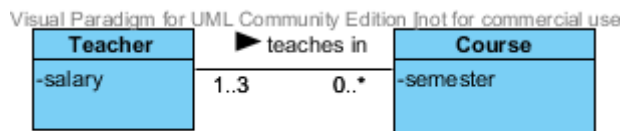
Det enkelte kursus har ingen eller 1 lærer tilknyttet.



Det enkelte kursus har præcis en lærer tilknyttet.



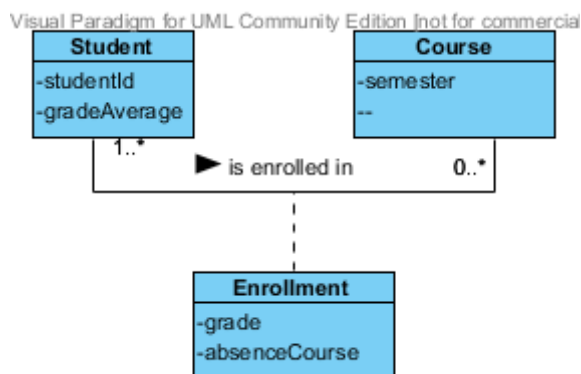
Det enkelte kursus har ingen eller flere lærere tilknyttet.



Det enkelte kursus har præcis en til 3 lærere tilknyttet.

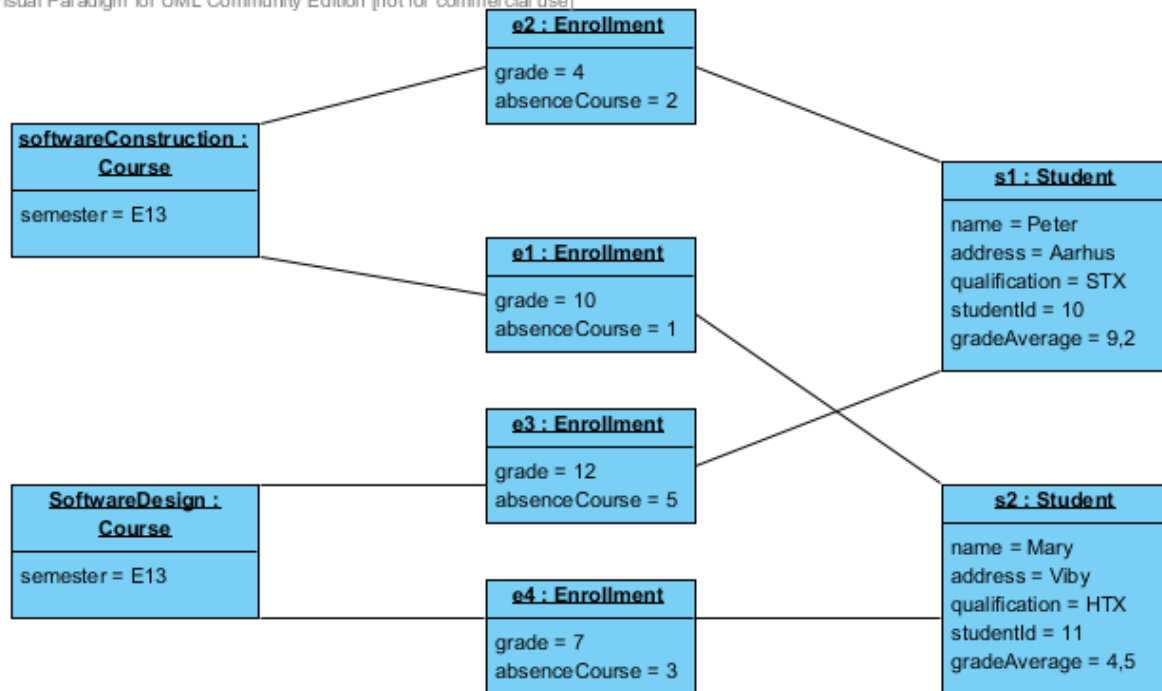
Associeringsklasse

Der modelleres med en associeringsklasse, hvis der på en mange til mange associering er brug for at registrere noget information på selve sammenhængen mellem de to klasser.



I eksemplet er der f.eks. brug for at registrere en karakter på sammenhængen mellem **Student** og **Course**. Karakteren kan ikke registreres på **Student**, da denne får mange karakterer, en karakter pr. kursus. Karakteren kan heller ikke ligge på **Course**, da et kursus har tilmeldt mange studerende, som hver skal have deres karakter. I den situation er vi nødt til at indføre en ny klasse, her kaldet **Enrollment**, som ligger mellem de to klasser. Denne klasse tegnes i analyse som en associeringsklasse, dvs. som en klasse med en stiplede linie fra klassen ind til associeringen. Et objekt af **Enrollment** vil referere til **et** objekt af **Course** og **et** objekt af **Student**, så derfor kan karakteren registreres her. Et eksempel på hvordan karaktererne er registreret på objekter af **Enrollment** kan ses i objektdiagrammet nedenfor.

Visual Paradigm for UML Community Edition [not for commercial use]

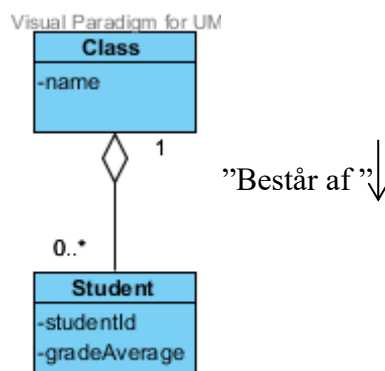


Figur 2.3 Objektdiagram for objekter af Course, Student og Enrollment

Aggregering

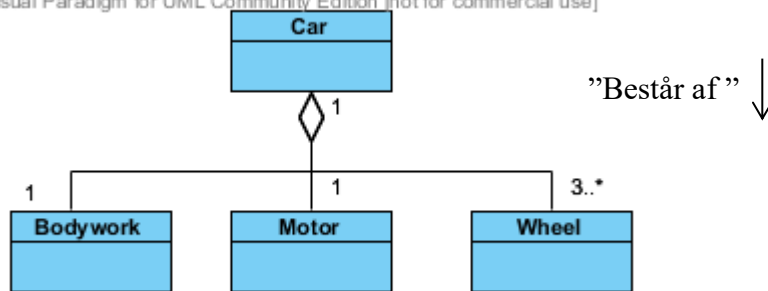
Aggregeringen er også en dynamisk sammenhæng mellem objekter. Aggregeringen anvendes når et overordnet objekt (helheden/foreningen) består af et antal mere underordnede objekter (delene/medlemmet).

Nedenfor har vi et eksempel på en **forening-medlem aggregering**. Modellen fortæller at en klasse består af et antal studerende. Det er typisk muligt at sige "består af" mellem helheden og delene, når der er tale om en aggregering.



Et eksempel på en **helhed-del aggregering** kan ses i nedenstående eksempel.

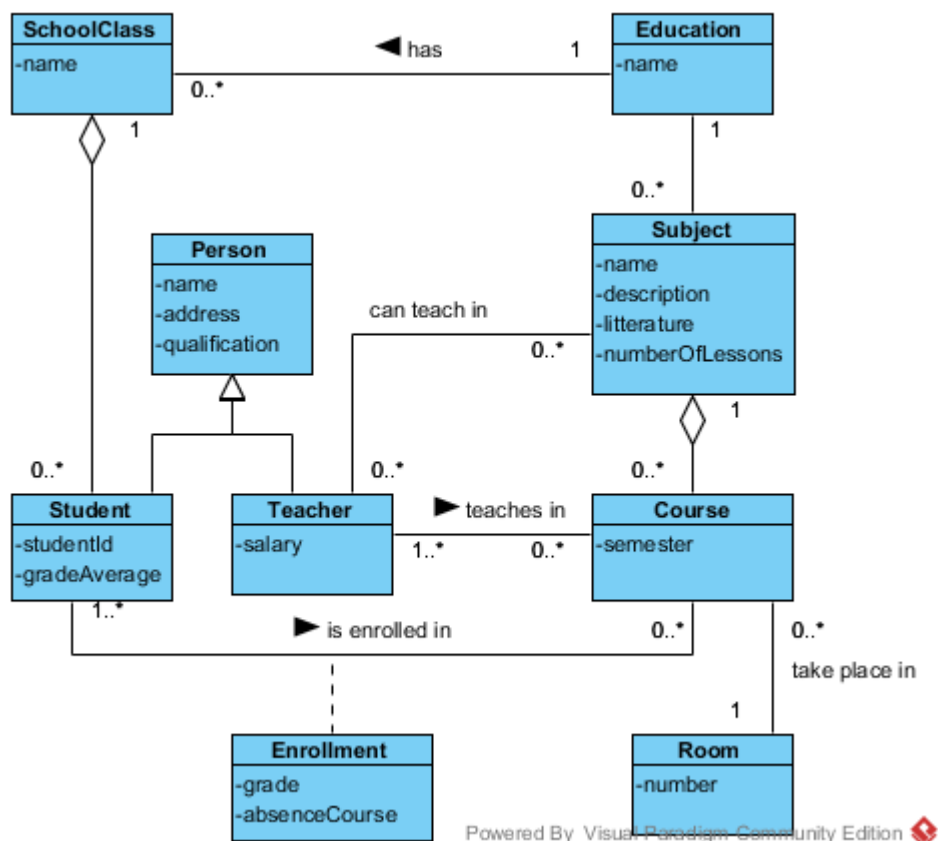
Visual Paradigm for UML Community Edition [not for commercial use]



Et objekt af **Car** har altså tilknyttet et objekt af **Bodywork**, et objekt af **Motor** og 3 eller flere objekter af **Wheel**.

2.3.3 Analyseklassediagram

Når alle analyse klasser, dvs. klasser der modellerer ting og begreber fra virkeligheden/domænet, er fundet og forbundet med strukturer, danner der sig et såkaldt klassediagram. Nedenfor ses et eksempel på et analyseklassediagram for vores skoleadministrationssystem.



Figur 2.4 Analyseklassediagram for skoleadministrationssystemet

2.3.4 Opgaver

Opgave 2.1 Hvor mange attributter skal have en værdi, når der oprettes et objekt af `Student` og hvor mange attributter skal have en værdi, når der oprettes et objekt af `Teacher`?

Opgave 2.2 Hvilke objekter skabes der, når en ny studerende melder sig til et kursus?

2.4 Analysemønstre

I den objektorienterede analyse skal vi blandt andet som nævnt tidligere finde ud af, hvad systemet skal indeholde og modellere det ved hjælp af klasser og sammenhænge. Nogle gange er det ikke helt oplagt, hvordan modellen skal se ud, hvis man ønsker et vist niveau af forståelighed, genbrugbarhed, fleksibilitet osv. i modellen. For at opnå det mest hensigtsmæssige niveau mht. disse ting i en model, skal der nogle gange graves lidt dybere og tænkes lidt ekstra.

Modelleringsprincip: *Modeller er ikke rigtige eller forkerte, de er mere eller mindre brugbare*

Man bør selvfølgelig overveje jf. designkriterierne, hvor vigtigt det er med fleksibilitet og genbrugbarhed, for nogle gange gør det også modellen mere kompliceret at skulle leve op til de kriterier. Målet er at lave den simplest mulige model i forhold til behovet, dvs. byg ikke fleksibilitet ind, som ikke skal bruges. Er målet til gengæld at lave fleksible og genbrugbare modeller, kan det være en fordel at tænke i mønstre (patterns). Det at anvende mønstre er de seneste år blevet mere og mere almindeligt, og de kan anvendes mange forskellige steder i systemudviklingsprocessen.

Def. mønster: Et mønster er en generel løsning til et generelt defineret problem

Da mønstre kan anvendes mange forskellige steder i systemudviklingsprocessen, er der mange forskellige typer mønstre. Der er analysemønstre, som f.eks. anvendes i klassemodelleringen, der er designmønstre, som anvendes i forbindelse med valg af arkitektur og der er programmeringsmønstre, som anvendes i selve kodningen. De mønstre denne note koncentrerer sig om er analysemønstrene, altså mønstre der kan tages i betragtning, når vi laver klassemodeller. Analyse mønstrene er et resultat af erfaringer. Nogle har igennem modelleringspraksis fundet ud af, at selvom der er tale om forskellige problemområder, så findes der problemer, der ligner hinanden. Man har også fundet ud at disse problemer modelleringsmæssigt nogle gange kan løses på samme måde, dvs. at klasser og sammenhænge følger et bestemt mønster. Det der derfor kendetegner et mønster er, at det er sprunget ud af praksis og man har fundet et godt navn til det. Mønsteret er brugbart i forhold til et bestemt problem, som har vist sig dukker op i mange sammenhænge. Mønsteret er beskrevet på et skitsemæssigt plan, der kan anvendes som beskrevet, men som også kan tilpasses den konkrete situation, dvs. flg. gælder for et mønster

- **Navn:** Bør være kort og sigende
- **Problem:** Bør være brugbart i mange situationer

- **Løsning:** Bør angives på et skitse-mæssigt plan, så anvenderen selv kan tilpasse det til den konkrete situation

Udover at vi i denne note kun præsenterer analyse-mønstre, så præsenterer vi også kun et udpluk af kendte analyse-mønstre. Vil man læse om flere mønstre kan der henvises til bogen ”Analysis Patterns” af Martin Fowler² eller bogen ”Objektorienteret analyse og design” af Lars Mathiassen³ o.fl.

2.4.1 Rollemønstret

Et af de mønstre, vi har valgt at tage med i denne note, er rollemønstret. Problemstillingen, der optræder i flere forskellige problemområder, er, at en person kan have flere forskellige roller på en gang eller skifte rolle over tid. Hvis vi ikke lige tænker over det, kunne vi godt komme til at vælge at modellere det, som det ses på figur 2.5.

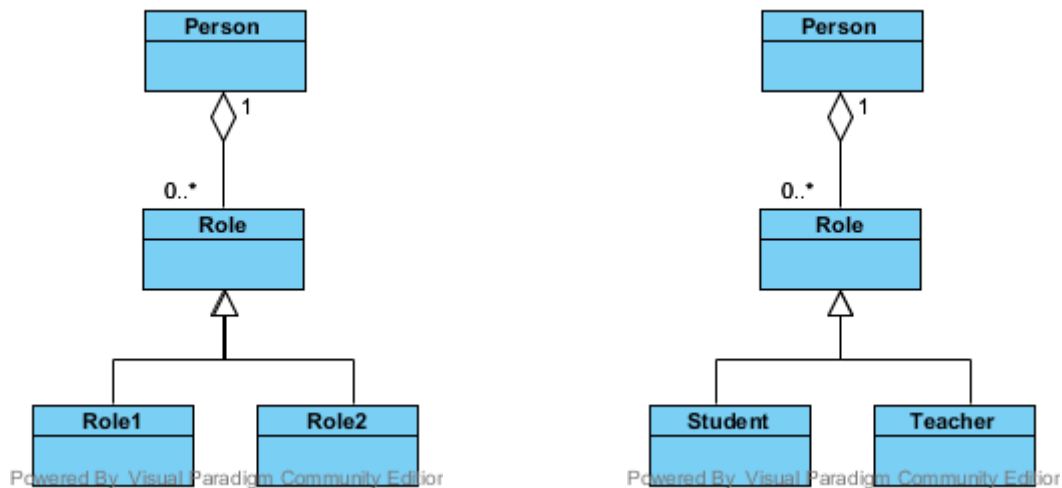


Figur 2.5: Statisk generel rollemodel og brugt i Skoleadministrationssystemet

Problemet med denne model er bare, at personen her ikke kan have flere roller på én gang eller skifte fra at have en rolle til at få en anden. Det er en statisk model, dvs. person objektet fødes som et objekt af `Person`, `Role1` eller `Role2` i den generelle model eller som objekt af `Person`, `Student` eller `Teacher` i modellen fra Skoleadministrationssystemet og vil forblive at være det, til det dør. I figur 2.6 nedenfor illustreres hvordan, man gennem praksis har fundet ud af, det er en god måde at modellere på, at en person skal kunne have flere forskellige roller på en gang eller skal kunne skifte rolle over tid.

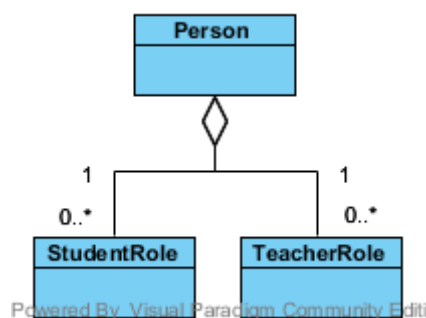
² ”Analysis patterns”, Martin Fowler, ISBN 0-201-89542-0

³ ”Objektorienteret analyse og design”, Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, Jan Stage, ISBN 87-7751-153-0



Figur 2.6: Rollemønsteret generelt og som det kunne have været brugt i skoleadministrations-systemet

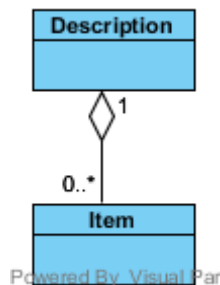
Denne model er, ved at der er indskudt en klasse *Role*, gjort mere dynamisk. Klassen *Person* aggregerer objekter af klassen *Role*, dvs. et objekt af *Person* kan have flere roller på en gang og der kan tilknyttes og fjernes roller som tiden går. Det man så bør bemærke her er, at et objekt af *Student* og *Teacher* ikke længere er et personobjekt, men et rolleobjekt med de egenskaber en rolle nu har. Specialiseringen fra klassen *Role* til *Student* og *Teacher* laves, hvis der er nogle egenskaber som rollerne *Student* og *Teacher* har til fælles. Har *Student* og *Teacher* rollerne ikke noget til fælles, kan det modelleres lidt enklere, hvilket kan ses på figur 2.7.



Figur 2.7: Rollemønsteret i en simplere variant

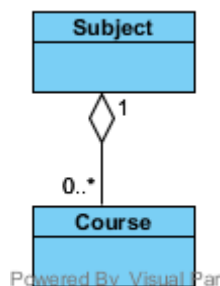
2.4.2 Genstand-beskrivelsesmønsteret

Et andet mønster som modellerer en ofte forekommende problemstilling er Genstand-beskrivelsesmønsteret. Problemstillingen her er, at et antal objekter har nogle generelle egenskaber til fælles, dvs. attributværdier til fælles. Den måde det modelleres på er, at der laves et objekt af en "overordnet" klasse, som holder på de fælles egenskaber for objekterne af den "underordnede" klasse. F.eks. definerer klassen "Description" de fælles egenskaber for alle de tilknyttede objekter af "Item".



Figur 2.8: Genstand-beskrivelsesmønsteret generelt

Figur 2.9 illustrerer en del af analyseklassediagrammet for skoleadministrationssystemet, hvor mønsteret ses anvendt i modelleringen af fag og selve afholdelsen af det konkrete kursus i faget.



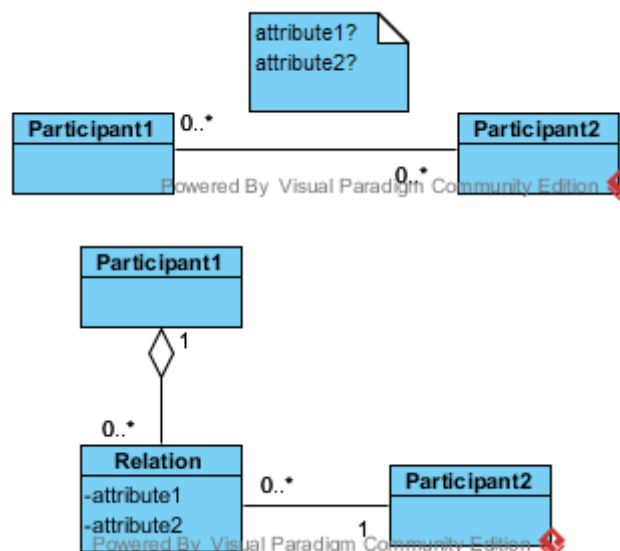
Figur 2.9: Genstand-beskrivelsesmønsteret anvendt i modellen for skoleadministrationssystemet

De forskellige afholdelser af kurser i et fag har den samme fagbeskrivelse, navn osv. til fælles. Den fælles fagbeskrivelse, navn osv. registreres i et objekt af `Subject`, hvorimod det,

der skal registreres for en afholdelse af et fag, nemlig det pågældende semester, de i dette semester og denne afholdelse tilknyttede lærere og studerende osv., registreres i et objekt af Course.

2.4.3 Relateringsmønsteret

Det næste mønster, som modellerer en ofte forekommende problemstilling er relateringsmønsteret. Som udgangspunkt er problemstillingen typisk, at Participant1 og Participant2 er blevet modelleret forbundet med en mange til mange associering. Undervejs i modelleringen finder man så ud af, at der er nogle egenskaber, man gerne vil registrere, men som ikke kan registreres på hverken Participant1 eller Participant2, men faktisk hører til på relationen i mellem dem. De egenskaber vælger man så at registrere på en ny klasse, som skydes ind imellem de to klasser. Modellen med mange til mange relationen og informationen, der ikke kan placeres på nogen af klasserne og den generelle løsning på problemet beskrevet i relateringsmønsteret ses nedenfor i figur 2.10.



Figur 2.10: Problemstillingen og det generelle relateringsmønster

Et mere konkret eksempel kunne være, at en bil over tid er ejet af flere personer, og at hver person kan eje flere biler. Hvis man vil registrere en *fra* og *til* dato for det konkrete ejerskab

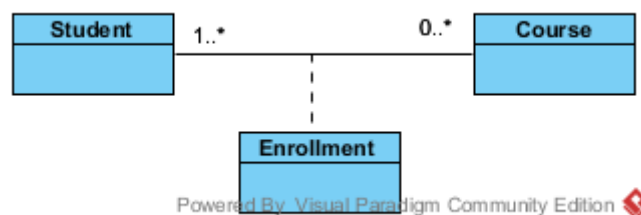
mellem en bil og en person, så har man, som det ses af figur 2.11, brug for en relationsklasse her kaldet *Ownership* mellem klasserne *Person* og *Car*.



Figur 2.11: Relateringsmønsteret anvendt i forhold til ejerskabet af en bil

I det generelle eksempel er der anvendt en aggregering fra *Participant1* til *Relation*. Med det antydes at *Participant1* holder på objekterne af *Relation*. Det behøver ikke være sådan, aggregeringen kan også være mellem *Participant2* og *Relation* eller, der kan være associeringer begge steder.

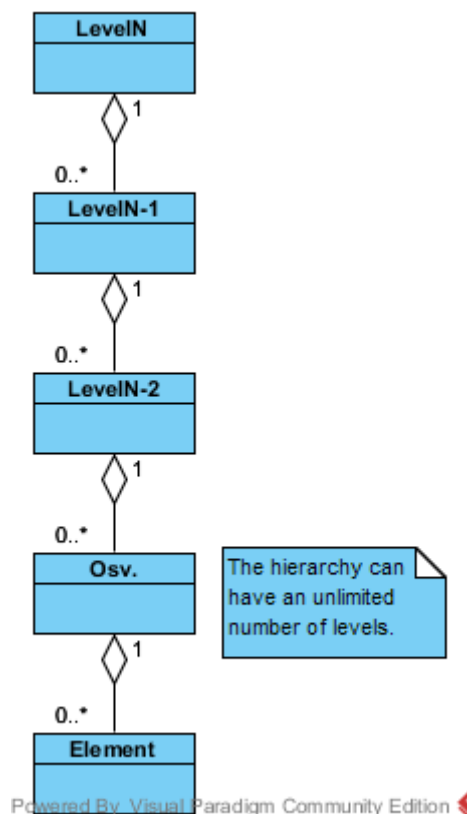
Nogle gange tegnes en relationsklasse i analyseklassediagrammet som en associeringsklasse. Dvs. når vi vælger at tegne en associeringsklasse i et analyseklassediagram, så tænker vi faktisk relateringsmønster. I analyseklassediagrammet for skoleadministrationssystemet er der tænkt relateringsmønster i forhold til, at de studerende kan tilmelde sig flere kurser og et kursus kan have tilmeldt flere studerende, og der er brug for at kunne registrere f.eks. en dato for tilmeldingen, en karakter pr. studerende pr. fag osv. Se figur 2.12 og kig evt. tilbage i noten og læs afsnittet omkring associeringsklasse.



Figur 2.12: Relateringsmønsteret anvendt i Skoleadministrationssystemet og tegnet som associeringsklasse.

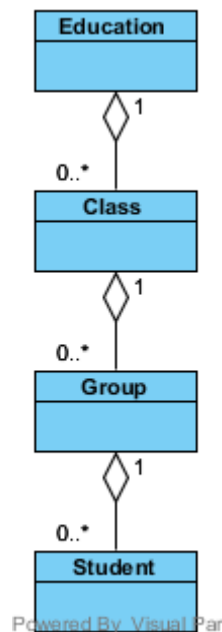
2.4.4 Hierarkimønsteret

I rigtig mange problemområder, er der en eller anden form for organisation, der skal modelleres. Det kan være, et firma der består af et antal afdelinger, som igen består af et antal sektioner, som igen består af et antal medarbejdere. Den slags hierarkiske organisationer kan som regel modelleres ved at søge inspiration i det mønster, der kaldes hierarkimønsteret. Det generelt beskrevne hierarkimønster ses nedenfor på figur 2.13.



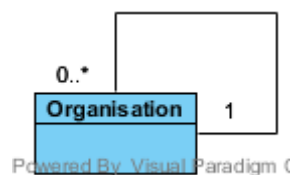
Figur 2.13: Generelt hierarkimønster

Det generelle mønster er beskrevet som om, der kan være uendeligt mange niveauer i hierarkiet. Når mønsteret konkretiseres, skal man beslutte sig for, hvilke og hvor mange niveauer, der skal være i hierarkiet. Der vil altså altid i den konkrete model være et bestemt og fast antal niveauer, som det kan ses i eksemplet modelleret i figur 2.14.



Figur 2.14: Hierarkimønsteret brugt i en konkret model, der, hvis man her havde ønsket at holde styr på og registrere noget om klasser og grupper, kunne have været modelleret ind i modellen for skoleadministrationssystemet

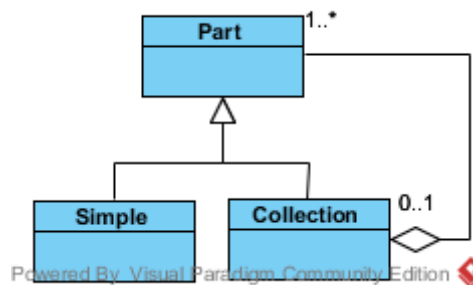
Hvis man ønsker en model, hvor det skal være muligt at lave et mere dynamisk hierarki, dvs. antallet af niveauer skal være dynamisk og kunne blive til flere over tid, så kan man lade sig inspirere af samlingsmønsteret figur 2.16, og modellere hierarkiet mere dynamisk som i modellen figur 2.15.



Figur 2.15: Hierarkimønster til modellering af et dynamisk hierarki

2.4.5 Samlingsmønsteret

Det næste mønster, som modellerer en af og til forekommende problemstilling er samlingsmønsteret. Det er ikke så ofte et anvendt mønster som de ovenfor beskrevne, simpelthen fordi problemstillingen i virkeligheden, hvor mønsteret kan bruges ikke optræder helt så ofte. Problemstillingen dukker dog op ind imellem, og så kan anvendelsen af samlingsmønsteret give en meget præcis måde at modellere problemstillingen på. Det generelt beskrevne samlingsmønster ses nedenfor på figur 2.16.



Figur 2.16: Generelt beskrevet samlingsmønster

Mønsteret kan bruges til at modellere en form for hierarki, men hvor strukturen og dybden af hierarkiet ikke kendes på modelleringstidspunktet. Som eksempel kan vi prøve at modellere en indholdsfortegnelse i et dokument. Dokumentet kan være opbygget af nogle afsnit, som igen kan være opbygget af nogle underafsnit og noget konkret tekst osv. se figur 2.17.

Indhold

Afsnit 1

Afsnit 1.1

Afsnit 1.1.1

Tekst

Tekst

Afsnit 1.2

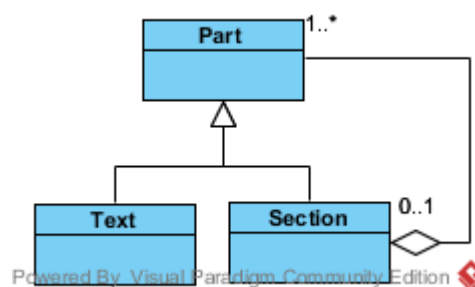
Tekst

Afsnit 1.3

Tekst

Afsnit 2

Tekst



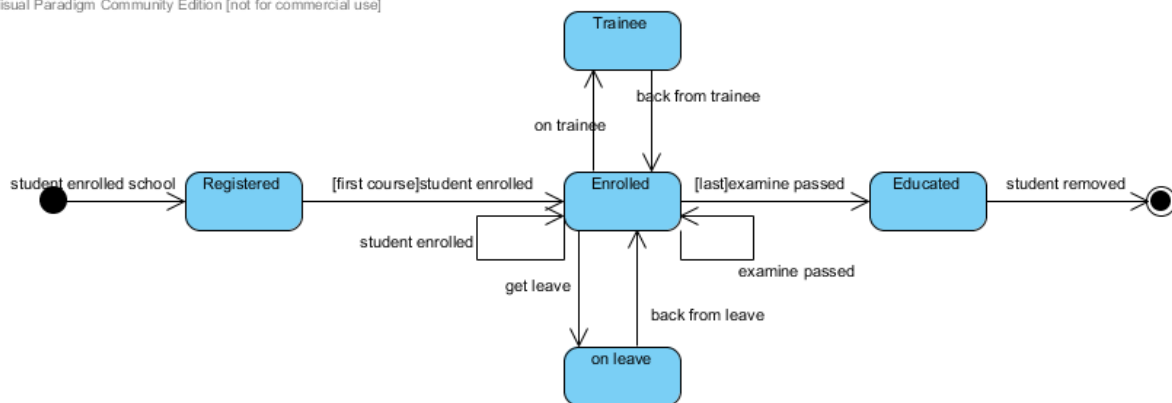
Figur 2.17: Samlingsmønsteret brugt til modellering af en indholdsfortegnelse i et dokument

Hvor afsnittene vil være objekter af klassen `Section` og kan bestå af og dermed samle nogle andre objekter af `Section` her underafsnit og/eller tekst, som er objekter af `Text` osv .

2.5 Klassers adfærd

Nogle objekter har undervejs i deres liv et bestemt adfærdsmønster, dvs. de bliver på bestemte tidspunkter påvirket af hændelser, som bringer dem i nogle bestemte tilstande. Hvordan hændelserne påvirker objektet og hvilke tilstande det ender i, beskrives i et adfærdsmønster. Nedenfor på figur 2.18 ses et adfærdsmønster for en studerende.

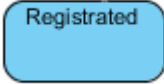



Visual Paradigm Community Edition [not for commercial use]



Figur 2.18 Adfærdsmønster/tilstandsdiagram for objekter af klassen Student

Den UML diagrammeringsteknik der bruges til at beskrive et adfærdsmønster kaldes et tilstandsdiagram. Diagrammet her viser, at et studerende objekt fødes i systemet, når hændelsen at den studerende bliver optaget sker. Den første tilstand et studerende objekt kommer i er tilstanden "Registered". Den tilstand kommer objektet i, når det registreres i systemet. Herefter sker hændelsen, at den studerende bliver tilmeldt en kursus, objektet kommer derfor i tilstanden "Enrolled". Adfærdsmønstret viser hvilke hændelser, der er mulige at påføre objektet i de enkelte tilstande. Når studerende objektet er i tilstanden "Enrolled", kan der ske en del hændelser. Den studerende kan blive tilmeldt en andet kursus eller kan bestå en eksamen. At den studerende bliver tilmeldt et andet kursus eller består en eksamen, ændrer ikke på objektets tilstand, det er stadig i tilstanden "Enrolled" og kan stadig påvirkes af de samme hændelser. Søger studenten til gengæld orlov og denne bevilges, så føres studenten i systemet over i en anden tilstand, nemlig tilstanden "on leave". I denne tilstand er det kun muligt at påvirke objektet med en bestemt hændelse, nemlig hændelsen, at den studerende kommer tilbage fra orlov. Når den studerende kommer tilbage fra orlov, bringes den studerende tilbage til tilstanden "Enrolled", hvor der igen kan bestå eksamener osv. Diagrammet er altså en beskrivelse af de mulige og lovlige hændelsesforløb for alle objekter af en klasse. Adfærdsmønsteret kan opfattes som et **abstrakt mønster** af hændelsesforløb for objekter i en klasse.

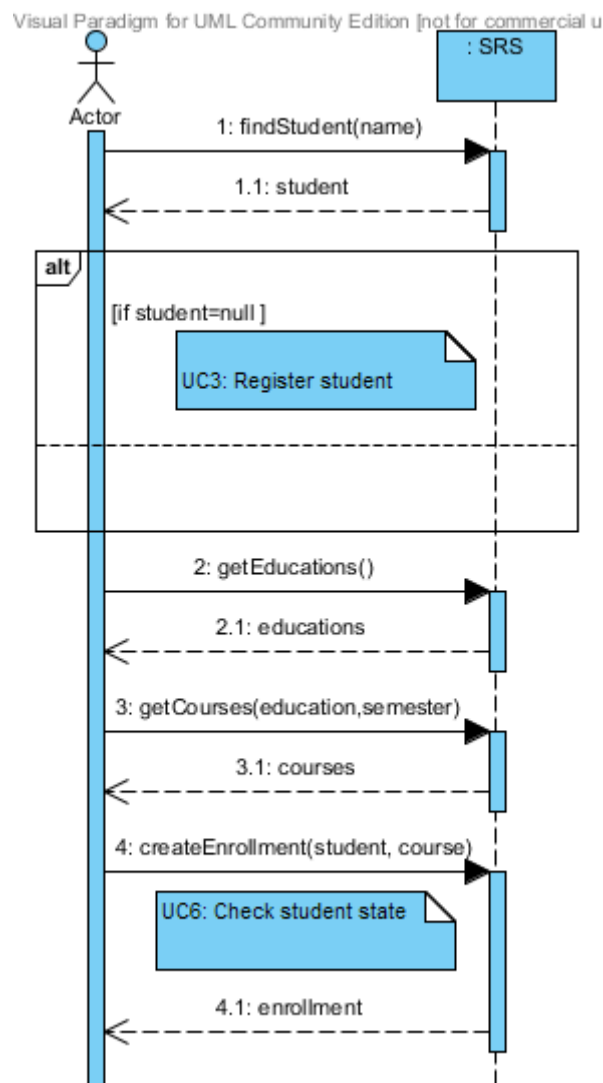
Adfærd for objekter af en klasse beskrives i et adfærdsmønster, når adfærd af klassens objekter er **tilstrækkelig kompliceret**, det beskrives altså ikke for alle klasser. Notationen i et adfærdsmønster er som i tilstandsdiagrammer og kan ses i tabellen nedenfor.

En tilstand	
En starttilstand	
En sluttilstand	
En hændelse, her med betingelse i []	[first course]student enrolled
En transition	

2.6 Sekvensdiagrammer

Systemsekvensdiagrammer

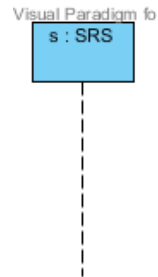
Use casen kan beskrives på et endnu mere detaljeret niveau end use case beskrivelsen. Teknikken til at beskrive use casen mere detaljeret er at anvende sekvensdiagrammer. I analysefasen anvendes det vi kalder for systemsekvensdiagrammer og i designfasen anvendes det vi kalder for designsekvensdiagrammer. Da vi stadig er i gang med analysen, koncentrerer vi os lige nu om systemsekvensdiagrammerne. Systemsekvensdiagrammerne bruges også som use case beskrivelserne til at modellere interaktionen mellem systemet og aktøren, men de har derudover specielt fokus på at vise, hvad og hvornår der er input og output til og fra systemet. Et eksempel på et UML systemsekvensdiagram for use casen "Enroll course" kan ses på figur 2.19 nedenfor.



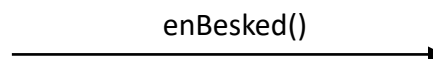
Figur 2.19 Systemsekvensdiagram for use case "Enroll Course"

Basis notationen i sekvensdiagrammer er flg.

- **Boks** (rektangel), i systemsekvensdiagrammet er der kun én boks, som udtrykker et objekt, der udgør det for hele systemet. Her skal objektet "s" f.eks. betragtes som et objekt, der udgøre hele School Registration Systemet.



- **Livline** (stiplet linie), indikerer objektets levetid. Den stiplede linie kan ses ovenfor.
- **Besked** sendes/modtages af et objekt, notationen på beskeden **kan** se ud som flg.



og kan have flg. syntaks

***[true/false betingelse] returværdi := beskednavn (parameterliste)**

* udtrykker gentagelse, kaldet kan forekomme flere gange, man kan også bruge en Loop-kasse, se nedenfor

[] udtrykker en betingelse, man kan også bruge en If-kasse, se nedenfor

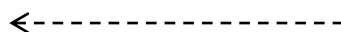
returværdi udtrykker en variabel til at opsamle en returværdi fra den sendte besked, kan bruges i stedet for en returpil.

:= udtrykker et assignment

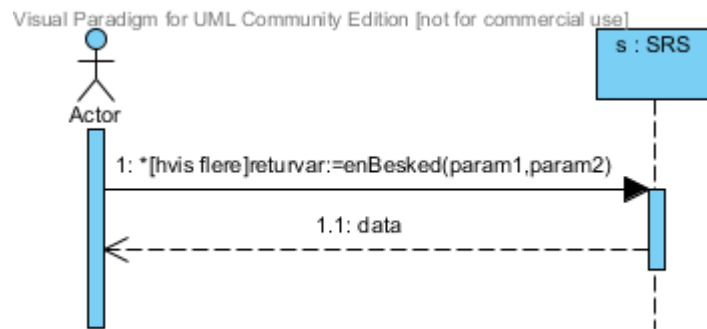
beskednavn udtrykker selve beskeden/metodekaldet til systemet

(parameterliste) udtrykker de parametre, der evt. sendes med beskeden.

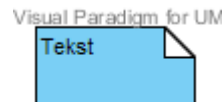
- **Returpil** (stiplet pil modsat beskedpilen), svaret på beskeden, her i systemsekvensdiagrammet det der afleveres tilbage fra system til aktør



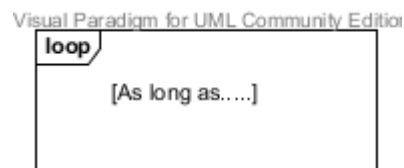
Eks. på besked og returpil tegnet i et case værktøj



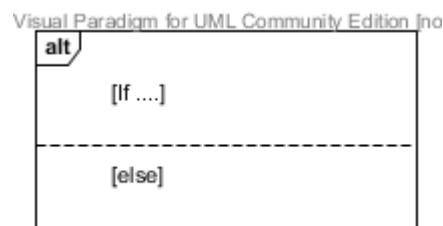
- **X** for enden af en livline, betyder at objektet dør.
- **Note**, notation for en include eller extend use case



- **Loop-kasse**, når noget interaktion gentages, alternativ til *



- **If-kasse**, når der under en betingelse skal ske én interaktion og ellers en anden interaktion



3 Design

I analysen har vi fundet frem til, hvad systemet skal indeholde, og hvad man skal kunne med systemet. I design skal vi nu have fundet ud af, hvordan IT-systemet, på baggrund af det vi har fundet ud af i analysen, skal realiseres. Ved at detaljere og forfine modellerne fra analysen (f.eks. analyseklassediagrammet og systemsekvensdiagrammerne) kan vi udarbejde et **detaljeret programmeringsgrundlag**. For at kunne tage nogle rigtige og fornuftige valg omkring designet af den interne logik i systemet, er vi nødt til at have nogle kriterier at designe efter. Vi skal altså i design starte med at vurdere, hvilke kriterier, der er vigtige for vores system.

3.1 Designkriterier

Et designkriterie er et kvalitetsmål, der fremhæver et bestemt aspekt ved et design. Jvf. [Mat] Lars Mathiasen kan flg. designkriterier være relevante at vurdere og prioritere.

Kriterie	Mål for
Brugbart	Tilpasningen af systemet til de organisatoriske, arbejdsmæssige og tekniske rammer
Sikkert	Sikringen mod uønsket adgang til systemets data og faciliteter.
Effektivt	Udnyttelsen af faciliteterne i den tekniske platform.
Korrekt	Opfyldelsen af de opstillede krav.
Pålideligt	Opfyldelsen af den krævede funktionalitet med den ønskede præcision.
Vedligeholdbart	Omkostningerne ved lokalisering og retning af fejl i det kørende system.
Testbart	Omkostningen ved test af systemet i forhold til de opstillede krav.
Fleksibelt	Omkostningen ved at ændre i det kørende system.
Forståeligt	Besværet for udvikleren ved at skaffe sig overblik over og forstå systemet.
Genbrugbart	Anvendeligheden af dele af systemet i andre beslægtede systemer.
Flytbart	Omkostningen ved at flytte systemet til andre tekniske platforme.
Integrerbart	Problemerne ved at sammenkoble systemet med andre systemer.

Figur 3.1: Klassiske kriterier for kvalitet af systemer

Via skemaet nedenfor vurderer og prioriterer man designkriterierne i forhold til det system, man står for at skulle udvikle. For rigtig at kunne arbejde med kriterierne er det dog ikke nok blot at prioritere dem, man er nødt til at bagefter for hvert prioriteret kriterie at definere et konkret krav for kriteriet. De krav, der på baggrund af kriterierne her defineres, er udtryk for en kvalitetsegenskab ved systemet, altså det samme som at opstille nogle **ikke-funktionelle krav** til systemet. Se mere om de ikke-funktionelle krav i afsnit 2.1.1.

Selve vurderingen og prioriteringen kan ske gennem udfyldelse af flg. skema.

Kriterie	Meget vigtigt	Vigtigt	Mindre vigtigt	Irrelevant
Brugbart				
Sikkert				
Effektivt				
Korrekt				
Pålideligt				
Vedligeholdbart				
Testbart				
Fleksibelt				
Forståeligt				
Genbrugbart				
Flytbart				
Integrerbart				

Kriterier som brugbart, fleksibelt og forståeligt vil næsten altid være vigtige kriterier. I praksis skal de realiseres gennem mere kontante kriterier, der knytter sig til designet af systemets dele: **kobling** og **samhørighed**.

Samhørighed er et udtryk for, hvor godt hver enkelt klasse, komponent eller delsystem hænger sammen. Det kan siges, at der er **høj samhørighed** når

- Sammenhængen indenfor klassen, komponenten eller delsystemet er høj
- Centrale operationer kan udføres indenfor klassen, komponenten eller delsystemet

Kobling er et udtryk for, hvor tæt to klasser, komponenter eller delsystemer hænger sammen, kobling forekommer f.eks. når

- en klasse, metode, komponent eller delsystem refererer til en anden klasse, komponent eller delsystem

Målet mht. kobling og samhørighed for at realisere kriterierne er

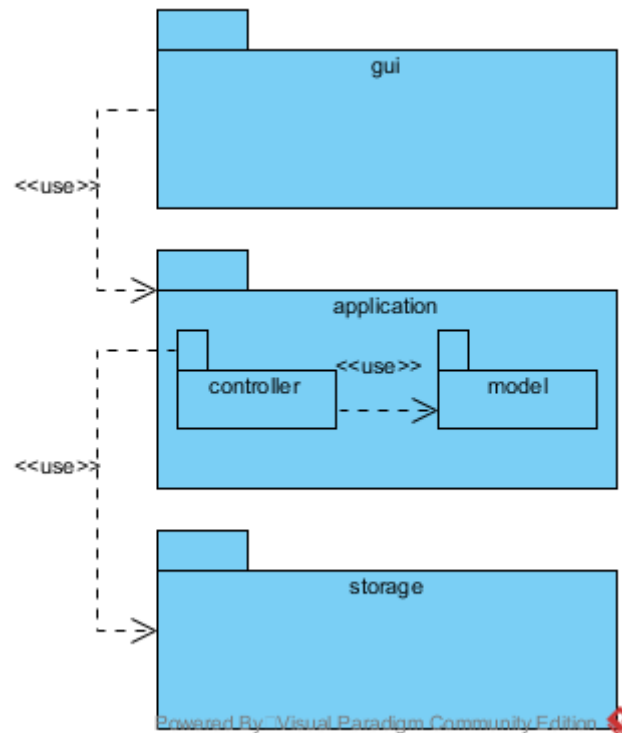
dele med **høj samhørighed** og **lav kobling**

Løsningen mht. opbygning af systemet, er altså at få opdelt sit system i nogle dele med høj samhørighed og lav kobling.

3.2 Arkitektur

Efter at have foretaget en vurdering og prioritering af kriterierne skal der vælges en arkitektur, der passer til de valgte kriterier. Flg. arkitektur er ét eksempel på en arkitektur, der kan være mange andre eksempler på arkitekturer, afhængig af prioritering af kriterier, hvilket system osv. Dette eksempel på en arkitektur er inspireret af den klassiske tre-lags model (View-

Domain-Data Access) og sikrer mht. den overordnede struktur i systemet ønsket om dele med høj samhørighed og lav kobling.



Figur 3.2 Eksempel på arkitektur

I en lagdelt arkitektur har vi følgende regel:

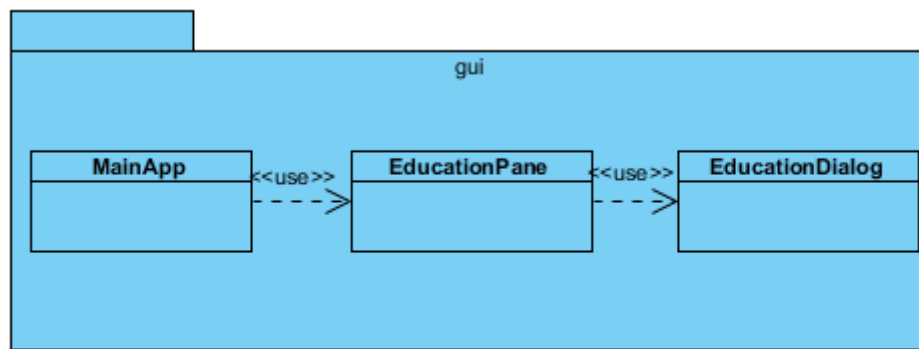
En klasse i et lag kan IKKE bruge noget fra lag OVENOVER dets eget lag

Så når du programmerer en klasse i et lag, må du lade som om, du ikke ved noget om (begreber, klasser og metoder) i laget ovenover.

Du kan altid bruge information fra klasser i det samme lag eller lag under.

3.2.1 GUI (Graphical User Interface)-laget

Dette lag indeholder klasser, der implementerer vinduerne (views) i brugergrænsefladen. Metoderne i dette lag kan ikke indeholde forretningslogik.



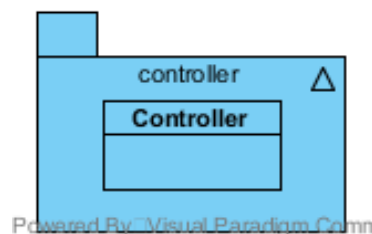
Powered By Visual Paradigm Community Edition

3.2.2 Applikationslaget

Dette lag indeholder pakkerne model og controller.

Controllerpakken indeholder en Controllerklasse (en eller flere afhængig af størrelsen på applikationen). Metoder i denne pakke kan indeholde forretningslogik.

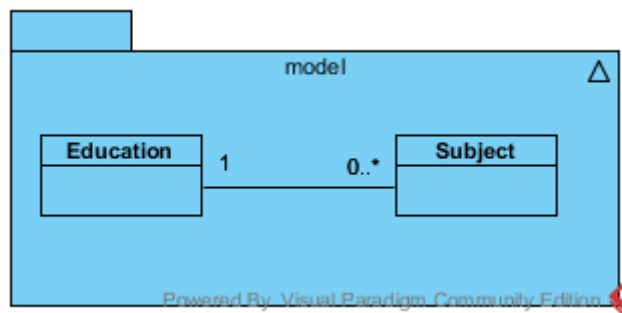
Controllerklasser håndterer use cases. I princippet har hver use case sin egen controllerklasse. I virkeligheden har vi ikke en controllerklasse for hver use case. De fleste use cases har kun brug for en enkelt metode i en controllerklasse, så vi kombinerer ofte flere use cases i en controllerklasse.



Controllerklassen har typisk metoder, der ikke naturligt kan placeres på en klasse i modelpakken, fordi metodens logik eksempelvis involverer flere klasser.

Controllerklassen har typisk metoder til at oprette og ændre en del af modellens objekter. Den har ikke metoder til at gemme objekterne, disse metoder er i denne arkitektur i klasserne i storage laget.

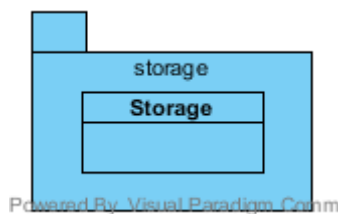
Modelpakken indeholder modelklasser (også kaldet forretningsklasser/domæneklasser) som Education, Subject etc. Associeringer eller andre strukturer forbinder klasserne. Metoder i model laget kan indeholde forretningslogik.



3.2.3 Storage laget

Dette lag indeholder klasser med metoder, der gemmer objekter af klasser, så man senere kan finde frem til og referere til disse objekter. Laget indeholder også klasser med metoder til at hente, opdatere og slette objekter.

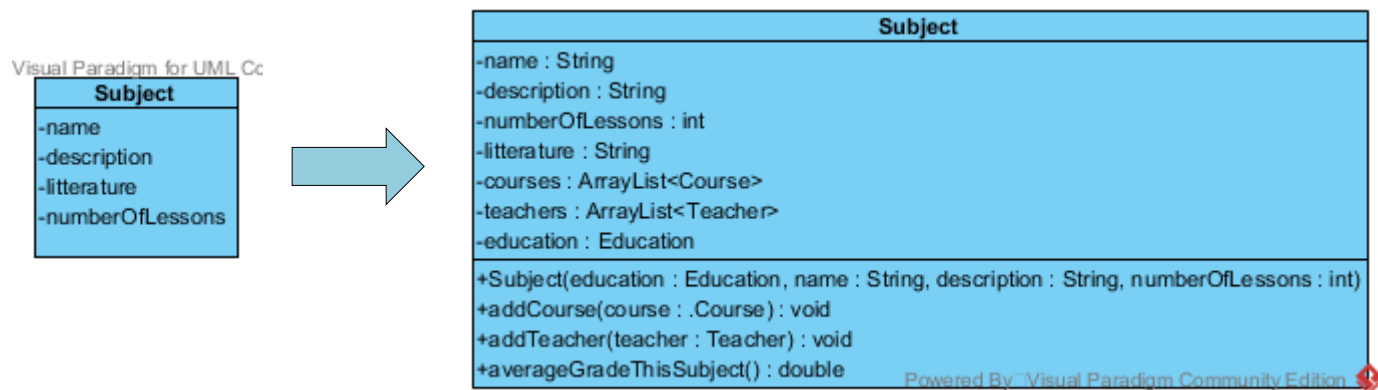
I princippet har hver klasse i model laget en tilsvarende Storage klasse. Der er dog model-klasser, som ikke skal gemmes i Storage klassen og i virkeligheden kombinerer vi ofte flere storage klasser i en klasse.



Model objekterne kan gemmes i computerens hukommelse (RAM), i filer, i en objekt-orienteret database eller i en relationel database. Ofte bruges en database, da objekterne i hukommelsen slettes når programmet lukkes ned.

3.3 Designklasser

I analysen har vi fokus på at finde frem til, hvad der skal registreres i systemet, dvs. vores analysemodel består af de klasser og attributter, der modellerer de ting og begreber fra den virkelige verden, som vi vil have registreret i systemet. Når vi bevæger os ind i designfasen, får vi mere og mere fokus på, hvordan vi kan realisere modellen i forhold til evt. valgte designprincipper og kriterier, men også i forhold til valgte programmeringssprog. Vi skal nu til at beslutte os for typer på attributter, hvilke metoder, hvor metoderne skal ligge, retninger på associeringer og aggregeringer, om nogle aggregeringer skal ophøjles til kompositioner, om der er brug for specielle designklasser osv.

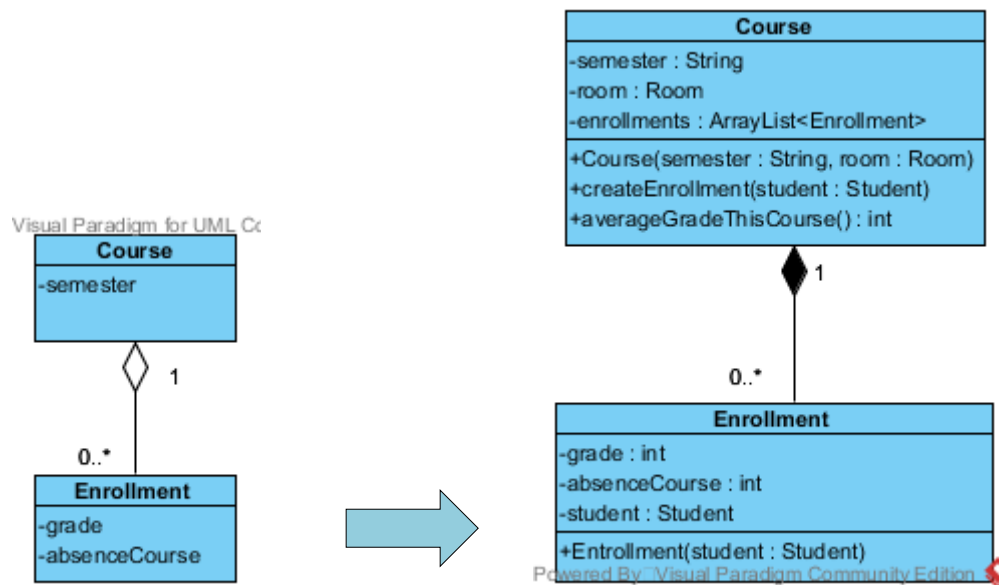


Figur 3.3 Analyseklasse og designklasse

Det kan ses på designklassen, at der er taget beslutninger omkring typer på attributterne. Der er taget beslutning om, hvilke metoder der skal ligge på denne klasse samt deres parametre og returtyper. Det er besluttet hvilke retninger, der skal være på associeringerne og attributterne `courses` og `teachers` er en realisering af disse associeringer fra `Subject` til `Course` og fra `Subject` til `Teacher`. De to attributter kalder vi også for link-attributter og da begge associeringer har multipliciteten ”mange” bliver typen en `Collection` her en `ArrayList`.

Aggregeringerne bør man ved design overveje en ekstra gang. Man bør overveje, om de skal forblive en aggregering, blive til en komposition eller blot skal være en alm associering. Valget afhænger af hvor stærkt objekterne hænger sammen i virkeligheden og dermed, hvor meget ansvar man i sit design lægger på helheds/foreningsklassen i forhold til håndtering af del/medlems objekterne. Er virkeligheden så **kun** helheds/forenings objektet skal kende til/referere til sine del/medlems objekter i hele deres livsforløb, kan helheden lige så godt stå for at oprette og nedlægge objekterne. I den situation vælges en komposition, hvilket altså betyder, at helheden **skal** stå for oprettelse og nedlæggelse af del-/medlemsobjekterne. Der kan være situationer, hvor sammenhængen mellem helhed og del ikke er så stærk, at en komposition vælges. En af de situationer er, når virkeligheden er sådan, at del-/medlemsobjekterne kan skifte relation til et andet objekt af helhedsklassen. I den situation vælges blot en aggregering.

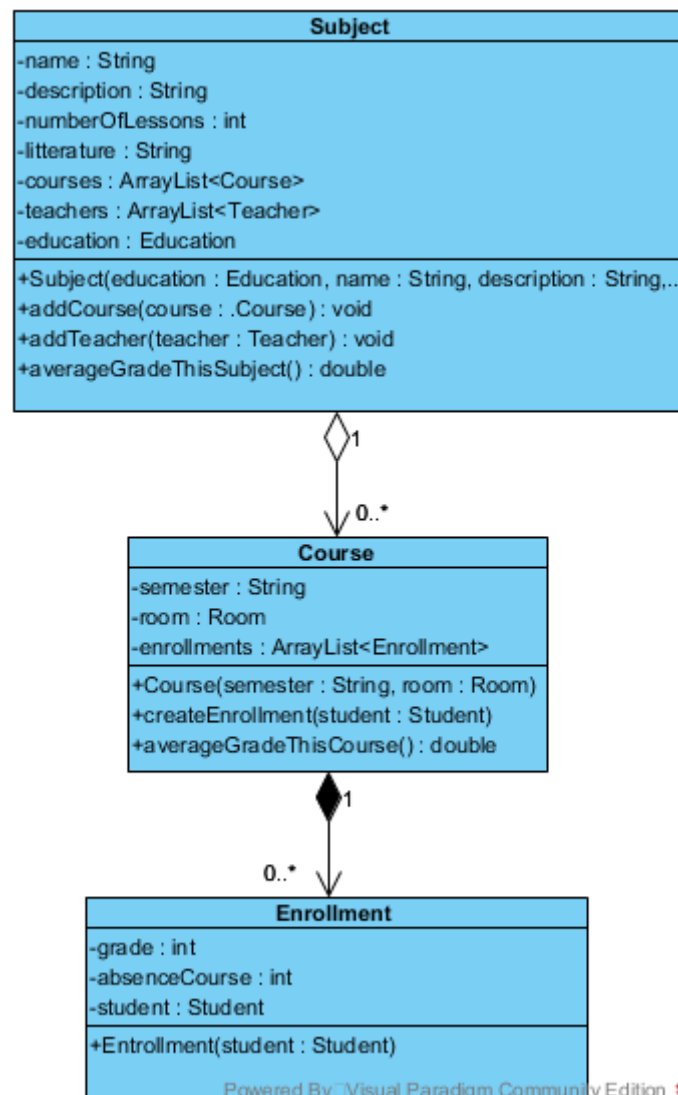
Hvis der er valgt aggregering, **kan** helheds/forenings objektet godt stå for oprettelse af del/medlems objekterne, men del/medlems objekterne kan også være oprettet af et andet objekt af helhedsklassen eller oprettet af et objekt af en helt anden klasse f.eks. en controller klasse, og derefter tilføjet til helheds/forenings objektets samling af del-/medlemsobjekter. Er virkeligheden så sammenhængen mellem objekterne er mere løs, og helheds/forenings objektet ikke skal have noget ansvar for oprettelse af del/medlems objekterne, vælges en associering. Se figur 3.4 for eksempel på at en aggregering bliver lavet om til en komposition.



Figur 3.4 En aggregering er valgt til at skulle blive til en komposition

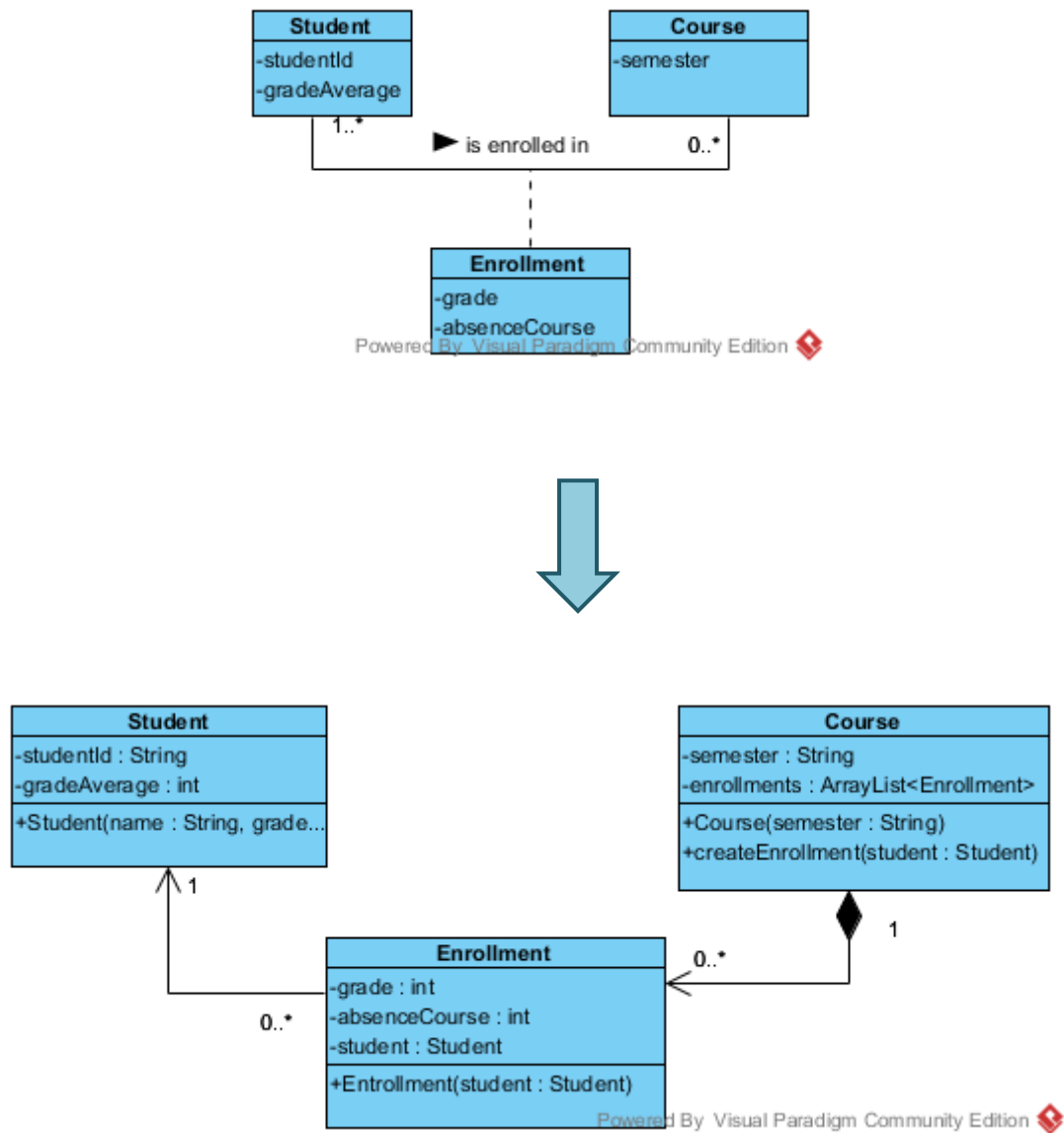
Hvis vi tænker i metoder, så betyder det, at ved **komposition skal** helheds/forenings objektet have metoder til **oprettelse** og evt. **nedlæggelse** af del-/medlemsobjekterne f.eks. metoden "createEnrollment(..)" på klassen "Course" i figuren ovenfor. Ved **aggregering kan** helheds/forenings objektet have metoder til både oprettelse (create-metode), evt. nedlæggelse og tilføjelse (add-metode) af del-/medlemsobjekterne. Ved **associering skal** helheds/forenings objektet kun have en metode til at tilføje (add-metode) del-/medlemsobjekterne.

Valg af **retninger** på associeringerne og aggregeringerne tages på baggrund af, om de enkelte metoder til udførelse af deres ansvar har brug for direkte adgang til relaterede objekter. F.eks. har metoden averageGradeThisSubject() på klassen Subject brug for en retning fra klassen Subject til klassen Course og videre til klassen Enrollment, så derfor pile og tilsvarende linkattributter på disse klasser. Se et eksempel på de i UML tegnede retninger samt linkattributter på figur 3.5 nedenfor.



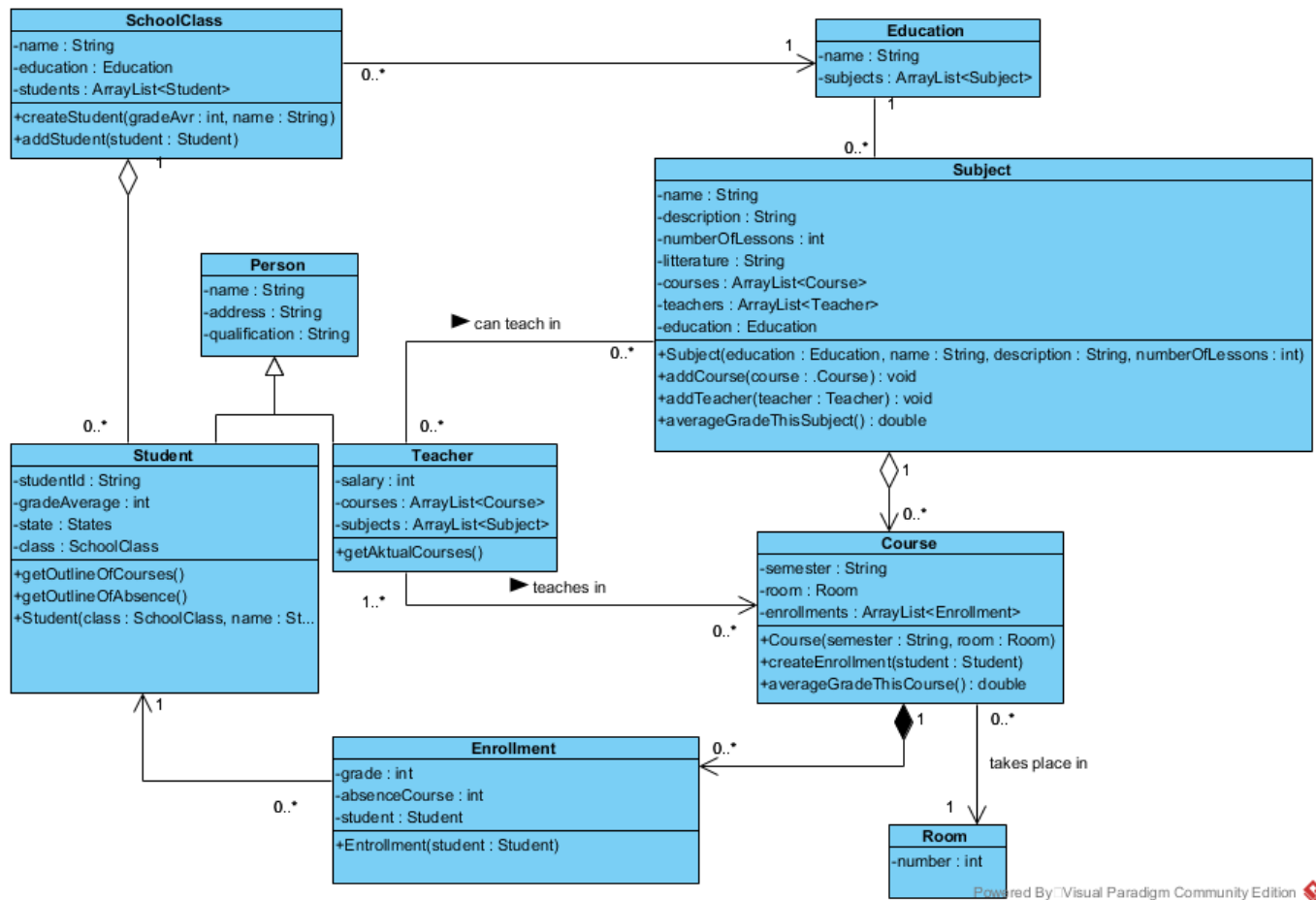
Figur 3.5 Retninger og linkattributter.

Associeringsklasser fra analyseklassediagrammet skal også designes, det er der imidlertid næsten en fast opskrift på. Associeringsklassen puttes ind mellem de to klasser, som det ses nedenfor på figur 3.6. På figuren nedenfor er der valgt strukturerne associering og komposition som sammenhænge, det er dog et valg. Det kan godt være almindelige associeringer begge steder, men ofte er det naturligt med en aggregering eller komposition et af stederne. Ansvar for oprettelse og opbevaring af objekter af associeringsklassen ligger ofte naturligt på en af de to klasser. Multipliciteterne på de to sammenhænge vil altid se ud som på eksemplet i figur 3.6, **en** ved de to klasser, her Student og Course, og **til-mange** på begge sider af associeringsklassen, her Enrollment.



Figur 3.6 Design af en associeringsklasse

Nedenfor på figur 3.7 kan man se et eksempel på et samlet UML designklassediagram. Det skal dog lige pointeres, at det ikke er et komplet diagram mht. metoder osv.

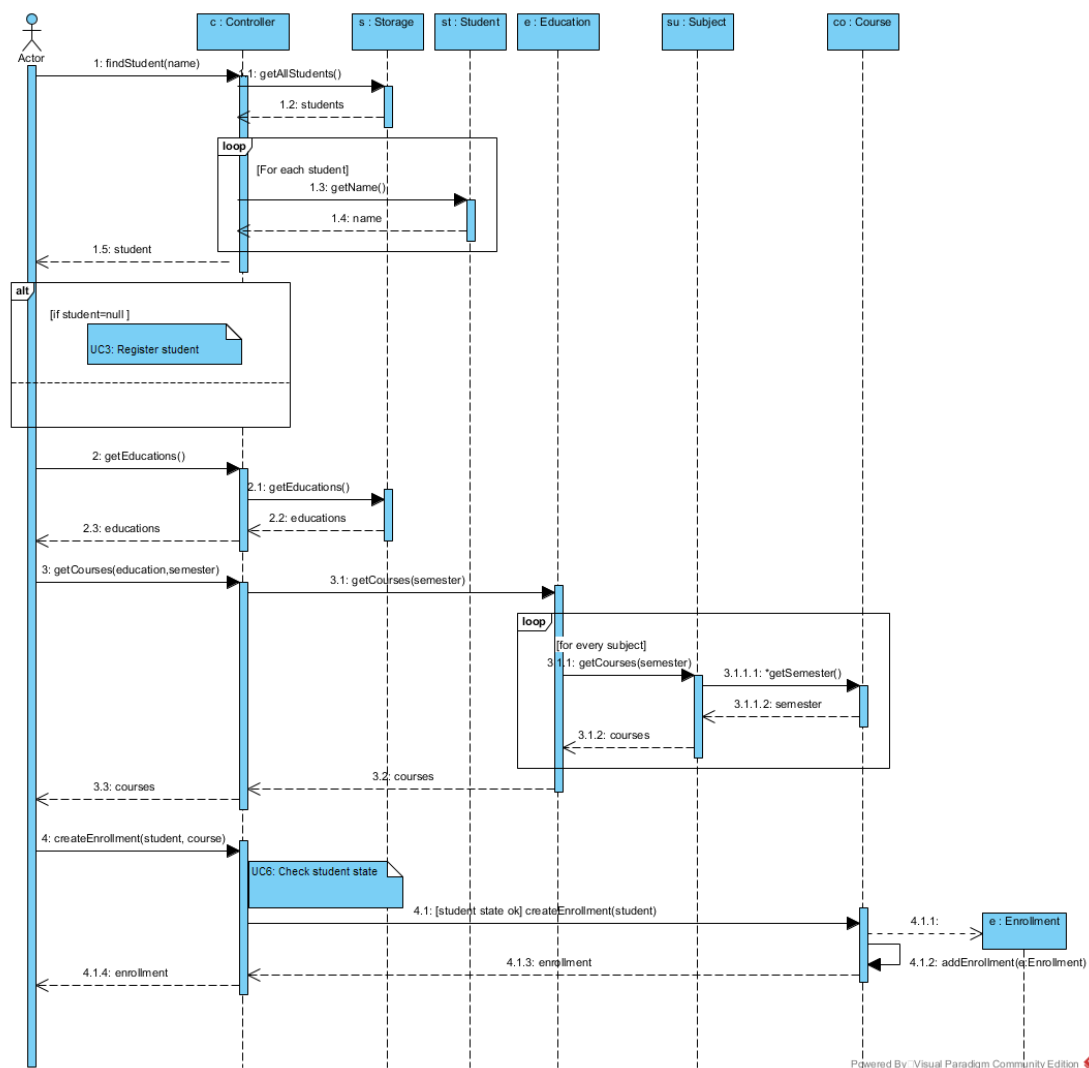


Figur 3.7 **Delvist** færdigt UML designklassediagram

3.4 Designsekvensdiagrammer

I designsekvensdiagrammerne modelleres samarbejde og interaktion mellem objekter, der realiserer en use case. **Interaktionsdiagrammer er abstraktioner**, dvs. vi vælger at vise noget og lade være med at vise noget andet, det er vores valg, hvad vi fokuserer på. Vi kan fokusere på en hel use case, dele af en use case eller enkelt metodekald i en use case osv.

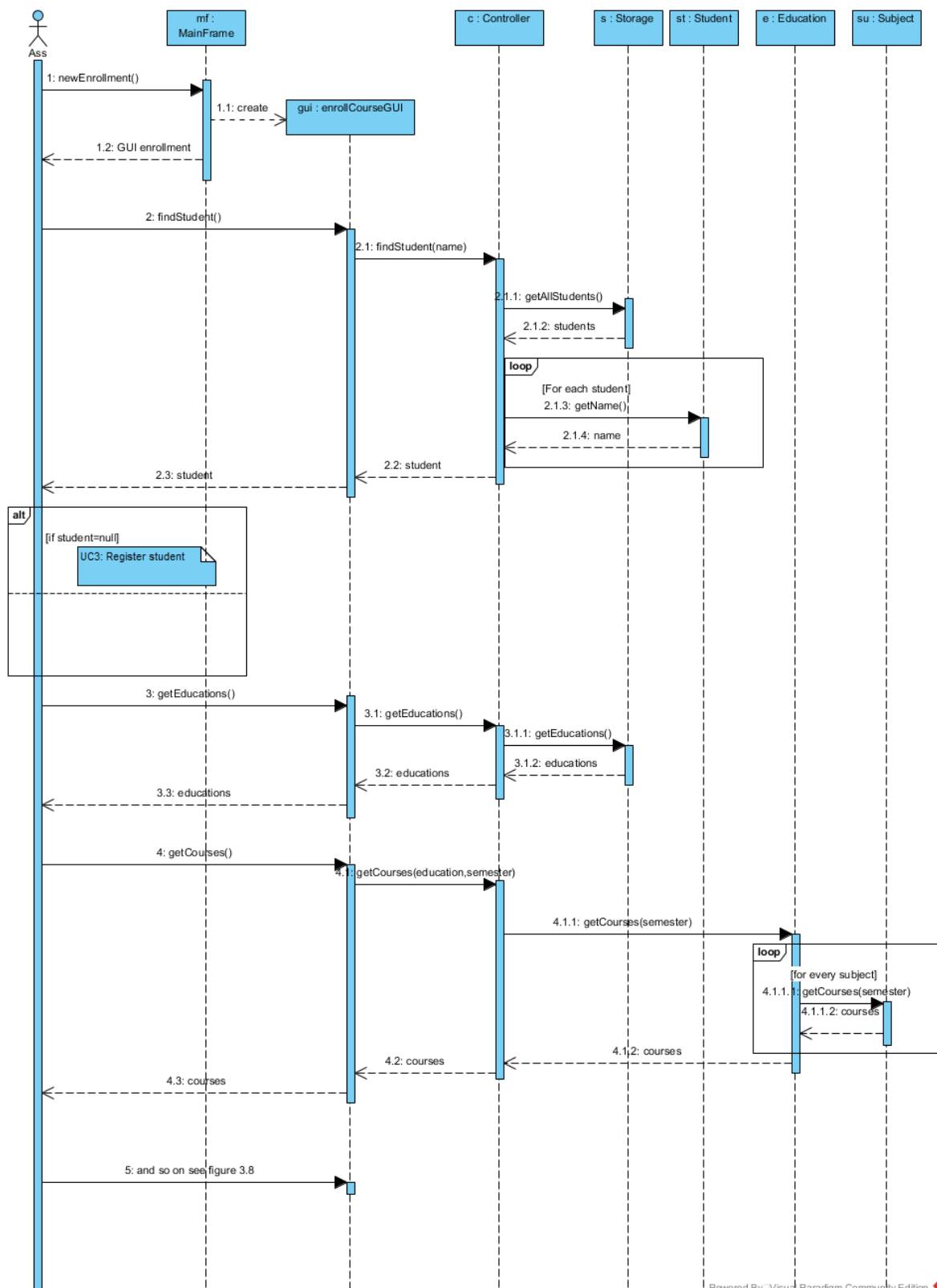
I designsekvensdiagrammer er der som regel flere bokse, da der som regel er flere objekter i spil. Boksen i designsekvensdiagrammet refererer nu til et **individuelt** objekt af en klasse. Notationen er et ":" og evt. et specifikt navn på et objekt foran klassenavnet, f.eks. "c:Controller". Et eksempel på et UML designsekvensdiagram for use casen "Enroll course" ses i figur 3.8.



Figur 3.8 Designsekvensdiagram for use case "Enroll course"

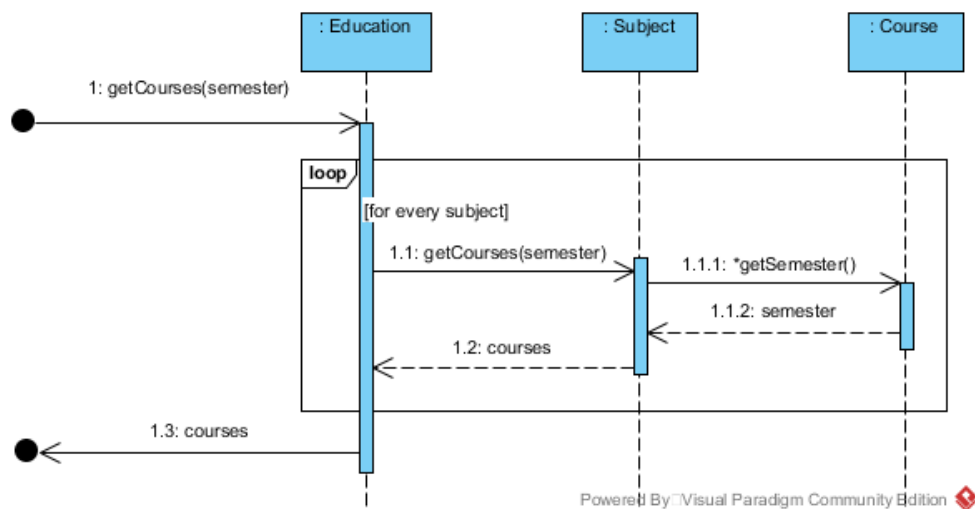
Bemærk at der er brugt en loop-kasse, som tegnes uden om al den interaktion mellem objekter, som hører til under loopet. Der er også brugt en if-else-kasse, som kan anvendes i situationer, hvor der gerne vil udtrykkes, at noget interaktion sker under en bestemt betingelse og noget andet interaktion sker under else-betingelsen.

Det næste eksempel på et designsekvensdiagram, som ses i figur 3.9, er tegnet så interaktionen mellem GUI-klasserne også er tegnet med. Det er altså et sekvensdiagram, der er tegnet mere på tværs af lagene, end det var tilfældet i figur 3.8. Det er et valg, hvor meget man tegner med i sit sekvensdiagram, det afhænger af, hvad man ønsker at fokusere på. Hvad man ønsker at fokusere på afhænger af, hvor der er noget kompleks interaktion, der er behov for at få designet eller vist gennem et designsekvensdiagram. Vi vælger som regel ikke at tegne GUI klasserne med i vores sekvensdiagrammer, da det ofte er den samme måde GUI'en interagerer med modellen på, og da det som regel ikke er så komplekst. Sekvensdiagrammer skal være overskuelige og give overblik, så man må altid tage nogle valg mht., hvor meget der tegnes med.



Figur 3.9 Designsekvensdiagram for use case "Enroll course" med GUI klasser.

Designsekvensdiagrammer kan som tidligere skrevet tegnes for en hel use case, dele af en use case eller blot et enkelt metodekald. Figur 3.10 viser hvordan det kan tegnes for et enkelt metodekald.



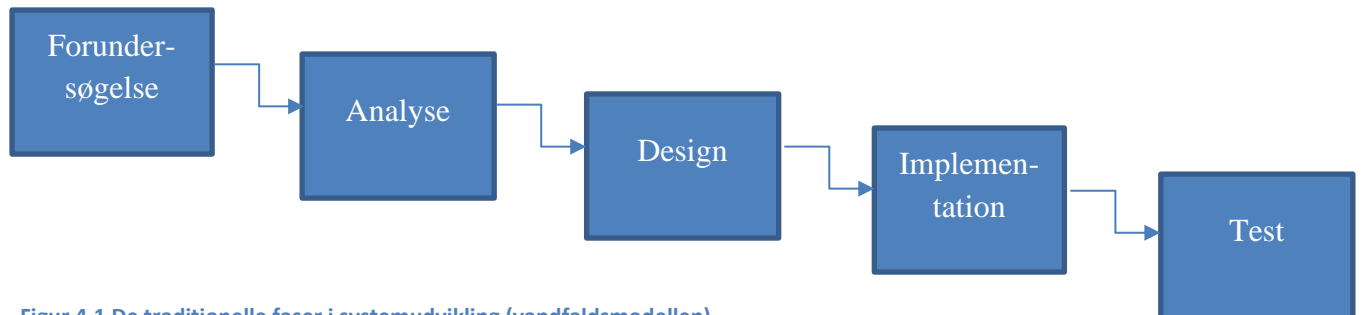
Figur 3.10 Designsekvensdiagram for kald af metoden ”getCourses()”

3.5 Opgaver

Opgave 3.1: Tegn et designsekvensdiagram for kald af metoden `averageGradeThisSubject()`.

4 Unified Process

Som vi skrev i starten, laver vi systemudvikling ved at gennemløbe en række faser forundersøgelse, analyse, design, implementering, test osv.



Figur 4.1 De traditionelle faser i systemudvikling (vandfaldsmodellen)

Enten gennemløbes fase efter fase eller også laves flere runder af gennemløb af faserne. Erfaringerne gennem årene har lært os, at det som regel er umuligt gennem f.eks. en stor analyse fase at få det fulde overblik over problemer og løsninger og dermed krav til indhold og funktionalitet.

Udviklerne er mennesker med begrænsede evner til at forstå og gennemskue alt i et nyt problemområde fra starten af. Brugere er mennesker, som ikke altid er klar over deres problemer og behov og derudover ofte har svært ved at udtrykke deres behov og ønsker. Især kan brugerne have svært ved konkret at udtrykke, hvordan de tænker sig systemet, da de måske ikke kender de tekniske muligheder og har ordforrådet til at udtrykke det. Brugere og udviklere er i de fleste systemudviklingssituationer i starten begrænsede af manglende viden omkring hinandens områder. Det at opnå tilstrækkelig viden tager tid og samarbejde mellem begge parter. Derfor er man i de fleste systemsituationer gået væk fra at forsøge at definere hele systemet fra starten af.

I dag arbejder man derfor ofte ikke i systemudviklingsforløb, hvor hele faser gennemløbes en for en. Man arbejder i dag ofte med forløb, hvor der laves flere runder af gennemløb af faserne eller noget der svarer til faserne. Afhængig af metode så kaldes tingene noget forskelligt. Vi vil i det følgende introducere systemudviklingsmetoden Unified Process, der er en systemudviklingsmetode, som er karakteriseret ved at der laves flere gennemløb af ”faserne”.

Inden vi beskriver metoden, vil vi lige introducere nogle grundlæggende begreber. Måden at arbejde med systemudvikling på kan beskrives på flere niveauer. Den kan beskrives som en model, som en metode eller som begge dele. Model og metode er to forskellige beskrivelsesniveauer og definitionen på begreberne er som følger.

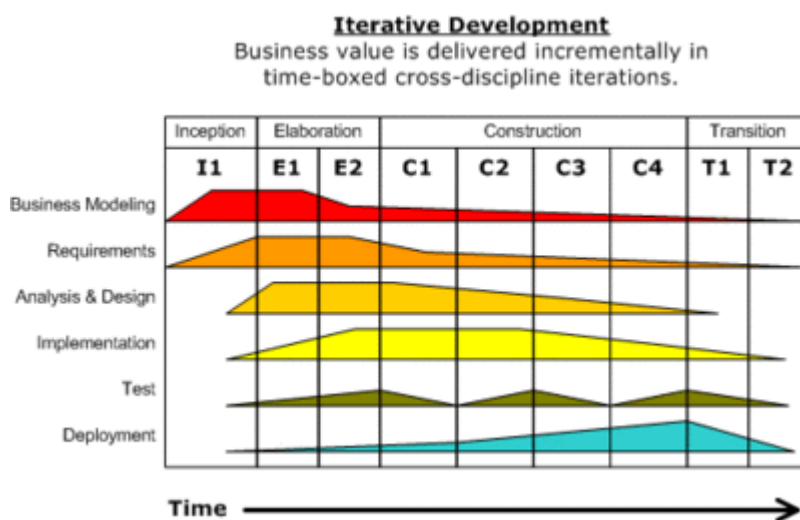
Def. model: Overordnet beskrivelse af en fremgangsmåde

En model illustreres ofte med en tegning.

Def. metode: Detaljeret beskrevet fremgangsmåde

En metode beskriver i detaljer, hvilke aktiviteter, rækkefølgen på aktiviteterne og hvilke teknikker der bruges hvornår.

Unified Process er en systemudviklingsmetode, der både er beskrevet som model og metode. På figur 4.2 kan Unified Process ses beskrevet som model.



Figur 4.2 Unified Process beskrevet som model

Unified Process er beskrevet med 4 faser Inception, Elaboration, Construction og Transition, som udtrykker, hvor langt man er i udviklingsforløbet fra første idé om nogle problemer i organisationen, der kan løses med et computer system til systemet er implementeret og afleveret til organisationen. Faserne i Unified Process må ikke forveksles med de faser, vi tidligere har beskrevet i den traditionelle udviklingsmodel, faserne i Unified Process er noget andet.

I hver fase arbejder man i et vilkårligt antal iterationer afhængig af projektets størrelse. På figuren er iterationerne benævnt som I1, E1, E2, C1 osv. I hver iteration, der skal betragtes som en gennemførelse af et udviklingsforløb, der udvikler en **del** af systemet, gennemløbes et antal discipliner.

Disciplinerne er her Business Modeling, Requirements, Analysis og Design, Implementation, Test og Deployment. Afhængig af hvilken iteration og i hvilken fase man er i, er disciplinerne vægtet lidt forskelligt. F.eks. laves der mere business modeling i en iteration i Inception

fasen end der gøres i en iteration i Construction fasen. Tilsvarende laves der mere implementation i en iteration i Construction fasen end i en iteration i Inception fasen. Der kan dog i princippet blive udført aktiviteter inden for alle discipliner uanset hvor i faserne, man er. For eksempel kan der i de første faser sagtens laves en kodet prototype, og derved kan der også ske noget implementation her. Disciplinerne udføres altså med forskellig vægt i forhold til, hvilken iteration i hvilken fase, man er i. Vægten af disciplinerne er illustreret med de farvede bølger i figur 4.2 ovenfor.

Unified Process er ikke kun beskrevet som model men også som metode. Før vi kommer ind på den mere detaljerede beskrivelse af metoden, vil vi beskrive de antagelser, der ligger til grund for metoden.

De grundlæggende antagelser er udsprunget af de erfaringer, man i praksis gennem tiden har fået.

Unified Process er altså designet til at forstærke kendte best practices. For eksempel har man fundet ud af, at det at arbejde iterativt ofte er en bedre praksis end at arbejde i hele faser. Punkterne nedenfor er udtryk for de best practices, man har bygget Unified Process op omkring.

- **Udvikl iterativt**
- **Definer og styr krav**
- **Brug komponent arkitektur**
- **Skab visuelle modeller**
- **Verificer kvaliteten**
- **Styr ændringer**

Ud over at **udvikle iterativt**, som vi har snakket lidt om tidligere, så siger de altså, at det er en god praksis at definere og styre på krav. Det at **definere nogle gode og konkrete krav**, er en god praksis i forhold til at få beskrevet, hvad systemet skal indeholde og bruges til. I Unified Process beskrives krav gennem use cases. Er use casene gode og konkret beskrevet, så er de også gode at planlægge med og styre efter i udviklingsprocessen, da de er lettere at estimere, lettere at tildele udviklere, lettere at teste osv. Den bedste praksis mht. at definere krav har vist sig at være, at arbejde iterativt og lade kravene komme til og udvikle sig løbende.

Man har også erfaret, at det er god praksis at definere, hvad der er de store dele i systemet og deres indbyrdes sammenhænge. Det er vigtigt at de store dele også kaldet komponenterne er defineret med hver deres tydelige ansvar, f.eks. en komponent med ansvar for GUI, en komponent med ansvar for modellen osv., og at de hænger sammen på en måde, der giver fleksibilitet, dvs. gør det muligt forholdsvist let at lave ændringer i. Det er specielt vigtigt med fleksibilitet i ens arkitektur, når man udvikler iterativt, da man ofte eller rettere hele tiden vil

komme til at ændre i allerede eksisterende kode. Trelagsmodellen, som vi tidligere har beskrevet, er et eksempel på en **komponentarkitektur** med en vis fleksibilitet.

Man har også erfaret, at det er en god praksis at skabe analysen og designets resultater som nogle **visuelle modeller**. Med visuelle modeller menes modeller som use case diagrammer, klassediagrammer og alle de andre UML modeller, I har lært om tidligere. UML modellerne er gode for udviklerne, til at blive enige om det mere detaljerede indhold og design af systemet. Nogle af dem kan imidlertid være svære for brugeren at gennemskue og forholde sig til, så et møde med brugeren, hvor der diskuteres UML modeller, vil nok være af begrænset værdi. Prototyper er også modeller. Prototyper er mere visuelle og arbejdende modeller af et system. F. eks. kan en prototype være en skitse af nogle skærbilleder på papir eller lavet i et eller andet værktøj, hvor det er let at skitsere et skærbillede og evt. på overfladen simulere noget funktionalitet. Prototyper er til forskel for UML modeller gode at snakke med brugere om og kan være et vigtigt værktøj i forhold til at komme nærmere en forståelse af hinandens verdener. Prototyperne er altså gode i processen med at få fundet og defineret gode konkrete krav og ønsker. Unified Process lægger vægt på at både UML modeller og prototyper er vigtige og anvendelige modeller til at beskrive, hvad der tænkes omkring indhold og funktionalitet i systemet.

Man har også erfaret, at det er godt at **verificere kvaliteten**, ja faktisk menes der, at det er en god idé **ofte** at verificere kvaliteten af systemet. Ved at arbejde i iterationer bliver det netop muligt **ofte** at verificere på kvaliteten, da man typisk ved en iterations afslutning vil gennemføre en form for test til verificering af kvaliteten. Testen kan både være intern og ekstern. Den interne går på, at teste om der er lavet god fejlfri kode og den eksterne går på at teste om systemet fungerer, som aftalt med brugeren.

Styr på ændringer er en praksis, der handler om, at man er bevidst om, at det at arbejde iterativt vil give løbende ændringer, som der planlægningsmæssigt skal tages hånd om. Ud over at den iterative måde at arbejde på, kan medføre ændringer, så er det faktisk også den iterative måde at arbejde på, der gør det muligt at håndtere ændringerne løbende. For hver iteration laves der nemlig en plan, og i planen for den eller de næste iterationer er det muligt at prioritere ændringer ind.

Unified Process er altså bygget op omkring de ovenfor beskrevne best practices og har defineret nogle grundlæggende antagelser, som reflekterer disse. De grundlæggende antagelser for metoden Unified Process er som flg.

- **Use case** og **risiko** drevet
- **Arkitekturcentreret**
- **Iterativ** og **inkrementel**

- **Unified Process** er en **systemudviklingsmetode**, der anviser hvornår og hvordan de forskellige UML-teknikker bruges i den objekt-orienterede analyse og design

At metoden siges at være **use case drevet** betyder, at det er use cases, man bruger til sammen med brugeren at beskrive krav til systemet med. Det er use cases, man prioriterer og planlægger med i planlægningen af iterationerne. Det er use casene, man tester for at vurdere om systemet fungerer rigtigt. Use casene er udgangspunktet for alt i metoden, det er dem, der **dri-ver** udviklingen.

At metoden er **risiko drevet** betyder, at man i prioriteringen af use casene kigger på, hvilken risiko, der er i dem. Er de risikofyldte fordi de er teknisk udfordrende eller uafklarede, fordi de er meget centrale og værdifulde dvs. afgørende for systemets succes, fordi der er store krav til brugervenlighed eller andet. Use cases med høj risiko prioriteres højt og skal helst tages hånd om først i udviklingsforløbet. Hver gang en ny iteration planlægges, overvejer man, hvilke af de manglende use cases, der er mest risiko i. Dem med mest risiko i tages med i iterationen.

I Unified Process arbejdes der også med risiko på et andet niveau. Der laves løbende analyse af risici i forhold til selve projektet og processen heri. Det kan f.eks. være risici ved de anvendte udviklingsteknologier, risici i forhold til bemanning, risici ved udviklingsplaner, risici i forhold til markedet osv. En risikoanalyse på dette niveau laves for at undgå, at man bliver overrasket over hændelser, der kan true processen. Man prøver at være på forkant med hændelsen, så den, hvis den indtræffer, hurtigt kan håndteres og udviklingsprocessen fortsætte. Til afdækning og vurdering af alvoren af de forskellige risici, der kan true processen, kan følgende model anbefales.

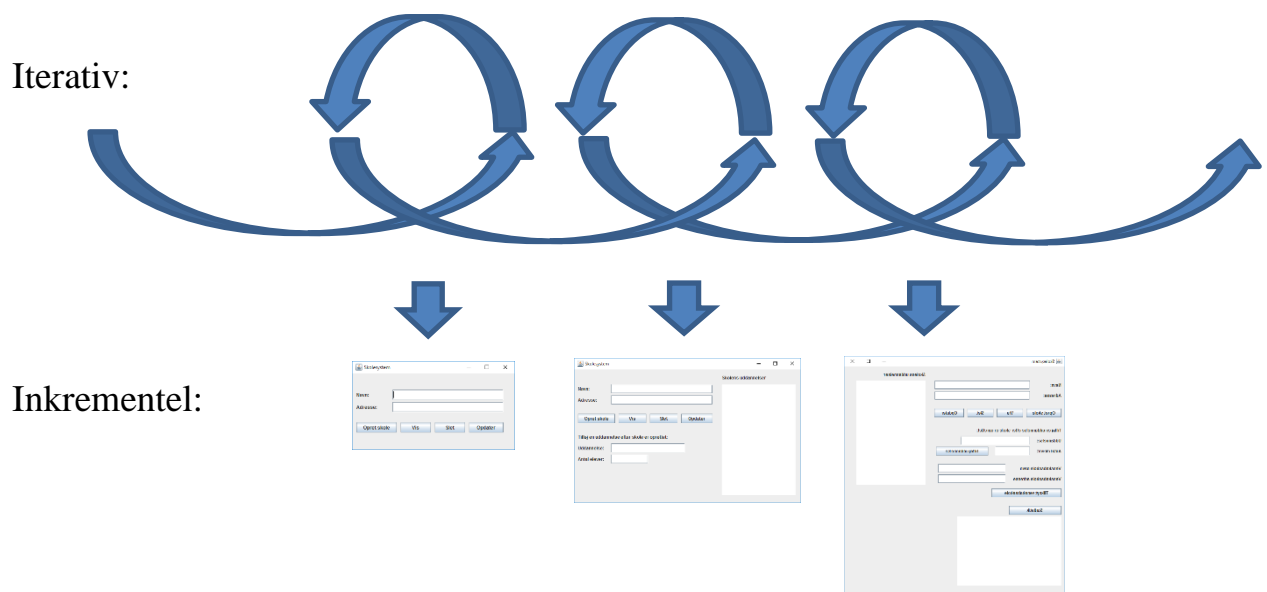
Risikoelement(beskrivelse)	Sandsynlighed i %	Alvor (en værdi 1-10 med 10 som alvorligst)	Risikotal = Sandsynlighed * Alvor (Højt er f.eks. over 300)
Netværksnedbrud	30	5	150
Vores kodeekspert i gruppen forsvinder undervejs	40	8	320
Osv.			

Figur 4.3 Tabel til risikoanalyse

Nogle gange blandes vilkår og risiko sammen. Et vilkår er almindeligt forekommende, og det vil man kunne planlægge og arbejde sig ud af, så processen forløber rimeligt alligevel. Et eksempel på et vilkår, som man ikke vil tage med i en risikoanalyse er flg. ”Enkelte dages sygdom i gruppen”. Et eksempel på en risiko som til gengæld godt kan tages med i en risikoanalyse er flg. ”Langtidssygdom i gruppen”.

En anden antagelse er at metoden er **arkitekturcentreret**. At være arkitekturcentreret handler om, at det er vigtigt med fleksibel arkitektur, for hele tiden at kunne implementere ændringer. Dette er tidligere beskrevet i forbindelse med den best practice, der handler om brug af komponentarkitektur.

Begreberne **iterativ** og **inkrementel** er til dels også beskrevet tidligere. Specielt det, at det er god praksis at arbejde iterativt. Det at arbejde iterativt går på processen, nemlig at arbejde på en del af systemet ad gangen. Det der ligger i begrebet inkrementel går mere på produktet. Inkrementel betyder at produktet gradvist vokser. Det vokser med den del, der er arbejdet på i iterationen, se figur 4.4.



Figur 4.4 Begreberne iterativ og inkrementel illustreret

Vi har nu beskrevet lidt om de grundlæggende antagelser bag Unified Proces. I det følgende vil vi gå mere ind i metoden og beskrive mål og aktiviteter i metodens faser og discipliner.

4.1 Faserne

De fire faser i metoden er som tidligere nævnt Inception, Elaboration, Construction og Transition. Faserne beskrives i det følgende en for en.



4.1.1 Mål for Inception

Det primære mål for fasen Inception er at fastlægge rammerne for projektet. De økonomiske rammer, tidsmæssige rammer, personalemæssige rammer osv. Der skal ske en vurdering af hvilke problemer der er i organisationen samt hvilken løsning på problemerne, der vil være den rigtige og mest rentable at arbejde videre med.

Flg. aktiviteter hører til i fasen Inception:

- Få afdækket og **beskrevet problemer** i problemområder. Få beskrevet **visioner** for løsning af problemerne
- Beskrive **arbejdsprocesserne**
- Indfange **essentielle krav** (10-20 % use cases) til løsningen.
- Evt. udarbejde prototype – evt. **validere tekniske krav** (teknisk prototype), finde frem til **krav til systemet** (smid-væk-prototyper)
- Lave **cost/benefit** for løsning evt. vurderet i business case
- Lave **risikoanalyse**
- Definere **afgrænsning** - risikoområder afprøvet i prototyper.
- **Etablere projektet**, beslutte proces, lave en initial projektplan, etablere team osv.

4.1.2 Mål for Elaboration

Det primære mål for fasen Elaboration er at komme godt i gang med opbygningen af systemet på en måde, så udviklingen af det endelige system kan ske indenfor de givne ramme.

Flg. aktiviteter hører til i fasen Elaboration:

- Designe **grundarkitektur** og udvikle **grundsystem** (ofte beskrevet med et klassediagram)
- **Forfine krav** og **udbygge use case diagram og beskrivelser til 80 %**
- **Definere kvalitetskrav**, kan beskrives som ikke funktionelle krav
- **Forfine risikoanalysen**, evt. er der dukket nogle nye risici op
- Udarbejde **detaljeret plan for construction fasen**

4.1.3 Mål for Construction

Det primære mål for fasen Construction er at opbygge et system, der er færdigudviklet til en betatest.

Flg. aktiviteter hører til i fasen Construction:



- Færdiggøre detaljering af krav.
- **Færdiggøre modellerne fra analyse og design**
- **Kode systemet færdigt**

4.1.4 Mål for Transition

Det primære mål for fasen Transition er at fremlægge det færdige system for kunder og brugere.

Flg. aktiviteter hører til i fasen Transition:

- **Forberede installation** hos rekvirent
- Tilpasse system til installering
- Udarbejde **brugerdokumentation**
- Gennemføre **brugeruddannelse**
- Afklare og beskrive **driftsprocedure**
- Gennemføre **betatest** – accepttest
- Fejlretning
- **Sætte systemet i drift**

Det gælder for alle faserne, at målet for fasen opnås ved pr. fase at planlægge og gennemføre et antal iterationer. Aktiviteterne for fasen gennemføres også gennem de planlagte iterationer. Aktiviteterne gennemføres automatisk gennem de til metoden hørende og nedenfor beskrevne discipliner

4.2 Disciplinerne

I det følgende vil vi prøve at beskrive de mål, aktiviteter og produkter, der karakteriserer de enkelte discipliner. Disciplinerne kan sammenlignes med faserne beskrevet i den traditionelle model, her gennemløbes de bare flere gange, nemlig for hver iteration. Afhængig af hvilken iteration i hvilken fase, man er i, skal man gennemføre mere eller mindre af aktiviteterne.

4.2.1 Business Modeling

Det primære mål for disciplinen Business Modeling er at komme til at forstå forretningsområdet/domænet og problemerne heri. Det kan ses af figur 4.2, at aktiviteterne i Business Modeling mest foregår i de første iterationer i den første fase.

Flg. aktiviteter gennemføres i disciplinen Business Modeling:

- **Forundersøgelse/foranalyse** som afdækker problemerne og identificerer potentielle visioner for projekter, skabe forretningsmodeller og beskrive forretningsprocesser
- Sikre at både udviklere og brugere forstår, hvor og hvordan det nye system passer ind i forretningsprocesserne og brugerorganisationen
- En cost benefit analyse af projekterne
- Evt. lave prototyper

Flg. produkter leveres i denne disciplin

- Forretningsmodeller /problembeskrivelse / procesbeskrivelser / aktivitetsdiagrammer
- Systemvisioner
- Evt. prototyper

Denne note har ikke medtaget teori omkring udarbejdelsen af produkterne i denne fase. Noten her fokuserer mere på teorien omkring udarbejdelsen af de produkter, der hører det egentlige udviklingsprojekt til. Den fokuserer altså mere på de produkter, der udarbejdes i de øvrige discipliner f.eks. de forskellige UML diagrammer, som er centrale i Unified Process.

4.2.2 Requirements

Det primære mål for disciplinen Requirements er at få indsamlet kravene til det system, der er besluttet skal laves. Der er både tale om de funktionelle krav, som forstås og beskrives gennem use cases og de ikke-funktionelle krav, som tekstuelt beskrives.

Flg. aktiviteter gennemføres i disciplinen Requirements:

- Indsamle detaljeret information om problemområdet gennem interview, workshops, observation osv.
- Definere funktionelle krav (use cases) og ikke funktionelle krav (kravliste)
- Udvikle dialogen og brugergrænsefladen
- Evaluere krav med brugerne

Flg. produkter leveres i denne disciplin

- Use case model, dvs. use case diagram og korte use case beskrivelser for centrale use cases
- Kravliste (ikke funktionelle)
- Prototype af brugergrænsefladen
- Ordliste



Requirements aktiviteterne kan forekomme mere eller mindre i alle faserne jvf. Figur 4.2, men er specielt fremtrædende i inception og elaboration faserne.

4.2.3 Analysis & Design

Det primære mål for disciplinen Analyse og Design er ud fra use case modellen at få opbygget analyse og design modellen. Det handler om at finde frem til de ting, begreber og sammenhænge, der skal registreres i systemet, så use casene kan gennemføres.

Flg. aktiviteter gennemføres i disciplinen Analyse og Design:

- Analyse af problemområdet dvs. analyse af hvad man skal kunne med systemet, hvordan interaktionen med systemet skal være, hvad det skal indeholde osv
- Designe arkitektur (web, netværk osv.) og deployment environment (client server)
- Designe arkitekturen dvs. hvad skal systemets store dele være og hvordan skal de arbejde sammen
- Designe hvordan use cases realiseres, dvs. designe hvordan de realiseres med de valgte klasser og sammenhænge.
- Designe databasen, hvilke tabeller og sammenhænge
- Designe brugergrænsefladen
- Design sikkerhed og kontroller i systemet

Flg. produkter leveres i denne disciplin

- Analyse model dvs. analyse klassediagram, adfærdsmønstre, systemsekvensdiagrammer
- Client server-/netværks-/web-arkitektur
- Design model dvs. design klassediagram og packagediagram (grundarkitektur)
- Use case realiseringer (designsekvensdiagrammer)
- ER-diagrammer
- Brugergrænseflade illustreret i prototype værktøj, skitseret på papir eller andet.

Analyse og design aktiviteterne kan forekomme mere eller mindre i alle faserne jvf. Figur 4.2, men er specielt fremtrædende i elaboration fasen.



4.2.4 Implementation

Det primære mål for disciplinen Implementation er at bygge systemkomponenterne, dvs. kode systemets dele.

Flg. aktiviteter gennemføres i disciplinen Implementation:

- Programmere klasser og metoder i de forskellige lag i arkitekturen, programmer sammenhænge, programmer brugergrænsefladen osv.
- Inkorporere genbrugbare klasser fra diverse klassebiblioteker
- Integrere klasser og delsystemer til kørende system

Flg. produkter leveres i denne disciplin

- Implementeringsmodel dvs. softwarekomponenter, filer, Dynamic Link Libraries (DLL'er), Services osv.
- Systemkomponenter

Implementation aktiviteterne kan forekomme mere eller mindre i alle faserne jvf. Figur 4.2, men er specielt fremtrædende i elaboration og construction faserne.

4.2.5 Test

Det primære mål for Test er at planlægge og gennemføre tests, så vi opnår **kendt** kvalitet.

Flg. aktiviteter gennemføres i disciplinen Test:

- Planlægge og udføre unit test
- Planlægge og udføre integrationstest
- Planlægge og udføre brugervenlighedstest
- Planlægge og udføre brugeraccepttest

Flg. produkter leveres i denne disciplin

- Test model bestående af testplan, testaktiviteter, tekstuelle testcases, kodede testcases osv.
- Resultatet af gennemførte interne og eksterne tests

Testaktiviteterne kan forekomme mere eller mindre i alle faserne jvf. Figur 4.2.

4.2.6 Deployment

Det primære mål for disciplinen Deployment er at planlægge og gennemføre en installering af systemet på div. servere computere, uddanne brugere osv., så systemet bliver operationelt.

Flg. aktiviteter gennemføres i disciplinen Deployment:

- Erhverve hardware og system software
- Installere komponenter
- Træne brugere i det nye system
- Konvertere og initialisere data

Flg. produkter leveres i denne disciplin

- Deployment model dvs. model over systemets omgivelser mht. den fysiske afvikling, model af processer og proceselementer osv.
- På div. servere, computere osv. et installeret system
- Div. brugervejledninger

Deployment aktiviteterne er mest fremtrædende i transition fasen.



4.3 Anvendelse af Unified Process

I afsnittene ovenfor er metodens grundlæggende antagelser, dens faser, dens discipliner og de tilhørende aktiviteter beskrevet. Trods disse mange beskrivelser, kan det være svært at gennemskue, hvordan man i praksis kommer i gang med at arbejde efter metoden. Det flg. afsnit vil derfor forsøge at beskrive, hvordan man i praksis kommer i gang med et udviklingsforløb med Unified Process. Projektet skal etableres på baggrund af valget af Unified Process som systemudviklingsmetode. Til organisering af projektet hører at etablere en projektgruppe med de kompetencer og roller, der er relevante for at arbejde med en proces efter denne metode. Relevante **roller** i Unified Process kan være flg.:

- Projektleder
- Systemanalytiker
- Softwarearkitekt
- Testansvarlig
- Metodeansvarlig osv.

Projektlederen er den, der styrer og leder projektet og dens deltagere. Evt. har projektlederen også ansvar for kommunikation med omgivelserne.

Systemanalytikeren er den, der har ansvar, interesse og evt. særlige evner for de mange analyse aktiviteter og modeller, der er i Unified Proces.

Softwarearkitekten har ansvar for arkitekturen, som jo er meget vigtig, når der arbejdes iterativt.

Den **testansvarlige** har ansvar for, at de forskellige tests bliver planlagt og gennemført, så kvaliteten bliver tjekket, og man får den nødvendige respons fra brugeren.

Den **metodeansvarlige** har ansvaret for at Unified Proces bliver fulgt og evt. tilpasset i forhold til den konkrete situation. En del af etableringen er også at aftale, hvor og hvornår vi arbejder, dvs. indrettes der et arbejdsrum, så alle sidder sammen, møder vi og går på samme tid, eller er der en form for fleks- og fixtid osv. Når gruppen er etableret, skal der udarbejdes en projektplan. Projektplanen er meget central i forhold til det at kunne styre en systemudviklingsproces. En projektplan kan laves på flere niveauer. Der er nedenfor vist et eksempel på en overordnet projektplan, der er lavet i starten af et systemudviklingsforløb. I planen bruges casen fra tidligere afsnit om udvikling af et skoleadministrationssystem.

4.3.1 Overordnet projektplan

En projektplan på overordnet niveau kan når Unified Proces anvendes være en plan over de fire faser, være god at få aftalt med brugeren. Det centrale i en overordnet projektplan at få dokumenteret en start og slut på projektet samt hvor lang tid de enkelte faser vil tage. Der kan

bruges forskellige estimeringsteknikker til at estimering af projektets størrelse eller tidshorisont. Teknikker der kan bruges i denne sammenhæng kan f.eks. være teknikkerne Trepunkts-estimering, Use case points, Cocomo estimering osv. Teknikkerne vil vi dog ikke komme nærmere ind på i denne note, men man kan f.eks. læse om dem i Stephen Bierings-Sørensens bog ”Praktisk IT-projektledelse”.

Eksempel på meget overordnet projektplan:

Faselinje 0: Start projekt - 05/09-2018

Metode Unified Process er valgt

Faselinje 1: Slut Inception - 16/09-2018

Interessenter er enige og har valgt en vision.

Faselinje 2: Slut Elaboration - 30/09-2018

Krav og use case dokumenter opdateret

Grundarkitektur er valgt

Risikoanalyse opdateret

De mest værdifulde og mest risikofyldte use cases er kodet og testet i testmiljøet

Faselinje 3: Slut Construction - 14/10-2018

Systemets funktionalitet og kvalitet er kodet og testet i testmiljøet

Faselinje 4: Slut Transition - 28/10-2018

Systemet, som beskrevet i visionen, er i drift og brugertest gennemført i produktionsmiljøet

Brugeruddannelse gennemført.

4.3.2 Planlægning af iterationerne – detaljeret projektplan

For at komme i gang med planlægningen af iterationerne er det nødvendigt at beslutte, hvor lange man tænker, det er relevant at iterationerne er i dette udviklingsforløb. En iteration er en **timebox**. En timebox er kendetegnet ved, at den har en **fast start** og **fast slut**. Start og slut ændres altså ikke, hvilket giver den fordel, at både kunde og udviklere kan regne med, at der leveres noget til kunden, når slut er nået. Til gengæld kan man ikke vide, hvad der leveres, det kan være mere eller mindre det forventede, da det vil være indholdet, der reguleres på, hvis udviklingsprocessen går langsommere eller hurtigere end forventet. Den faste start og

slut på iterationerne giver mange fordele for både udviklere og kunde. Det vil for begge parter være lettere at planlægge, sikre en rytme i udvikling og respons, give mere effektiv udvikling, give et bedre produkt osv.

Antallet og længden af iterationerne afhænger selvfølgelig af hvor stort et projekt, der er tale om, hvor lang tid, det er estimeret til at tage, og hvor kompliceret og usikkert projektet er. Er der tale om et lille simpelt projekt vil en iteration pr. fase nok være tilstrækkeligt. Er der tale om et projekt med lidt mere substans i kan man f.eks. vælge at lave en iteration for inception fasen, to iterationer for hhv. elaboration og construction fasen og en for transition fasen. Er der tale om et endnu større projekt evt. med mange usikkerheder og risici i forhold til teknologi, design, use cases osv. så kan man vælge at lave flere iterationer i de enkelte faser. Jo større usikkerheden er i projektet jo større er behovet for regelmæssigt at få respons fra brugeren, jo kortere bør iterationerne være og jo flere vil der blive. Længden på iterationerne i Unified Proces er typisk på 1-3 måneder, men kan med fordel gøres kortere, hvis usikkerheden er stor.

I eksemplet nedenfor har man valgt at første iteration, som udgør hele inceptionfasen, er to uger og de øvrige iterationer er en uge. Det er lidt korte iterationer i forhold til, at det er UP, der er valgt som metode, da de ofte med UP er længere. Længden af iterationerne kan dog altid tilpasses efter behov, f.eks. findes der en variant af UP med kortere iterationer, som kaldes AUP. AUP⁴ står for Agile Unified Process.

Selve udarbejdelsen af planen for iterationerne er en løbende proces. Et af formålene med at udvikle iterativt er at sikre et regelmæssigt samarbejde med brugeren, så alle bliver klogere på krav og mulige løsninger. Dette løbende samarbejde betyder, at der hele tiden kan komme nye krav, ændringer til eksisterende krav nye prioriteringer osv. Da der hele tiden kan komme nye krav, ændringer til eksisterende krav og nye prioriteringer giver det ikke mening i detaljer at planlægge mange iterationer frem. Det anbefales at man kun detaljeret planlægger to iterationer frem ad gangen.

Overordnet er indholdet af **planen** for nuværende iteration og efterfølgende iteration, som beskrevet nedenfor

- En plan for **nuværende iteration** for at kunne følge fremskridt i iterationen
 - Planen indeholder **aktiviteter** med **tid** og **ansvarlig**
 - Planen indeholder **vurderingskriterier** for iterationen og evt. **tjekpunkter** undervejs
- En plan for **næste iteration**. Planen for næste iteration påbegyndes mht. aktiviteter halvvejs i nuværende iteration og færdiggøres i starten af næste iteration.

⁴ https://en.wikipedia.org/wiki/Agile_Unified_Process

Projektplanen nedenfor er lavet med et til lejligheden konkret antal valgte iterationer, en i inception- og transitionfasen og 2 i hhv. elaboration- og constructionfasen. I planen kan man se, at kun en iteration er planlagt i detaljer. Hvad der bliver valgt som mål med den enkelte iteration bestemmes i starten af iterationen af kunden, brugeren og andre interessenter. Målet med iterationen kan være påvirket af resultatet af review og test af forrige iteration og de prioriteringer, de på baggrund af det ender med at have.

Når iterationen planlægges tages altså udgangspunkt i de af kundens prioriterede use cases, use casene er ofte prioriteret i forhold til hvor værdifulde eller risikofyldte de er. Målet med iterationen bliver at nå et **antal** af de højest prioriterede. Antallet, dvs. hvor mange use cases der kan nås i iterationen, findes ved at estimere mængden af ressourcer i form af den **indsats** et antal udviklere kan levere i iterationen og ved at estimere på størrelsen og kompleksiteten af use casene. Igennem iterationen udvikler use casene sig gennem arbejdet med kernerdisciplinerne i Unified Proces og systemet og arkitekturen udvikler sig i omfang.

Projektplanen nedenfor er lavet som et eksempel, der dækker over alle fire faser med eksempler på konkrete krav og use cases. Planen er, som det også er i praksis, kun detaljeret og konkret i første del af planen, resten af planen vil være mere overordnet beskrevet. Planen udvikler sig og bliver mere og mere detaljeret efterhånden, som man tager hul på iterationerne.

Blot for at illustrere, hvordan en plan **kan** se ud, er der nedenfor vist et eksempel på en mere **detaljeret projektplan** med **nogle mere eller mindre planlagte iterationer** for **Skoleadministrationsprojektet**.

Faselinje 0: Start Projekt - 5/9-2018

Metode UP er her valgt

Fase 1: Inception 5/9 – 16/9-2018

Iteration 1

Aktiviteter:

1. Etabler gruppe med roller – 1 dag Peter, Søren, Hanne, Margrethe
2. Beskriv problemer og påbegynd ordbog – 0,8 dag Hanne
3. Beskriv scope af systemet og hermed også afgrænsning – 0,2 dag Hanne
4. Beskriv central funktionalitet gennem krav og use cases samt lav en prioritering, De fleste use cases beskrives med den korte use case beskrivelse – 1 dag Margrethe
5. Lav evt. aktivitetsdiagrammer til beskrivelse af arbejdsprocesser og aktiviteter 1 dag Søren



6. Udarbejd risikoanalyse - 0,5 dag Peter
7. Beskriv teknologi – 0,5 dag Søren
8. Beskriv overordnet arkitektur – 0,7 dag Søren
9. Udarbejd evt. prototype til afprøvning af teknologi og afdækning af krav – 2 dage Peter, Søren
10. Udarbejd visioner og beskriv fordele og ulemper – 1 dag Hanne
11. Review og valg af vision – 1 dag Peter, Søren, Hanne, Margrethe
12. Lav en overordnet plan for de næste faser – 1 dag Margrethe

Faselinje 1.1: Slut iteration 1 - 16/9-2018

Visionerne er beskrevet mht. omfang, funktionalitet, økonomi, teknologi. Fordele og ulemper ved visionerne er beskrevet og de er klar til valg. Prioriteret kravliste. Use cases beskrevet med den korte use case beskrivelse.

Faselinje 1: Slut inception – 16/9-2018

Interessenter er enige og har valgt en vision.

Fase 2: Elaboration 16/9 – 30/9-2018

Iteration 2

Aktiviteter:

1. Etabler gruppe med evt. nye personer og roller
2. Lav detaljeret projektplan for iterationen
3. Udarbejd use case diagram med aktører
4. Beskriv use case UC3: Register student, UC4: Enroll course med den lange eller mellemlange use case beskrivelse
5. Dokumenter prioritering af designkriterier
6. Vælg arkitektur jf. designkriterier og beskriv den
7. Udarbejd analyse klassediagram til realisering af use casene UC3 og UC4
8. Udarbejd system sekvensdiagrammer, hvis use case er tilstrækkelig kompleks
9. Overvej adfærdsmønstre og lav dem, hvis der klasser med interessant adfærd
10. Lav designklassediagram til realisering af de valgte use cases
11. Lav designsekvensdiagram til realisering af de valgte use cases, hvis use case er tilstrækkelig kompleks
12. Design skærbillederne for UC3 og UC4
13. Kod UC3 og UC4
14. Påbegynd plan for iteration 3
15. Planlæg test ved at lave testcases til de forskellige testniveauer for de valgte use cases
16. Kod unit test
17. Planlæg test og review ved at vælge tid, sted, deltagere osv.



18. Gennemfør review
19. Gennemfør tests
20. Opdater dokumentation mht. krav og use cases, hvis nye er kommet til eller nogle skal fjernes. Lav evt. krydstabel. Prioriter krav og use cases.
21. Vurder risikoanalyse

Faselinje 2.1: Slut iteration 2 – 23/9-2018

Test gennemført og dokumenteret på UC3 og UC4. Dokument med krav og use cases er opdateret.

Iteration 3

Aktiviteter:

1. Lav detaljeret projektplan for iterationen, vælg de use cases, der er højest prioriterede og passer til den estimerede størrelse af iterationen.
2. Beskriv de valgte use cases UCX, UCY osv.
3. Udvid analyse klassediagram til realisering af use casene
4. Udarbejd system sekvensdiagrammer, hvis use case er tilstrækkelig kompleks
5. Overvej adfærdsmønstre og lav dem, er der klasser med interessant adfærd
6. Udvid designklassediagram til realisering af use casene
7. Lav designsekvensdiagram til realisering af use casene, hvis use casene er tilstrækkeligt komplekse
8. Design skærbillederne for UCX, UCY osv.
9. Kod UCX, UCY osv.
10. Planlæg test ved at lave testcases til de forskellige testniveauer for de valgte use cases
11. Kode unit test
12. Planlæg test og review ved at vælge tid, sted, deltagere osv.
13. Gennemfør review
14. Gennemfør tests
15. Opdater dokumentation mht. krav og use cases, hvis nye er kommet til eller nogle skal fjernes. Opdater evt. krydstabel. Prioriter use cases.
16. Vurder risikoanalyse

Faselinje 2.2: Slut iteration 3 – 30/09-2018

Test gennemført og dokumenteret på UCX, UCY osv. Dokument med krav og use cases er opdateret.



Faselinje 2: Slut elaboration - 30/09-2018

Central højt prioriteret funktionalitet dvs. UC3, UC4 osv. er kodet og testet i testmiljøet.

Krav og use case dokumenter opdateret

Risikoanalyse opdateret

Fase 3: Construction 30/09 – 14/10-2018

Iteration 4

Aktiviteter:

1. Lav detaljeret projektplan for iterationen
2. Udvid analyse og designklassediagram til realisering af evt. nye højest prioriterede use cases, der evt. er kommet til i de forrige iterationer
3. Realiser use cases, der vedligeholder data, f.eks. div. RUD use cases
4. Lav designsekvensdiagram til realisering af use casene, hvis use casene er tilstrækkelige komplekse
5. Opdater design og forfin skærbillederne
6. Kod use casene
7. Planlæg test ved at lave testcases til de forskellige testniveauer for use casene
8. Kod unit test
9. Planlæg test og review ved at vælge tid, sted, deltagere osv.
10. Gennemfør review
11. Gennemfør tests
12. Opdater dokumentation mht. krav og use cases, hvis nye er kommet til eller nogle skal fjernes. Opdater evt. krydstabel.
13. Vurder risikoanalyse
14. Påbegynd deploymentplan.

Faselinje 3.1: Slut iteration 4 – 7/10-2018

Osv.

Iteration 5

Aktiviteter:

1. osv.



Faselinje 3.2: Slut iteration 5 - 14/10-2018

Osv.

Faselinje 3: Slut construction - 14/10-2018

Systemets funktionalitet og kvalitet er kodet og testet i testmiljøet

Fase 4 Transition 14/10 – 28/10-2018

Aktiviteter:

1. Lav detaljeret projektplan for installering mht. hardware og software
2. Kodet af systemet, der skal til for at få det kan installeres
3. Planlæg test af installeringsprocedure
4. Test installeringsprocedure
5. Lav en brugervejledning
6. Planlæg brugeruddannelse
7. Planlæg accept og brugertest
8. Installer systemet mht. både hardware og software
9. Gennemfør accept og brugertest
10. Gennemfør brugeruddannelse
11. Lav support og vedligeholdelsesplan

Faselinje 4: Slut transition – 28/10-2018

Systemet, som beskrevet i visionen, er i drift og brugertest gennemført i produktionsmiljøet

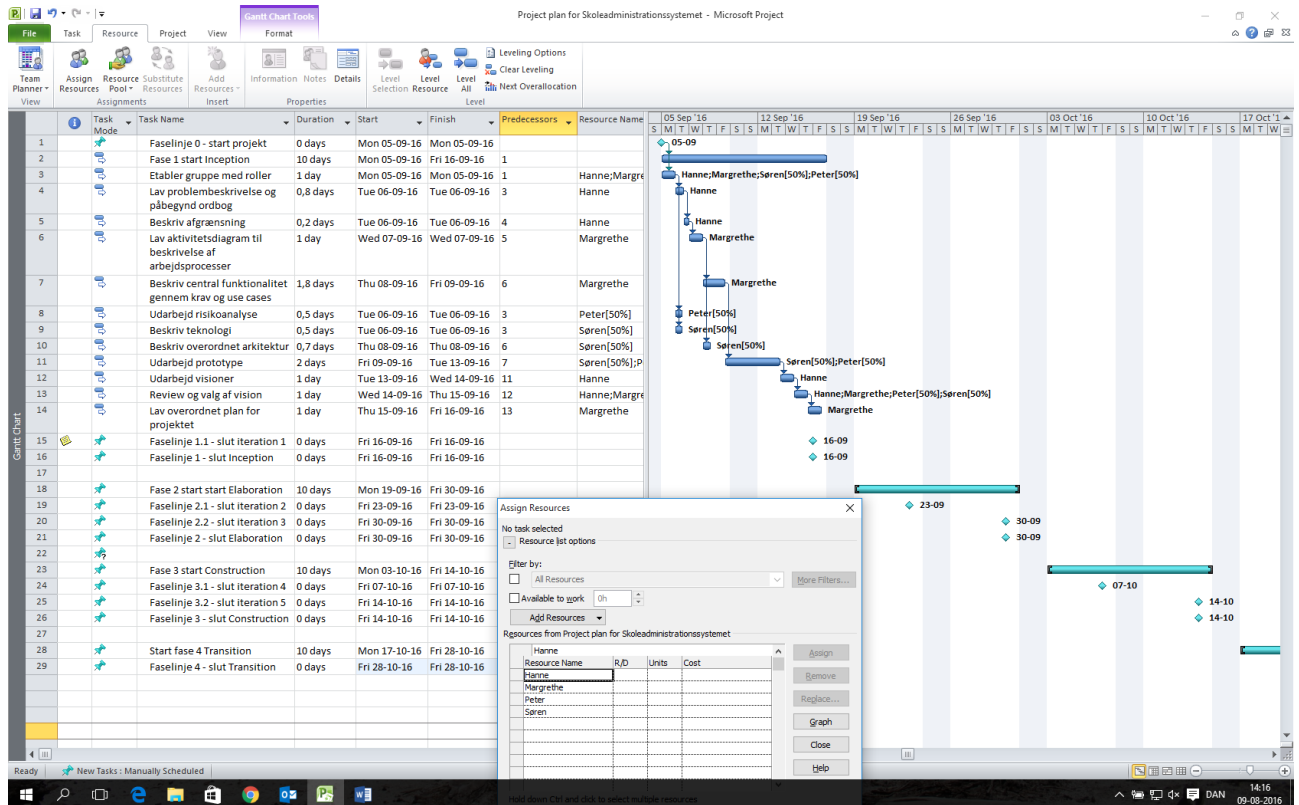
Brugeruddannelse gennemført.

4.3.3 Planer og værktøjer

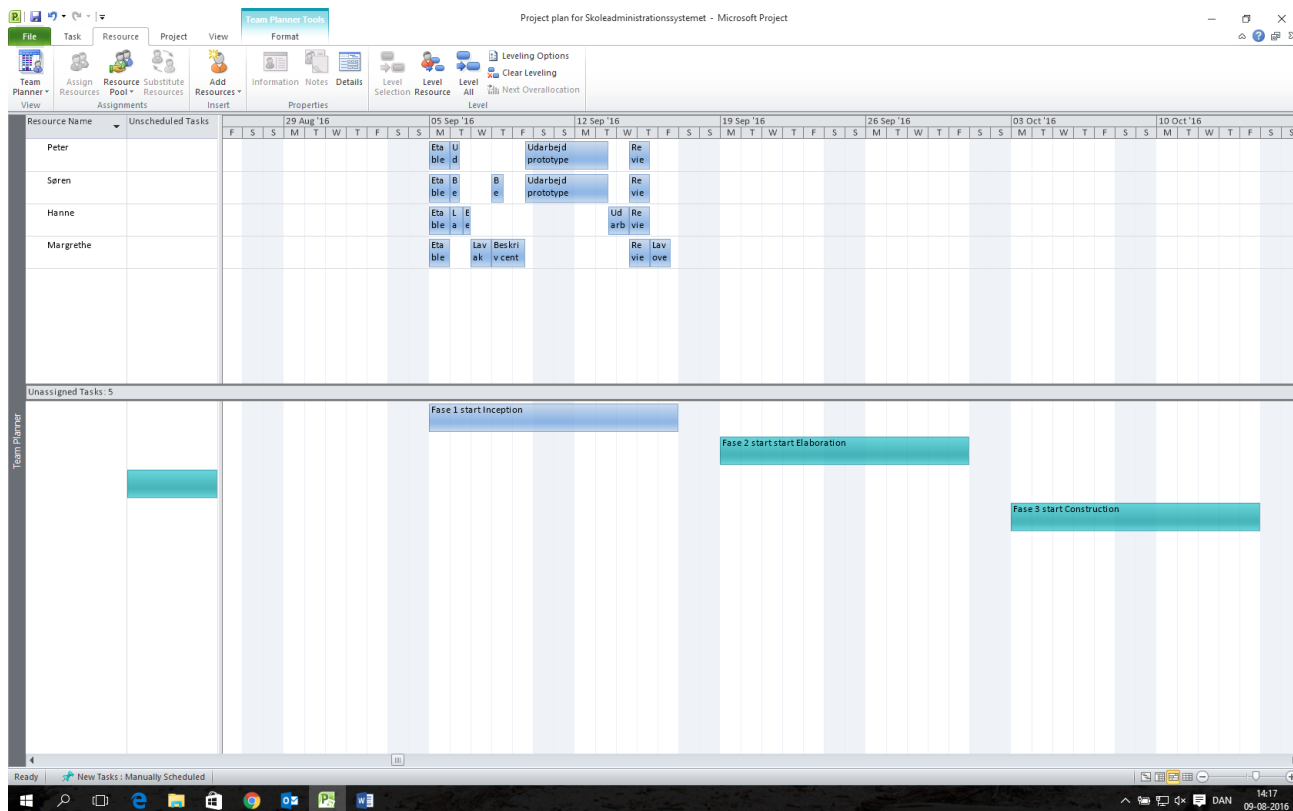
En iterationsplan er ret detaljeret og kan derfor med fordel laves som et Gantt-skema i et eller andet projektstyringværktøj. Der findes mange forskellige både gratis og professionelle projektstyringsværktøjer. Det mest kendte er nok Microsofts "Project". Nedenfor kan ses et eksempel på noget af planen lavet i Project.

Et Gantt-skema illustrerer datoer for start- og sluttidspunkt for opgaverne i projektet og viser et resumé af projektet. Nogle Gantt-skemaer viser også afhængigheder og dermed strukturen og rækkefølgen af opgaverne. Fordelen med de elektroniske værktøjer er, at der på baggrund af de indtastede data kan genereres forskellige brugbare grafer og oversigter, se figur 4.5 og

4.6. Bagdelen med planer i elektroniske værktøjer er til gengæld, at planerne ikke er så synlige, de er godt gemt inde i et system.



Figur 4.5 Gantt-skema svarende til planen beskrevet ovenfor



Figur 4.6 Plan over de enkelte projektgruppemedlemmers aktiviteter



5 Test

5.1 Indledning og formål med test

I test workflowet i Unified Proces og i test generelt handler det om at få verificeret om resultatet af implementationen lever op til det specificerede eller på anden måde planlagte. Det handler om at få testet om kvaliteten af software produktet er på det niveau, man har bestemt sig til, det skal være på. Der kan laves mange forskellige typer fejl i et systemudviklingsforløb. Der kan laves fejl i processen med at finde frem til, hvad det rigtige system er. Det rigtige system skulle gerne leve op til kundens opstillede krav mht. funktionalitet, data og kvalitet. Undervejs i systemudviklingsprocessen kan vi validere, om vi er på vej mod det rigtige system. Der ligger en del validering i at vælge en passende systemudviklingsmetode. En systemudviklingsmetode med et passende niveau af brug af analyse- og designteknikker, specifikationer, review, brugersamarbejde osv. Der kan også laves fejl i processen med at kode systemet. Når systemet er kodet færdigt, verificeres det, om det med passende nøjagtighed lever op til det specificerede og de opstillede kvalitetsmål. Det kan være kvalitetsmål mht. f.eks. korrekthed, robusthed osv.

Der er altså to begreber i spil i kvalitetssikringen af ens system, nemlig

- Validering – laves det rigtige system, laves det system brugerne gerne vil have og
- Verifikation – laves systemet rigtigt, opfører systemet sig, som vi forventer

Mange af UML-teknikkerne samt UP, som metode beskrevet i de foregående kapitler, kan bruges til at sikre et vist niveau af validering i vores systemudviklingsproces. Der kan dog med fordel også suppleres med forskellige former for eksperimenter, review og inspektioner internt og eksternt. Begrebet validering har vi altså tidligere i noten, introduceret teknikker til at opfylde. I forhold til begrebet verifikation mangler vi dog noget. Verifikation er test af, om det system, vi har lavet, opfører sig, som vi gerne vil have det, f.eks. at systemet ikke pga. programmeringsfejl går ned hele tiden eller regner forkert. Dette kapitel vil omhandle teknikker til verifikation af systemer.

5.2 Test generelt

Test er en integreret del af de fleste forskellige systemudviklingsmodeller og metoder. I hver af de flg. generelle systemudviklingsmodeller

- sekventiel (f.eks. vandfaldsmodel)
- iterativ/inkrementel (f.eks. Unified Proces, nogle agile metoder eks. SCRUM)
- eksperimentel (f.eks. nogle agile metoder, f.eks. XP)

er test tænkt ind, men på forskellig måde og med hver sin rolle i forhold til udviklingen af og kvalitetssikringen af softwaren. I dette kapitel kommer vi både til at tale om test generelt og om test i kontekst af Unified Proces. Emnerne vi vil komme ind på er flg.

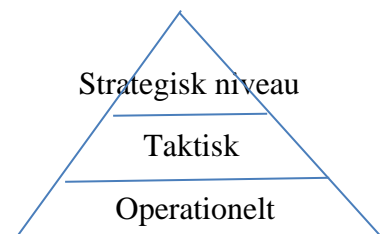
- de forskellige typiske testniveauer
- basis testteknikkerne, dvs. black-box test, white-box test og gray-box test
- teknikker til at finde frem til testdata og udarbejde testcases

Generelt handler test om at gøre flg.

- **Planlægge** de nødvendige tests for hver fase, iteration (UP) eller andet, planlægge **hvilket system** eller hvilke dele af systemet der skal testes, planlægge **hvad** der skal testes, planlægge af **hvem** der skal testes samt planlægge **hvornår** der skal testes.
- **Design og implementere** test ved at finde frem til passende og dækkende antal testdata og udarbejde testcases på baggrund af disse, dvs. udarbejde de forberedende tekstdokumenter med testcases samt de mere elektronisk kodede testcases. Ofte laver man i dag begge dele, men det kan også være, man kun laver en af delene. Begge dele specificerer, hvad der skal testes, med hvilke input data, der skal testes, og med hvilket output som resultat.
- **Udføre de forskellige test** og systematisk håndtere resultatet af hver test, dvs. rapportere fejl på en systematisk måde, så der kan blive fulgt op på dem af de rette personer, typisk udviklerne.

Man kan mht. **test**, som i mange andre sammenhænge tale om, at der er nogle forskellige niveauer, der opereres på

- Det strategiske niveau (mål og strategi)
- Det taktiske niveau (planlægning)
- Det operationelle niveau (udførelsen)



Før man kan gå i gang med planlægningen, må man altså have bestemt en **strategi** for testen. Strategien handler om at få bestemt, hvad der er de vigtigste systemegenskaber, der skal testes, det kan være egenskaber som korrekthed, robusthed, effektivitet i brug, genbrugbarhed, forståelighed osv. Egenskaber der kan være opstillet nogle mål eller kriterier for jf. afsnit 3.1. Strategien handler også om at få bestemt hvilke principper/politikker, vi tester efter, samt hvilken overordnet fremgangsmåde vi vil følge i testen. Den overordnede fremgangsmåde for en test kaldes også for en testmodel. Testmodellen kan f.eks. være en test-først model, hvor testene udarbejdes før produktet, en mere traditionel test-sidst model, hvor testen udarbejdes efter produktet, en V-model osv. Der skal tages stilling til, hvilke metoder og processer, der passer bedst i forhold til den valgte testmodel og målet med testen. Der skal tages stilling til, hvilke værktøjer og hvilke teknikker der skal anvendes i testen, f.eks. skal testcasene udeluk-

kende dokumenteres i noget test kode (f.eks. JUnit) eller skal de dokumenteres i nogle tekst-dokumenter eller begge dele. Der skal også tages stilling til organiseringen i projektgruppen, skal der evt. laves en egentlig testorganisation eller blot uddelegeres nogle testroller.

5.3 Planlægning af test

Når strategien er bestemt fortsættes på det taktiske niveau, hvor testen skal **planlægges**. Planlægningen går ud på at finde frem til, hvilke aktiviteter der er nødvendige for at gennemføre testen, hvornår aktiviteterne kan gennemføres og hvilke ressourcer der skal til. Hvornår aktiviteterne gennemføres hænger meget på, hvilken overordnet fremgangsmåde også kaldet test-model, der er valgt i strategien. Det skal planlægges hvornår, hvor meget og hvordan der skal testes på de forskellige **testniveauer**. Eksempler på forskellige testniveauer **kan** være niveauerne, der er beskrevet nedenfor

- **Unittest**, test af metoder, klasser (som en helhed dvs. attributter, metoder, tilstande osv.), komponenter osv., før de integreres med anden software
- **Integrationstest**, test af en gruppe af metoder, klasser og komponenter
 - Test af **sammenhænge mellem komponenter og mellem klasser**, planlægningen af denne test kan evt. tage udgangspunkt i en udarbejdet arkitekturmodel, et packagediagram og/eller et klassediagram.
 - Test af **samarbejde mellem komponenter og klasser** for at opfylde en eller anden funktionalitet f.eks. en use case, planlægningen af denne test kan evt. tage udgangspunkt i et for en use case udarbejdet sekvensdiagram
- **Systemtest**, test af om systemet er komplet og korrekt, f.eks. test af at de beskrevne use cases kan gennemføres og virker korrekt.
- **Brugertest**, test af forskellige scenarier af use casene, om de fungerer tilfredsstillende og tilstrækkeligt brugervenligt for brugeren. Niveauet af brugervenlighed kan evt. være beskrevet i nogle brugervenlighedskrav i en kravspecifikation.

Der findes dog også andre testniveauer, f.eks. **Accepttest**, der handler om efter system og brugertest formelt at afgøre om systemet lever op til kravene til systemet og om det dermed er klar til at komme i produktion, **Betatest**, der handler om at afprøve det færdige system i virkelige omgivelser i en lille del af den organisation, der skal modtage systemet, f.eks. at afprøve Skoleadministrationssystemet på en eller to skoler ud af de f.eks. 50 skoler, der skal ende med at have systemet og **Regressionstest**, som handler om efter ændringer at genteste systemet, så man er sikker på, der ikke er kommet følgeføj. Disse tre sidst nævnte testniveauer, vil vi i dette kapitel ikke beskrive yderligere.

Afhængig af den valgte testmodel og hvor man er i testforløbet arbejdes der med de forskellige niveauer i forskellig rækkefølge. Er man i gang med planlægningen og hermed udarbejdelsen af testcasene, sker det ofte først på brugertestniveau og sidst på unittestniveau. Er man derimod i gang med udførelsen af testene sker ofte først på unittestniveau og sidst på brugertestniveau. Vi siger også, at vi tester nedefra. Fordelen ved at teste nedefra er at, hvis der findes fejl på et eller andet niveau, så kan man næsten være sikker på at fejlen skal findes på det niveau, man er på eller over, da niveauerne nedenunder jo er testede.

Input til planlægningen og designet af testen kan altså være mange og komme fra det analyse- og designarbejde, der evt. er dokumenteret i div. beskrivelser og UML diagrammer.

Resultatet af planlægningen af testen kan være en **detaljeret plan** over gennemførslen af en række test **aktiviteter** se figur 5.1. En testaktivitet er en dokumentation af fremgangsmåden i testen og af hvilke testcases, man har valgt skal gennemføres i sammenhæng. Et eksempel på en dokumenteret testaktivitet kan ses i figur 5.2

1. Testplan

Simpel Testplan

Testområde	Test-aktivitet	Testcase	Krav	Planlægges af – inden	Testes af – inden	Fase	Prioritering – 1-3	Status
Brugergrænseflade	TA1	TC-fp01.01	R0.1	FP/JH		1+2	1	
		TC-fp01.02						
	TA2	TC-fp02.01	R0.2	FP/JH		1+2	1	
		TC-fp02.02						
		TC-fp02.03						
	TA3	TC-fp03.01	R0.3	FP/JH		1	1	
Skabeloner Aut. Oprettelse af hold og fag		TC-fp03.02						
		TC-fp03.03						
	TA04	TC-fp04.01	R0.4	FP/JH		1	1	
	TA5	TC5.01	R0.5	HASO		2	1	
		TC5.02						
		TC5.03						
Integration til andre systemer		TC5.04						
	TA6	ingen	R0.6	JEAR		2	3	
	My site	ingen	R0.7	FP/JH		1	3	Irrelevant
		ingen	R0.8	FP/JH		1	1	Venter
	TA9	TC9.01	R0.9	JEAR		1	1	
		TC9.02						
Kalender		TC9.03						
		TC9.04						
			R0.10					Ej leveret
	TA11	TC11.01	R0.11	JEAR		2	1	
		TC11.02						
	TA12	TC-fp.12	R0.12	FP/JH		1	1	
Oprettelse af fora	ingen	TC13.01	R0.13	HASO		1	3	
			R0.14					Ej leveret
	Adgang	TC015.01	R0.15	AMJ		1	2	
		TC015.02						
		TC015.03						

Figur 5.1 eksempel på en testplan med testaktiviteter, testcases, ansvarlige osv.

Testaktivitet	
System: ShareNet	Fase: Brugertest
Navn: Niveaudifferentiering	
Id: TA 2	Krav: R0.2
Omfang/krav: Brugergrænsefladen skal kunne differentieres på flere niveauer, afhængigt af brugerens rolle. Tilgængelig funktionalitet afspejler alene brugerens rolle.	
Fremgangsmåde: Udfør testcase TC-FP 02.01 Udfør testcase TC-FP.02.02 Udfør testcase TC-FP.02.03	
Forgængere:	
Efterfølgere:	
Testcases: TC-FP 02.01 TC-FP.02.02 TC-FP.02.03	
Udfærdiget af: FP	Dato: 13. juni 2006
Status:	
Godkendt af:	

Figur 5.2 eksempel på en testaktivitet

På det operationelle niveau udføres de planlagte tests. Under udførslen samles der op på, hvordan testen er gået. På testcasen påføres det reelle output og evt. status mht. om testen er godkendt eller er fejlet. Evt. laves også en fejlrapport til udviklingsafdelingen. Derudover kan resultatet af testen yderligere dokumenteres i en testlog, der kan give et samlet overblik over afviklede tests, status på testcases, hvilke fejlrapporter testene har givet anledning til osv., se eksempel på testlog i figur 5.3.

[illegible]

Figur 5.3 Eksempel på testlog

Noget meget centralt i selve testplanen er altså testcasene, derfor vil vi i næste afsnit komme nærmere ind på, hvordan vi finder frem til testcasene, hvordan de dokumenteres osv. Der kan læses mere om dokumentation af testplaner, testaktiviteter osv. samt om IEEE standarden herfor i ”Poul Staal Vinjes” bog ”Softwaretest – teknik, struktur, metode”.

5.4 Testcases som centralt element i en test

Testcases designes ud fra de testteknikker, der er defineret i teststrategien. De skal være gentagelige dvs. kunne udføres flere gange, da der ofte vil være brug for gentest eller regressi-onstest. De skal være verificerbare, dvs. det skal være muligt at afgøre, om det er gået godt. De skal være sporbare i forhold til kravene. Der er tre typiske og klassiske testteknikker ”white-box”, ”black-box” og ”gray-box”, som kan få indflydelse på vores design af test cases. De tre teknikker gennemgår vi i det følgende.

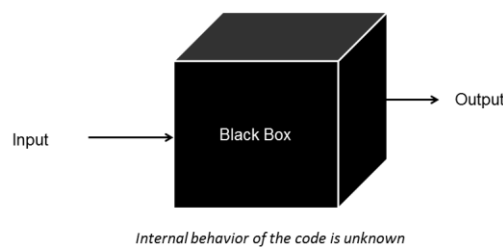
5.4.1 Testteknikker

De nedenfor beskrevne testteknikker ”black-box”, ”white-box” og ”gray-box” har forskelligt udgangspunkt men kan alle bruges mere eller mindre afhængig af situation og behov. De kan mere eller mindre bruges i alle testaktiviteter uanset, hvilket testniveau man befinder sig på.

- Black-box – specifikationsbaseret, dvs. den tager udgangspunkt i, hvad der findes af beskrivelser og modeller af, hvad der skal komme ud af et system, en komponent, en metode, en use case eller andet.
- White-box – strukturbaseret, dvs. den tager udgangspunkt i programmets interne struktur
- Gray-box – specifikationsbaseret test men planlagt af folk, der kender den interne struktur af programmet

5.4.1.1 *Black-box*

Black-box test er som skrevet specifikationsbaseret, dvs. det er kun ud fra specifikationen af funktionaliteten, at testdata, dvs. input og output findes og beregnes. Dvs. man har ingen viden om selve programkoden, man har kun div. modeller og beskrivelser at tage udgangspunkt i.



5.4.1.1.1 *Ækvivalensklasser*

Som udgangspunkt, når vi laver systemer, vil vi jo gerne teste alt i systemet. At teste alt vil dog give et næsten uendeligt antal testcases, som vil kræve rigtig mange ressourcer at lave, vedligeholde og gennemføre. Den mængde ressourcer har man sjældent, hvorfor vi må finde en måde at finde frem til det antal testcases, der lever op til de i strategien opstillede mål for testen. For at finde frem til et passende antal testcases, skal vi først finde frem til den passende og nødvendige mængde testdata. Når vi har fundet den nødvendige mængde testdata, kan vi udarbejde de nødvendige men også tilstrækkelige testcases. I en black-box test, gennemføres flg. skridt i forhold til at finde testdata og udarbejde testcases

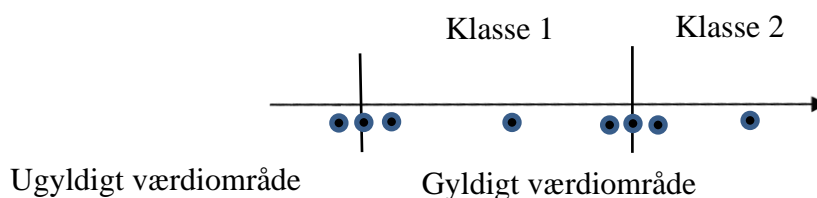
- Identificer det **gyldige** og det **ugyldige værdiområde** for de forskellige input
- Opdel, ved at se på den specificerede funktionalitet, de to værdiområder i **ækvivalensklasser**
- Beskriv **grænseværdierne**
- Udarbejd **testcases**

Grunden til man finder det gyldige og ugyldige værdiområde, er at det kun er på det gyldige område, selve metoden skal testes. Data fra det ugyldige værdiområde skal systemet også

kunne håndtere, men de håndteres som regel et andet sted i systemet, f.eks. i nogle valideringer i grænsefladen eller i nogle valideringsmetoder i et controllerlag lige under grænsefladen.

De to områder opdeles i ækvivalensklasser, dog har vi her mest fokus på det gyldige værdiområde.

At opdele i ækvivalensklasser er en teknik til at finde frem til det nødvendige men også tilstrækkelige antal testcases. Det handler om at få opdelt input i ækvivalensklasser, så der inden for samme ækvivalensklasse foretages samme beregning. Det vil derefter være nok at lave testcases for én værdi inden for hver af ækvivalensklasserne. Ud over at lave testcases for én værdi pr. ækvivalensklasse, bør man have lidt ekstra fokus på grænserne mellem klasserne, dvs. sørge for at have testcases, der tester på grænseværdier. Det skal man for at være sikker på, at beregningen på dem falder til den rigtige side. Når data fra ækvivalensklasserne og grænseværdierne er valgt, kan testcasene med beregnet forventet output laves.



Figur 5.4: Illustration af 2 ækvivalensklasser for de gyldige data og 1 klasse for de ugyldige data. Punkterne illustrerer de testdata, man bør vælge som input til sine testcases. Én input værdi fra hver af de to ækvivalensklasser og derudover nogle grænseværdier.

Nedenfor er de fire trin i black-box testen illustreret gennem et eksempel.

Eksempel: Black-box test af en metode `moveDay(int number)` i en klasse `Date` med en `date` attribut.

Specifikationen af metoden `moveDay(int number)`:

`moveDay` skal i forhold til værdien af dato attributten og et antal dage regne ud, hvad datoen det givne antal dage frem hedder, f.eks. hvis datoen er 14/10-2018 og antallet er 5, så skal algoritmen komme frem til d. 19/10-2018. Algoritmen skal kunne skifte måned og år, hvis antallet af dage, der skal regnes frem giver en ny måned og/eller nyt år. Metoden her antager, at der er 31 dage i alle måneder.



Gyldige og ugyldige værdiområder:

Input til metoden er det antal dage, man ønsker at skrive dagen frem med. De gyldige værdier for antal dage er alle positive hele tal. De ugyldige værdier er de negative tal.

Ækvivalensklasser:

Ækvivalensklasserne i dette eksempel er fundet ved at kigge på specifikationen af metoden og er flg.

1. Dato flyttes antal dage, så ny dato bliver i samme måned
2. Dato flyttes antal dage, så ny dato skifter måned
3. Dato flyttes antal dage, så ny dato skifter både måned og år

Valg af input og testcases:

Nu vælges input i form af et antal dage, så der er en værdi fra hver af de tre mængder samt værdier som repræsenterer grænseværdierne.

Hvilken dato der bruges som den dato vi regner frem fra er ligegyldig, så der vælges bare en. Den form for data, som er nødvendig for testen, men som ikke er bestemt af ækvivalensklasserne kalder vi for basisdata.

I dette eksempel vælges datoen 14/10-2018 som basisdato. Testcasene, lavet på baggrund af de valgte input, dokumenteres i en testtabel. Et eksempel på sådan en testtabel ses nedenfor.

Testcase	Input	Expected output	Actually output	State
TC1	number = 5	19/10-2018	19/10-2018	ok
TC2	number = 16	30/10-2018	30/10-2018	ok
TC3 - grænse-værdi	number = 17	31/10-2018	0/11-2018	error
TC4	number = 18	1/11-2018	1/11-2018	ok
TC5	number = 78	30/12-2018	30/12-2018	ok
TC6 - grænse-værdi	number = 79	31/12-2018	0/1-2019	error
TC7	number = 80	1/1-2019	1/1-2019	ok
TC8 - ugyldig	number = -1	Fejlmeddelelse		

Bemærk at der er valgt meget konkrete testdata. Det er vigtigt for, at det senere er muligt at genskabe problemet ved en evt. gentest.

Eksempel på JUnit kode, der tester TC1:

```
private Dato dato;

@Before
public void setUp() throws Exception {
    dato = new Dato(2018, 10, 14);
}

@Test
public void testMoveDay() {
    // TC1
    dato.moveDay(5);
    assertEquals(19, dato.getDay());
    assertEquals("19/10-2018", dato.toString());
}
```

Efter at have kørt testen, så dokumenteres det reelle output i skemaet og status registreres. Som det ses af tabellen her, er jeg kommet til at lave en meget typisk kodefejl. Det er netop pga., at det er en typisk fejl, at det er vigtigt at teste på grænseværdierne. Testen på grænseværdierne her giver et forkert resultat i forhold til det forventede, hvorfor der må være en fejl i koden. Se på figur 5.5 og prøv at finde fejlen☺

Dette var et eksempel på, hvordan testcases til en black-box test på struktureret vis udarbejdes og dokumenteres.

I tilfælde af mange forskellige input og forskellige ækvivalensklasser for hver af disse, kan det, hvis alle kombinationer af input skal testes, blive til rigtig mange forskellige testcases. Det er ikke altid muligt at teste så mange tilfælde, så derfor bruges nogle gange teknikken ortogonale arrays til at begrænse antallet af test cases. Teknikken ortogonale arrays handler om at lave testcases så det sikres at en værdi fra en ækvivalensklasse for **et** input parres med en værdi fra en anden ækvivalensklasse for et **andet** input mindst én gang. Det der ligger i det er

altså, at har man testet nogle input i sammenhæng én gang så er det nok til at vide, at det virker. Alle mulige kombinationer af værdier behøver ikke, medmindre man vil op på en 100% dækning for alle typer test, at testes. Der kan læses mere om ortogonale arrays på flg. link https://en.wikipedia.org/wiki/Orthogonal_array_testing , eller i Louise Tamres: Introducing Software Testing kap. 6.4.1⁵.

I næste afsnit vil vi bruge samme eksempel til at vise, hvordan testcases til en white-box test på struktureret vis kan udarbejdes.

5.4.1.2 *White-box*

White-box testen afledes systematisk fra den interne struktur i programmet, dvs. strukturen af statements og logiske kontrolstrukturer (if, for, while osv.) i programmet. Da strukturen i et program kan være kompleks, og hvis alle mulige veje rundt i programmet skal testes, kan det give anledning til et hav af testcases. Derfor bør man inden, man går i gang med at udlede testcases forholde sig til et såkaldt dækningskriterie, som fortæller noget om, hvor ambitiøs man er med testen i forhold til at komme hele vejen rundt. Dækningskriteriet fortæller altså, hvornår udarbejdelsen af testcases kan stoppe.

Det gælder altså for strukturbaserede teknikker:

- Oplysninger om, hvordan softwaren er opbygget, bruges til at udlede testcases, f.eks. kode og designmodeller
- Hvor godt vi ønsker at dække softwaren er udtrykt i et dækningskriterie og kan måles for eksisterende testcases

I denne note kommer vi med nogle bud på dækningskriterier, som kan ligge til grund for valget af og testcases

- **Statementdækning** – hvis 100% statementdækning, så designes testcases, så alle statements gennemløbes mindst én gang.
- **Forgreningsdækning** – hvis 100% forgreningsdækning, så designes testcases, så alle if/else-, switch/case-, for- og while-statements identificeres og rammes.
- **Stidækning** - hvis 100% stidækning, så designes testcases, så alle mulige stier rundt i programmet identificeres og gennemløbes mindst én gang.

⁵ Louise Tamres: Introducing Software Testing, Pearson Education, ISBN 0-201-71974 6, kap. 6.4.1⁵

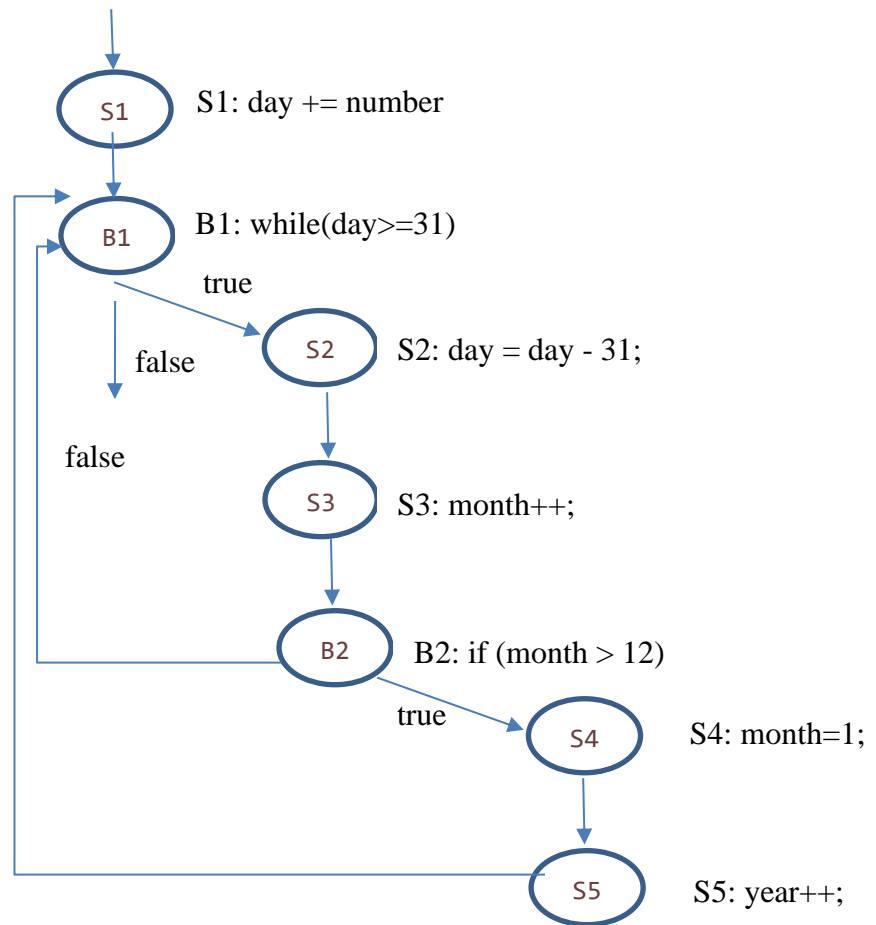
- **Betingelsesdækning** - hvis 100% betingelsesdækning, så designes testcases, så alle sammensatte betingelser afprøves, så alle mulige kombinationer af true/false er dækket ind.

I forhold til alle dækningskriterierne skal der, inden man går i gang med at lave testcases, tages en beslutning om dækningsprocenten. Procenten vælges ved at se på, hvad der giver mening i forhold til de opstillede kvalitetsmål, og/eller hvad der er ”råd” til i forhold til et konkret projekt. Det kan blive meget omfattende, dyrt og måske endda umuligt, hvis alt skal testes til 100%.

I forhold til udledning af testcasene til en white-box test er et meget relevant værktøj det, vi kalder rutediagrammer. Et eksempel på et rutediagram for kodeeksemplet i figur 5.5, kan ses i figur 5.6.

```
public void moveDay(int number) {  
    day += number;  
    while (day >= 31) {  
        day = day - 31;  
        month++;  
        if (month > 12) {  
            month = 1;  
            year++;  
        }  
    }  
}
```

Figur 5.5 Koden for metoden moveDay, som vi nedenfor har lavet et rutediagram for



Figur 5.6 Rutediagram for metoden moveDay.

Rutediagrammet bruges til at udlede testcases, og i tabellen figur 5.7 kan man se, hvad testcasesne skal ramme for at opnå de forskellige dækningskriterier.

Statementdækning	S1, S2, S3, S4, S5
Forgreningsdækning	S1, S2, S3, S4, S5
Stidækning	S1 S1, S2, S3 S1, S2, S3, S4, S5
Betingelsesdækning	!B1 B1 && B2 B1 && !B2

Figur 5.7 Eksempel på testcases udledt af rutediagrammet

Eksemplet her giver ikke så stort et rutediagram. Der er ikke så mange forskellige stier, så måske kunne testcasene udledes uden. Der bliver imidlertid ofte kodet store komplekse algoritmer, hvor et rutediagram kan være en stor hjælp til at finde testcasene.

Som hjælp til at vide, hvor meget af ens kode, man har fået dækket af testcases, kan anvendes såkaldte "code coverage" værktøjer. En del udviklingsværktøjer stiller disse værktøjer til rådighed. Værktøjerne kan med farver anskueliggøre, hvor meget af ens kode, der er dækket af testcases. I Eclipse er værktøjet i de seneste versioner blevet integreret. Nedenfor kan ses hvordan code coverage er brugt til test af metoden `moveDay()`. Den grønne farve viser det, der er dækket af en testcase, mens det røde ikke er.

```
public void moveDay(int number)
{
    day += number;
    while (day >= 31) {
        day = day - 31;
        month++;
        if (month > 12) {
            month = 1;
            year++;
        }
    }
}
```

5.4.1.3 Gray box

Gray-box test er en blanding af Black-box og white-box. Det skal forstås på den måde, at det faktisk er en black-box test, hvor testcases er udledt af folk, der kender den interne struktur i programmet. Det at strukturen af programmet er kendt, vil altid på en eller anden måde komme til at påvirke valget af testcases, selvom målet med testcasene er en black-box test. Altså en black-box test, der pga. kendskab til koden er blandet lidt hvidt i 😊

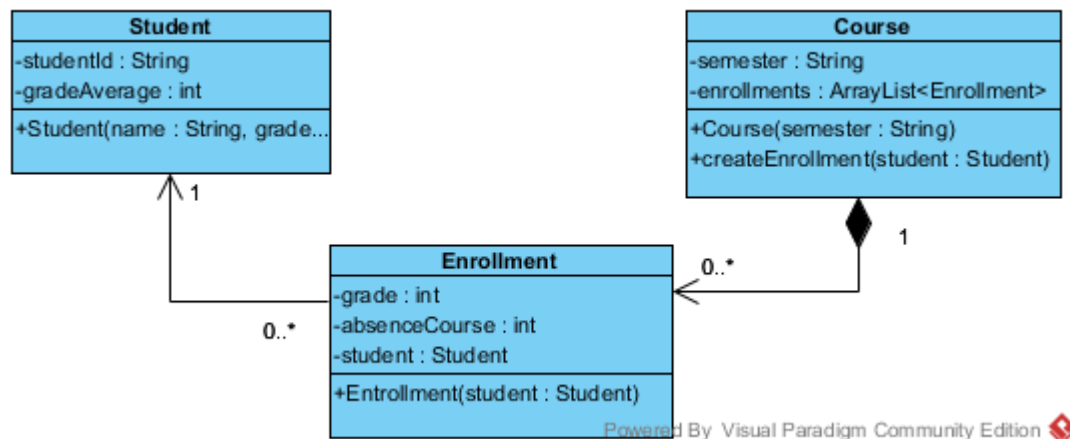
5.5 Testniveauer

5.5.1 Test på unittest niveau

Eksemplet med metoden `moveDay()`, der er brugt ovenfor til beskrivelse af de forskellige testteknikker, er et typisk eksempel på en test på unittest niveau. Vi kan dog også på dette niveau teste hele klasser dvs. deres attributter, metoder og tilstande som en helhed. Hele komponenter kan også her testes som en helhed, før de integreres med andre klasser eller komponenter.

5.5.2 Test på integrationstest niveau

Her testes typisk sammenhænge på forskellige niveauer, det kan være sammenhænge til eksterne systemer, sammenhænge mellem større komponenter, sammenhænge og samarbejde mellem klasser osv. Til test af sammenhænge mellem systemer tages udgangspunkt i beskrivelserne af grænsefladen mellem systemerne. Til test af sammenhænge mellem komponenter kan der evt. tages udgangspunkt i en arkitekturmodel eller packagediagram. Til test af sammenhænge og samarbejde mellem klasser kan der evt. tages udgangspunkt i et klassediagram se figur 5.8 og/eller sekvensdiagram. Et eksempel på en testcase, der tester om de rigtige sammenhænge mellem objekter skabes i forbindelse med at et objekt skabes, er vist nedenfor i figur 5.9



Figur 5.8 Designmodel af Student, Course og Enrollment.

Eksempel: Test af metoden `createEnrollment(Student student)`

Basisdata:

```
Course course = new Course("E15");
```

Testcase	Input	Expected output	Actually output	State
TC1	Student "Hans Hansen"	There has been made a new Enrollment object that has a relation to Student "Hans Hansen". The enrollments of the object of Course contains the new Enrollment object		
Osv.				

Figur 5.9: Tabel med testcase, som beskriver hvilke sammenhænge, der skal testes i forbindelse med at metoden createEnrollment() kaldes.

Eksempel på JUnit kode, der tester TC1:

```

private Course course;

@Before
public void setUp() throws Exception {
    course = new Course("E15");
}

@Test
public void testCreateEnrollment() {
    // TC1
    Student student = new Student("Hans Hansen", "100", 9.2);

    Enrollment enrollment = course.createEnrollment(student);

    assertNotNull(enrollment);
    assertTrue(course.getEnrollments().contains(enrollment));
    assertEquals(student, enrollment.getStudent());
}

```

5.5.3 Test på systemtestniveau

Når vi skal afgøre om systemet er komplet og korrekt dvs. lever op til både de funktionelle og ikke funktionelle krav, skal der testes på et højere niveau end på klasse og metode niveau. For at afgøre det, skal der testes på det vi kalder systemtestniveau. Når vi nu i denne note er i en Unified Process verden, kan vi med fordel bruge use cases som udgangspunkt for vores test af funktionaliteten. En use case kan betragtes som en metode på et højere niveau, dvs. der kan specificeres et pre og et post for use casen og der er en sekvens af interaktioner i use casen svarende til sekvensen af statements i metoden. Use casens sekvens er interaktioner er beskrevet i en use case beskrivelse, og på baggrund af denne kan vi udlede de nødvendige testcases. Vi vil prøve at eksemplificere udledningen af testcases ud fra et eksempel på en use case. Eksemplet kan ses i figur 5.10 nedenfor. Vi vil samtidig vise, hvordan man dokumenterer testcases til test af en use case.

Use Case Name:	UC4: Enroll course	
Trigger Event:	Student wish to take a course	
Brief Description:	When the student choose to take a new course of a specific education, he or she register the enrollment to the course in the system	
Actors:	Student or Department	
Related use cases:	UC3: Register student, UC6: Check student state	
Stakeholders:	Student, School	
Precondition:	Course is registered	
Postcondition:	Student is enrolled course	
Flow of events:	<p>Actor</p> <ol style="list-style-type: none"> 1. The actor enters name or cpr number of the student 2. If the system did not find the student in the system go to 3. The actor asks for possible educations 4. The actor chooses the wanted education and the correct semester 5. The actor chooses a course and confirms the choice 	<p>System</p> <ol style="list-style-type: none"> 1.1 The system displays the information about the student 3.1 The system displays available educations 4.1 The system displays the courses of the chosen education and semester 5.1 The system checks the student state <<include>> UC6: Check student state and creates an enrollment of the student to the course
Exceptional Flows:	<ol style="list-style-type: none"> 4. Condition: No course is suited for the student <ol style="list-style-type: none"> a. The actor interrupts the use case 5.1. Condition: The student state is not valid <ol style="list-style-type: none"> a. The actor is forced to interrupt the use case 	

Figur 5.10: Use casen her giver altså 4 testcases, som tester hvert sit scenarie. De 4 testcases kan ses i figur 5.11 nedenfor.

Testcase ID	What do we test?	Initial system state	Input	Expected output	State
TC1	Normal flow without Register Student	Enroll Course screen	<p>Enter name "Hans Hansen"</p> <p>Ask for possible educations</p> <p>Choose the education "Computer Science"</p> <p>Choose the course "SU1"</p>	<p>Displays information about the student</p> <p>Displays the possible educations of this semester</p> <p>Displays the courses of "Computer Science" for this semester</p> <p>Confirmation of the enrollment</p>	
TC2	Normal flow with Register Student	Enroll Course screen	<p>Enter name "Hans Hansen"</p> <p>Go to "Register Student"</p> <p>Ask for possible educations</p> <p>Choose the education "Computer Science"</p> <p>Choose the course "SU1"</p>	<p>Tells that the student is not found</p> <p>Displays the possible educations of this semester</p> <p>Displays the courses of "Computer Science" for this semester</p> <p>Confirmation of the enrollment</p>	
TC3	Exceptional Flow event 4.	Enroll Course screen	Enter name "Hans Hansen"	Displays information about the student	

			Ask for possible educations	Displays the possible educations og this semester	
			Choose the education "Computer Science"	Displays the courses af "Computer Science" for this semester	
			Interrupts		
TC4	Exceptional Flow event 5.1.	Enroll Course screen	Enter name "Hans Hansen"	Displays information about the student	
			Ask for possible educations	Displays the possible educations og this semester	
			Choose the education "Computer Science"	Displays the courses af "Computer Science" for this semester	
			Choose the course "SU1"	Tell that the student state is not valid	
			Interrupts		

Figur 5.11 Testcases til test af use case beskrevet i figure 5.10

Andre ting, der også kan testes og dermed skal laves testcases til under systemtesten er sådan noget som robusthed, performance, vedligeholdbarhed osv. Ting der i kravspecifikationen kan være stillet en række ikke funktionelle krav op for. Test af robusthed kan handle om at teste om systemet reagerer fornuftigt på hårdhændet, ufornuftig eller uventet brug af systemet, hvilket kan være at taste noget helt forkert i felterne, komme til at trykke på flere ting på én gang eller for hurtigt efter hinanden osv. Som minimum bør systemet ikke gå ned i de situ-

ationer. Test af performance kan f.eks. handle om at teste sådan noget som svartider, er svartiderne på f.eks. store datamængder rimelige, eller kommer man til at vente for længe og blive utålmodig.

5.5.4 Brugertest

Som tidligere skrevet er brugertesten en test af, om systemet udover at være korrekt også er egnet til brug. Det handler meget om at få testet eventuelt opstillede ikke funktionelle krav i forhold til tilfredshed, brugervenlighed, forståelighed osv. De ikke funktionelle krav er som regel opstillet og beskrevet i kravspecifikationen.

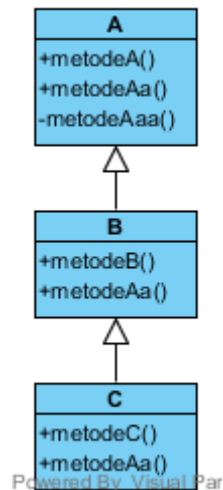
5.5.5 Opsummering mht. testniveauer

Tabellen nedenfor opsummerer de ovenfor beskrevne testniveauer

Testniveau	Specifikation der tages udgangspunkt i	Hvad testes	Teknikker der kan anvendes	Hvem tester
Unit test	F.eks. designsekvensdiagrammer, metodebeskrivelser, aktuel kode osv.	Små enheder som metoder og klasser	White-box, gray-box, black-box	Program-møren
Integrationstest	F.eks. Arkitekturdiagrammer, Klassediagrammer osv.	Sammenhænge mellem klasser og komponenter	White-box, gray-box, black-box	Program-møren
Systemtest	F.eks. funktionelle krav og use cases	Om produktet lever op til det beskrevne mht. funktionalitet, f.eks. det der er beskrevet i de funktionelle krav og use cases	Black-box	Intern tester (ej programmør) eller en bruger
Brugertest	Ikke funktionelle krav, funktionelle krav og use cases	Om produktet som helhed lever op til beskrevet funktionalitet med beskrevet kvalitet	Black-box, spørge-skema	Bruger
Accepttest	Højest prioriterede krav, accept kriterie specificeret af kunden	Om systemet kan siges at leve op til de højest prioriterede krav, accept kriterie	Ikke rigtig test, mere en beslutning	Kunde, rekvirent
Betatest	Ad hoc arbejdsopgaver i organisationen	Om systemet fungerer i virkelige omgivelser til løsning af virkelige opgaver	Black-box	Bruger
Regressionstest	Ændrede design beskrivelser, diagrammer osv.	Hvad som helst af det ovenfor beskrevne	Black-box, white-box, gray-box	Enhver af dem ovenfor beskrevet

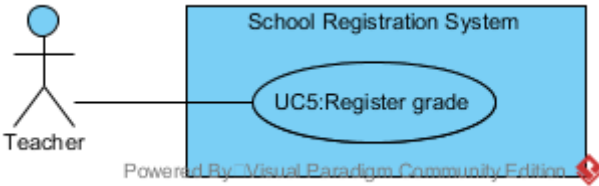
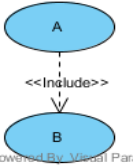
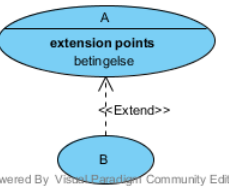
5.6 Test og OO

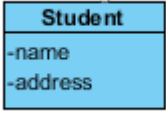
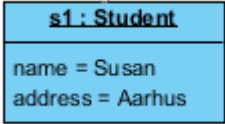
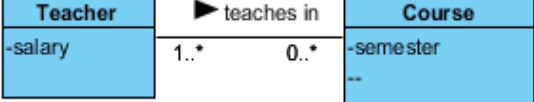
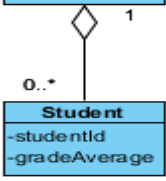
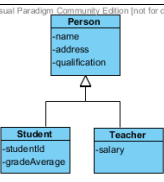
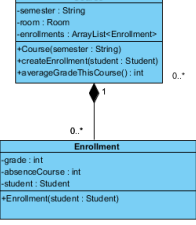
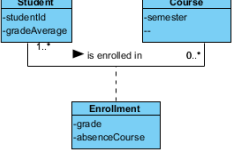
Når vi laver objekt orienteret systemudvikling, er der nogle specielle ting, vi bør teste. Vi bør f.eks. ved brug af specialisering teste alle nedarvede metoder og alle redefinerede/overskrevne metoder, ved at kalde dem med objekter af subklasserne.

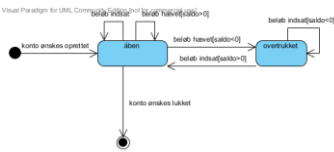
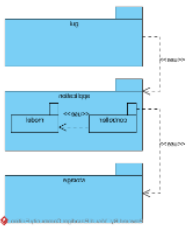


Det betyder altså i forhold til modellen ovenfor, at de metoder, der skal testes på et objekt af A, er metoderne metodeA(), metodeAa() og den private metode metodeAaa(). De metoder, der skal testes på et objekt af B, er metodeA() nedarvet fra A, metodeAa() redefineret i B og metodeB(). De metoder, der skal testes på et objekt af C, er metodeA() nedarvet fra A, metodeAa() redefineret i C, metodeB() nedarvet fra B og metodeC() fra C.

6 Begrebsliste

Begreb	Beskrivelse
UML	Unified Modelling Language, International standard diagrammeringsteknik
Use case	Tidsmæssig afgrænset sekvens af interaktioner med systemet, der giver et resultat af værdi for aktøren
Aktør	Rolle omfattende brugere eller andre systemer, der interagerer direkte med systemet
Use case diagram	 <p>Diagram over systemets use cases og deres relationer til aktøren</p>
Include	 <p>Opfattes sådan, at use case B er en del af use case A og udføres hver gang (eller næsten hver gang)</p>
Extend	 <p>Udvider en use case, dvs. beskriver en speciel interaktion, der kun udføres en gang imellem, nemlig når en bestemt betingelse er opfyldt.</p>
Use case beskrivelse	<p>Beskrivelse af interaktionen mellem aktør og system i en use case. Der kan laves en kort, mellemlang og fuld beskrivelse.</p> <p>Eksempel mellemlang beskrivelse:</p> <p>UC3: Register student</p> <p>Main flow:</p> <ol style="list-style-type: none"> 1. The actor enters the cpr number for the student 2. The system verifies that the student is not already registered 3. The actor enters other personal data about the student (name, address, phone number etc.) 4. The actor chooses the education that the student applies for 5. The actor enters exam data 6. The system verifies that the exam data fulfills the requirements for the chosen education 7. The system accepts the student
Krav	<p>Funktionelle krav er udtryk for det, man skal kunne gøre med systemet</p> <p>De ikke-funktionelle krav er kvalitetsegenskaber ved systemet</p> <p>Krav dokumenteres efter flg. skabelon:</p> <p style="text-align: center;">Nr – beskrivelse – prioritet</p>
Krydstabel	Laves til kvalitetsjæk af, om de funktionelle krav bliver realiseret gennem de valgte use cases.

Fænomen	Substans (volume og sted i tid og rum), målelige egenskaber, transformationer
Klasse	<p>Visual Paradigm for UML</p>  <pre> classDiagram class Student { -name -address } </pre> <p>En klasse er en model af et begreb fra den virkelige verden. En beskrivelse af en mængde af ens objekter dvs. med samme struktur, adfærd og attributter</p>
Objekt	<p>Visual Paradigm for UML Community Edition (not for commercial use)</p>  <pre> classDiagram class s1_Student { name = Susan address = Aarhus } </pre> <p>Et objekt er en model af et konkret fænomen fra den virkelige verden. Helhed med identitet, tilstand og adfærd. Et objekt er en instans af en klasse.</p>
Statisk sammenhæng	Struktur mellem klasser som beskrivelse, ikke dynamisk sammenhæng mellem objekter.
Dynamisk sammenhæng	En dynamisk sammenhæng, er en sammenhæng mellem objekter, der kan komme og gå i systemet
Associering	<p>Visual Paradigm for UML Community Edition (not for commercial use)</p>  <pre> classDiagram class Teacher { -salary } class Course { -semester } Teacher "1..*" -- "0..*" Course : teaches in </pre> <p>Associeringen er en dynamisk sammenhæng mellem ligestillede objekter.</p>
Aggregering	<p>Visual Paradigm for UML</p>  <pre> classDiagram class Class { -name } class Student { -studentId -gradeAverage } Class o-- "0..*" Student </pre> <p>Aggregeringen anvendes når et overordnet objekt (helheden/foreningen) består af et antal mere underordnede objekter (delene/medlemmet).</p>
Multiplicitet	<p>Powered By</p> <p>Eksempel på multiplicitet nul til mange. Kan også kaldes mangfoldighed eller kardinalitet.</p>
Generalisering/specialisering	<p>Visual Paradigm for UML Community Edition (not for commercial use)</p>  <pre> classDiagram class Person { -name -address -qualification } class Student { -studentId -gradeAverage } class Teacher { -salary } Person < -- Student Person < -- Teacher </pre> <p>Der modelleres med den statiske sammenhæng specialisering/generalisering, når der i virkeligheden findes fænomener, der har nogle egenskaber til fælles men også har nogle egenskaber, der er specielle for det enkelte fænomen.</p>
Composition	<p>Visual Paradigm for UML Community Edition (not for commercial use)</p>  <pre> classDiagram class Course { -semester : String -room : Room -enrollments : ArrayList<Enrollment> +Course(semester : String) +createEnrollment(student : Student) +averageGradeThisCourse() : int } class Enrollment { -grade : int -absenceCourse : int -student : Student +Enrollment(student : Student) } Course "1" *-- "0..*" Enrollment </pre> <p>En komposition bruges når virkeligheden er sådan, at kun helheds/forenings objektet skal kende til/referere til sine del/medlems objekter i hele deres livsforløb. Helheden skal stå for at oprette og nedlægge objekterne.</p>
Associeringsklasse	<p>Visual Paradigm for UML Community Edition (not for commercial use)</p>  <pre> classDiagram class Student { -studentId -gradeAverage } class Course { -semester } class Enrollment { -grade -absenceCourse } Student "1..*" -- "0..*" Course : is enrolled in Enrollment -- Student Enrollment -- Course </pre> <p>Der modelleres med en associeringsklasse, hvis der på en mange til mange associering er brug for at registrere noget information på selve sammenhængen mellem de to klasser.</p>
Analysemønster	Def. mønster: Et mønster er en generel løsning til et generelt defineret problem

	<p>Analysemønstre er mønstre der kan tages i betragtning, når vi laver analyseklassediagrammer. Der findes Rolle-mønsteret, Genstand-beskrivelses mønsteret, Relations-mønsteret, Hierarki-mønsteret osv.</p> <p>Fordelen ved brug af mønstre er, at det er gennemafprøvede løsninger sprunget ud af praksis.</p>
Klassediagram	Fundne klasser forbundet med strukturerne associering, aggregering osv.
Retning	Valg af retninger på associeringer, aggregeringer og kompositioner tages på baggrund af, om de enkelte metoder til udførelse af deres ansvar har brug for direkte adgang til relaterede objekter. Enkeltrettet tegnes med en pil i den ene ende, dobbeltrettet tegnes uden pile.
Linkattribut	Den attribut der realiserer en valgt retning.
Adfærdsmønster	 <p>Hvordan hændelserne påvirker objektet og hvilke eventuelle tilstande, det ender i, beskrives i et adfærdsmønster.</p>
Tilstandsdiagram	Den UML diagrammeringsteknik der bruges til at tegne et adfærdsmønster.
Designkriterie	Et designkriterie er et kvalitetsmål, der fremhæver et bestemt aspekt ved et design. F.eks. kriteriet "fleksibelt", krav for omkostningen ved at ændre i det kørende system.
Arkitektur	 <p>En arkitektur, dvs. systemets store dele, vælges på baggrund af prioriteringen af kriterier, hvilket system osv.</p> <p>I en lagdelt arkitektur, f.eks. denne 3-lags model, har vi følgende regel: <i>En klasse i et lag kan IKKE bruge noget fra lag OVEN-OVER dets eget lag</i></p>
Kobling	Kobling er et udtryk for, hvor tæt to klasser, komponenter eller delsystemer hænger sammen, kobling forekommer f.eks. når en klasse, metode, komponent eller delsystem refererer til en anden klasse, komponent eller delsystem. Målet er lav kobling , kan dog ikke undgå noget kobling, hvis der samtidig skal være samhørighed.
Samhørighed	Samhørighed er et udtryk for, hvor godt hver enkelt klasse, komponent eller delsystem hænger sammen. Målet er høj samhørighed .
Applikationslag	Dette lag indeholder pakkerne model og controller.
Controllerpakken	Controllerpakken indeholder en eller flere controllerklasser, der typisk har metoder, der ikke naturligt kan placeres på en klasse i model pakken, fordi metodens logik eksempelvis involverer flere klasser.
Modelpakken	Modelpakken indeholder model klasser (også kaldet forretningsklasser/domæneklasser) som Education, Subject etc. Associeringer eller andre strukturer forbinder klasserne.
Storagelag	Dette lag indeholder klasser, der opbevarer objekter af modelklasser, så man senere kan finde frem til og referere til disse objekter. Klasserne indeholder metoder til at hente, tilføje og fjerne objekter.
Sekvensdiagram	Et <i>sekvensdiagram</i> viser sekvensen af interaktionen mellem aktører og/eller objekter, i realiseringen af en use case.
Systemsekvensdiagram	Systemsekvensdiagrammerne bruges også som use case beskrivelserne til at modellere interaktionen mellem systemet og aktøren. Deres formål er at

	have specielt fokus på at vise, hvilke beskeder, der er sendt til systemet, hvad og hvornår der er input og output til og fra systemet.
Designsekvensdiagram	I designsekvensdiagrammerne modelleres samarbejde og interaktion mellem objekter, der realiserer en use case
Systemudviklingsmodel	Overordnet beskrivelse af en fremgangsmåde i udviklingen af systemer. Ofte blot en illustration/tegning.
Systemudviklingsmetode	Detaljeret beskrevet fremgangsmåde i udviklingen af systemer, dvs. en beskrivelse af hvilke aktiviteter, teknikker, produkter osv.
Unified Process	Navnet på en iterativ use case drevet systemudviklingsmetode, der benytter sig meget af UML som diagrammeringsværktøj.
Fase	Henviser til faserne i Unified Process Inception, Elaboration, Construction og Transition, som er de faser, man gennemløber i metoden. Der snakkes også om faser i andre systemudviklingsmodeller f.eks. vandfaldsmodellen.
Disciplin	I en iteration i en fase gennemløbes en række discipliner f.eks. Business Modelling, Requirement, Analysis & Design, Implementation, Test, Deployment osv.
Iteration	En gennemførelse af et udviklingsforløb, der udvikler en del af systemet.
Iterativ	Det at arbejde iterativt går på processen, nemlig at arbejde på en del af systemet ad gangen.
Inkrementel	Inkrementel betyder at produktet gradvist vokser. Det vokser med den del, der er arbejdet på i iterationen.
Projektplan	Tidsplan over projektet dvs. en start og slut. Derudover nogle faser og fase-linjer undervejs. Faserne indeholder aktiviteter med tid og ansvarlig. Fase-linjerne indeholder produkter, vurderingskriterier og vurderingsprocedurer for vurdering af kvaliteten af produkterne.
Teststrategi	Mål og strategi for testen f.eks. test først, test sidst, nul-fejl osv.
Testplan	Plan over gennemførelsen af en række test aktiviteter, ansvarlige osv.
Testaktivitet	En testaktivitet er en dokumenteret fremgangsmåde i testen, dvs. hvilke testcases, man har valgt, skal gennemføres i sammenhæng
Testdækning	Testdækning er et udtryk for nogle dækningskriterier, der udtrykker, hvor godt vi ønsker at dække softwaren med testcases. F.eks. statementdækning – hvis 100% statementdækning, så designes testcases, så alle statements gennemløbes mindst én gang. Derudover er der noget der hedder forgreningsdækning og stidækning.
Testcase	Testcases designes ud fra de testteknikker, der er defineret i teststrategien. De dokumenteres som regel i en tabel med et testcaseid, et input, et forventet output, et reelt output og en status.
Testniveau	Det skal planlægges hvornår, hvor meget og hvordan der skal testes på de forskellige testniveauer f.eks. unittest, integrationstest, systemtest, brugertest osv.
Unittest	Test af metoder, klasser (som en helhed dvs. attributter, metoder, tilstande osv.), komponenter osv.
Integrationstest	Test af sammenhænge mellem komponenter og mellem klasser
Systemtest	Test af om systemet er komplet og korrekt, dvs. at de beskrevne f.eks. use cases kan gennemføres og virker korrekt.

Brugertest	Test af forskellige scenarier af use casene. Test af om de fungerer tilfredsstillende i forhold til evt. opstillede ikke funktionelle krav.
Accepttest	Handler om efter system og brugertest formelt at afgøre om systemet lever op til kravene til systemet, og om det dermed er klar til at komme i produktion
Betatest	Handler om at afprøve det færdige system i virkelige omgivelser i en lille del af den organisation, der skal modtage systemet
Regressionstest	Handler om efter ændringer at genteste systemet, så man er sikker på, der ikke er kommet følgefejl.
Testteknikker	På de forskellige testniveauer anvendes en række teknikker, f.eks. black-box test, white-box test, gray-box-test.
Blackbox test	Specifikationsbaseret, dvs. den tager udgangspunkt i, hvad der findes af beskrivelser og modeller af, hvad der skal komme ud af et system, en komponent, en metode, en use case eller andet.
Whitebox test	Strukturbaseret, dvs. den tager udgangspunkt i programmets interne struktur. Meget fokus på at teste forskellige veje rundt i koden jf. if, while, case osv.
Graybox test	Specifikationsbaseret test men planlagt af folk, der kender den interne struktur af programmet, hvilket ikke kan undgå at komme til at påvirke valget af testcases.
Ækvivalensklasse/ ækvivalensmængde	Det handler om at få opdelt input i ækvivalensklasser, så der inden for samme ækvivalensklasse foretages samme beregning. Derefter laves testcases for én værdi inden for hver af ækvivalensklasserne samt testcases, der tester på grænseværdierne mellem klasserne.
Rutediagram	Rutediagrammet illustrerer vejene igennem et stykke kode og bruges til at udlede testcases i en white-box test.
Orthogonale arrays	Teknikken ortogonale arrays handler om at lave testcases, så det sikres at en værdi fra en ækvivalensklasse for et input parres med en værdi fra en anden ækvivalensklasse for andet input mindst én gang. En teknik der kan bruges til at begrænse antallet af test cases.

7 Referencer og supplerende litteratur

[Mat] Lars Mathiassen: *Objekt orienteret analyse og design*, ISBN 87-7751-129-8

[Satz] Satzinger, Jackson, Burd: *Object-Oriented Analysis & Design with*. Thomson, 2005,
ISBN 0-619-21643-3