Ingeniørhøjskolen i Århus

iha.dk

SPU – UML

note

Systematisk
Program-
Udvikling
med **UML**

Finn Overgaard Hansen

Ingeniørhøjskolen i Århus

Versionshistorie

Versionsnr.	Dato	Initialer	Versionen omfatter
0.9	18.08.2003	FOH	Første version – kapitel 5 er pt. foreløbigt, idet der her planlægges med mere info.
0.901	10.08.2004	FOH	Opdatering af nedenstående link
0.902	11.08.2005	FOH	Opdatering af referenceliste

Forord

Denne opdatering af SPU håndbogen er alene udarbejdet af undertegnede medforfatter til håndbogen og er således min egen udgave og fortolkning af en opdatering af SPU håndbogen. Denne opdatering er udført med speciel fokus på relationen mellem SPU og UML.

I forfatterkredsen har vi gennem tiden talt om en evt. opdatering, men da dette er et kæmpearbejde og da de øvrige medforfattere til SPU håndbogen i dag alle arbejder i forskellige firmaer og organisationer, har dette ikke været praktisk gennemførligt.

Da SPU håndbogen anvendes i forbindelse med semesterprojekter på Ingeniørhøjskolen i Århus og man her anvender UML i undervisningen fra første semester er der her et behov for en sådan opdateringsnote.

Denne note vil blive publiceret via min hjemmeside på Ingeniørhøjskolen i Århus til brug for studerende og lærere samt andre, der måtte have interesse i SPU og i denne UML relaterede opdatering (http://staff.iha.dk/foh/). I august 2005 vil denne note også blive tilgængelig via en opdateringsside til SPU håndbogen på Nyt Teknisk Forlag

Århus, august 2005.

Finn Overgaard Hansen,
Ingeniørhøjskolen i Århus

Indholdsfortegnelse

1 Introduktion

1.1 Formål

Formålet med denne note er at opdatere ”Håndbog i Struktureret ProgramUdvikling (SPU)” [SPU88] specielt i relation til anvendelsen af UML (Unified Modeling Language) [UML2003] i forbindelse med objektorienteret softwareudvikling.

1.2 Baggrund for SPU

SPU konceptet blev udviklet i 1985/86 i forbindelse med gennemførelse af et teknologirådsstøttet projekt. Formålet med projektet var at forbedre softwareudviklingen i danske virksomheder specielt indenfor teknisk programudvikling, der typisk omhandlede udvikling af software til apparater. Projektet bestod i at udvikle konceptet samt udvikling af et tilhørende kursusforløb, der på fem dage simulerede et udviklingsforløb, hvor der blev udviklet dele af et patientovervågningssystem. Efter afholdelse af utallige firmakurser og plankurser dels på Teknologisk Institut og dels på Delta (det daværende Elektronikcentralen) blev der i 1987/88 gennemført endnu et teknologirådsstøttet projekt, hvis resultat medførte en bearbejdning af konceptet og kursusmaterialet til i SPU håndbogen, der blev udgivet på Teknisk Forlag i 1988.

Rationalet for udvikling af SPU konceptet var følgende:
- det skulle være operationelt og dermed let at anvende
- det skulle kunne anvendes på såvel mindre som større projekter
- det var målrettet mod teknisk programudvikling
- det omhandlede udelukkende softwareudviklingen
- det skulle være metodeuafhængigt

Eksemplerne fra den gennemgående case på det tilhørende kursus blev valgt til eksemplerne i håndbogen. Nogle af disse eksempler blev baseret på metoden ”struktureret analyse og design af realtidssystemer (Ward&Mellor)”, der var den mest udbredte metode til denne type systemer på det tidspunkt SPU håndbogen blev udarbejdet. Disse eksempler ville i dag typisk blive udarbejdet vha. objektorienterede metoder og dokumenteret med UML notationen.

Da hovedparten af SPU håndbogen omhandler eviggyldige ”Software Engineering” principper vil denne note forsøge at give en opdatering af håndbogen med specifik fokus på anvendelsen af UML i en objektorienteret udviklingsproces.

Noten vil i øvrigt dels beskrive baggrunden og intentionen med SPU konceptet samt give svar på nogle af de spørgsmål og kommentarer jeg har fået i årenes løb.

Vedrørende den oprindelige titel ”SPU – Struktureret programudvikling” så har det senere ærgret mig og de øvrige forfattere, at vi ikke fik kaldt bogen og konceptet for ”SPU – Systematisk Program Udvikling” – da det var denne betydning af ordet ”struktureret” vi oprindelig mente og ikke som det senere er blevet fortolket som værende synonym med ”de strukturerede metoder” og dermed i modsætning til de senere objektorienterede metoder. Disse strukturerede metoder var fremhersekende i 80’erne og står i dag i skyggen af de nyere objektorienterede metoder, der blev udviklet i 90’erne (f.eks. OMT metoden fra 1993, der senere i 1997 blev en hovedbestanddel af UML notationen).

Jeg vil derfor i denne note redefinere forkortelsen SPU til at stå for Systematisk Program-Udvikling.

1.3 UML og objektorienteret udvikling

UML står for Unified Modeling Language, der er navnet på en OMG standard (www.omg.org) for objektorienteret udvikling. OMG (Object Management Group) er en sammenslutning af ca. 800 virksomheder. UML blev udgivet version 1.1 i nov. 1997 og er i 2003 udkommet i version 2.0 [UML2003].

UML anvendes i dag verden over som beskrivelsesværktøj i forbindelse med SW udviklingsprojekter.

UML understøttes af et stort antal værktøjer, der kaldes for CASE værktøjer (Computer Aided Software Engineering). UML er et sæt af objektorienterede begreber med tilhørende grafiske notation, der indgår i et forskellige typer af diagrammer. UML er derimod **ikke** en **udviklingsmetode** eller en **udviklingsproces**.

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

Objektbaseret udvikling er den delmængde af objektorienteret udvikling, der også kan implementeres vha. et ikke objektorienteret programmeringssprog som f.eks. C og assembler. Objekt baseret udvikling kan naturligvis også implementeres i objektorienterede programmeringssprog som f.eks. C++, Java og C#.

Objektorienteret udvikling tilføjer begreber som generalisering/specialisering (nedarvning) og polymorfi.

Det modulbegreb der præsenteres i SPU håndbogen svarer til klassebegrebet i den objektbaserede og objektorienterede udvikling.

1.4 Læsevejledning

Kapitel 2 giver en kort overordnet beskrivelse af de ændringer og opdateringer til de 8 SPU vejledninger, der berøres af UML. I de efterfølgende tre kapitler behandles de tre vejledninger, der i væsentlig omfang påvirkes af UML. Det er hhv. vejledning i struktureret programudvikling, vejledning i kravspecifikation og vejledning i design. Informationen i denne note er udelukkende tænkt som et supplement til vejledningerne i SPU bogen og ikke som en erstatning af disse.

Ønsker man en baggrundsbeskrivelse for SPU konceptet samt en perspektivering så findes denne i afsnit 1.2.

2 Opdatering af SPU vejledninger i relation til UML

Dette kapitel vil kort opsummere ændringerne i de enkelte SPU vejledninger. For de første tre vejledningers vedkommende behandles opdateringerne i relation til UML i hvert sit kapitel.

2.1 Vejledning i struktureret programudvikling (SPU)

Opdateringer til denne vejledning behandles i kapitel 3. Opdateringerne vedrører primært anvendelsen af en iterativ udviklingsmodel ved udvikling af systemer, hvori der indgår software.

2.2 Vejledning i kravspecifikation

Opdateringer til denne vejledning behandles i kapitel 4. Opdateringerne vedrører primært anvendelsen af Use Case teknikken til at finde og beskrive de funktionelle krav med. Use Case notationen indgår som en del af UML.

2.3 Vejledning i design

Opdateringer til denne vejledning behandles i kapitel 5. Opdateringerne vedrører anvendelse af en objektorienteret designmetode baseret på UML notationen ved designarbejdet og en redefinering af designaktiviteterne.

2.4 Vejledning i softwaretest

SPU vejledningen i software test baserer sig på den anerkendte V-model for test, der fokuserer på tidlig testplanlægning før den egentlige testudførelse.

Vejledningen omhandler fire testtyper: modultest, modulintegrationstest, procesintegration og accepttest. I forbindelse med objektorienteret udvikling kaldes modultesten ofte for unittest eller klassetest. Ved objektorienteret udvikling er unit- eller klassetesten, den mindste testenhed, hvor klassens operationer og attributter testes. Det giver ikke mening at teste de enkelte operationer alene, da de fungerer i klassens kontekst og er fælles om klassens attributter.

Vejledningen omhandler primært testprocessen og testdokumentation og en generel introduktion til test. Ved test af objektorienterede systemer bør den derfor suppleres med teori, der omhandler test af objektorienterede systemer.

I forbindelse med udviklingsprocessen XP (eXtreme Programming) lægges der meget vægt på unittesten, hvor man skriver testprogrammet til en klasse før man skriver koden for selve klassen. Dette er ud over parprogrammering (dvs. to sammen) et at hovedelementerne i XP.

2.5 Vejledning i review

Denne vejledning indeholder en god og kort introduktion til review teknikken, der stadig er den ene teknik, der har de største kvalitetsforbedrende virkninger og den største effekt i forhold til indsatsen. Denne vejledning vil ikke på nogen måde være påvirket af UML, men det vil være UML baseret dokumentation man reviewer.

Review teknikken er mindst ligeså vigtig i en iterativ udviklingsproces som i den mere traditionelle vandfaldbaserede og dokumentbaserede udviklingsmodel.

2.6 Vejledning i projektstyring

Denne vejledning indeholder eviggyldige sandheder og den præsenterede teori er således ikke direkte påvirket af UML notationen.

Derimod vil anvendelsen af en iterativ udviklingsproces baseret på Use Case teknikken have indvirkning på projektstyringen. Dette skyldes at Use Case teknikken ud over at være en væsentlig teknik til at udtrykke de funktionelle krav med også kan anvendes til at definere iterationerne ud fra. Projektet planlægges således at hver iteration afsluttes med et kørende delsystem, der kan være enten en intern leverance eller en ekstern leverance. Da Use Cases beskriver funktionalitet som systemet udfører for brugerne er de netop meget anvendelige til definere disse delleveringer med. Hvor SPU Håndbogen baserer sig på en dokumentstyret udviklingsproces vil SPU-UML i større udstrækning basere sig på en iterativ og risikostyret udviklingsproces, der baser sig på kørende delsystemer eller prototyper, der udvikler sig til det endelige produkt eller system.

Stephen Biering-Sørensen, der er medforfatter til SPU Håndbogen, har i øvrigt en ny bog på vej om ”IT projektledelse”, der supplerer og uddyber denne vejledning med den nyeste viden på området [Biering2004].

2.7 Vejledning i programdokumentation

Denne vejledning beskriver SPU reglen ”Dokumenter undervejs”, der siger at dokumentation af softwaren er en løbende proces i en moderne udviklingsproces. Dette er endnu vigtigere i en udviklingsproces, der baserer sig på iterationer, hvor næste iteration anvender dokumentation fra den foregående iteration.

Anvendelsen af UML som udviklingsnotation vil lette dokumentationsarbejde, da der nu eksisterer en standardnotation UML, der kan anvendes til at dokumentere softwaredesignet og med mulighed for også at vise den tilhørende hardware vha. UML deploymentdiagrammer.

Det viste eksempel på en indholdsfortegnelse for en programdokumentation kan videreudvikles hen imod UML ved f.eks. at anvende ideerne fra ”4+1” view modellen for dokumentation af softwarearkitektur [Krutchen85].

2.8 Vejledning i konfigurationsstyring

Denne vejleding giver en introduktion til konfigurations- og versionsstyring, der er en meget vigtig udviklingsaktivitet, specielt i de stadig større projekter der i dag igangsættes.

Her vil anvendelsen af objektorienteret udvikling bevirke at de mindste kodeenheder man ønsker at dokumentere er klasser og ikke som tidligere enkeltfunktioner. UML har endvidere indført et pakkebegreb, der også vil lette administrationen af koden.

Anvendelsen af UML vil derudover medføre et antal UML diagrammer og den tilhørende information. Denne dokumentation skal versions- og konfigurationsstyres. Her får man hjælp af CASE værktøjerne, der (næsten) alle har grænseflader til konfigurationsstyringsværktøjer.

3 Vejledning i Systematisk Program-Udvikling opdateret mht. UML

Den største opdatering i håndbogens ”Vejledning i struktureret programudvikling” vedrører en modifikation af SPU udviklingsmodellens således at den i SPU-UML udgaven beskriver en iterativ og Use Case baseret udviklingsmodel.

3.1 Hvad er systematisk programudvikling (SPU-UML)?

SPU og SPU-UML konceptet består i al sin enkelhed i at anvende følgende 8 elementer i forbindelse med udvikling af systemer, hvori der indgår software:

SPU-UML

1. Benyt en udviklingsmodel
2. Udarbejd en kravspecifikation
3. Design før kodning
4. Planlæg test
5. Anvend reviewteknikken
6. Foretag projektstyring
7. Dokumenter undervejs
8. Foretag konfigurationsstyring

Figur 1. SPU-UML konceptets 8 elementer

Disse otte punkter er udformet som eviggyldige sandheder, man ikke kommer uden om, hvis man vil gennemføre en succesfuld system- og softwareudvikling. Hvert af disse elementer er beskrevet i sin egen vejledning i SPU håndbogen.

Dette var i øvrigt en af grundideerne bag udformning af SPU konceptet, at det skulle være et metodeuafhængigt koncept, dvs. en slags ”knagerække” hvorpå, man efterfølgende kunne ”hænge” konkrete metoder, teknikker og værktøjer til gennemførelse af de forskellige aktiviteter i konceptet. En sådan metode og teknik er objektorienteret udvikling med anvendelse af UML notationen.

SPU-UML regel 1: Benyt en udviklingsmodel

Her vil man typisk i forbindelse med objektorienteret udvikling i dag anvende en udviklingsmodel, der er baseret på gennemførelse af et antal iterationer, der planlægges og styres ud fra Use Cases, der er identificeret i forbindelse med kravspecifikationen. Man taler også her om en Use Case drevet udviklingsproces.

SPU-UML regel 1 udtrykker, at man skal anvende en udviklingsmodel men ikke, hvordan den præcist skal se ud.

Udviklingsmodellen i SPU var begrænset til udelukkende at omhandle softwareudviklingen i et projekt, hvilket af mange senere er ønsket udvidet til at omhandle den totale udvikling af et produkt, der både omfatter hardware- og softwareudvikling.

SPU modellen er også af mange blevet læst som en vandfaldsmodel, hvad den da også ligner meget, men den var oprindelig tænkt også at kunne indgå i en iterativ udvikling, hvilket dog ikke fremgår specifikt nok i håndbogen. Dette vil denne note derfor belyse mere detaljeret.

SPU-UML noten præsenterer en modificeret udviklingsmodel SPU-UML. Denne modificerede udviklingsmodel er en iterativ udviklingsmodel, der omfatter udvikling af komplette apparatsystemer med både hardware- og softwareudvikling. SPU-UML modellen er en Use Case drevet udviklingsmodel, hvor Use Case teknikken anvendes ved flere forskellige aktiviteter.

SPU-UML regel 2: Udarbejd en kravspecifikation

Før man går i gang med et hvilket som helst projekt bør der forelægge en form for kravspecifikation. I en iterativudviklingsmodel kan man dog også i nogle tilfælde iterere over udarbejdelsen af kravspecifikationen.

I SPU-UML foreslås det at kravspecifikation udarbejdes vha. Use Case teknikken. Dette har vist sig at give bedre specifikationer, hvor specifikationen tager udgangspunkt i opfyldelse af funktioner, der er til gavn for systemets aktører og interessenter.

Use casene anvendes i udviklingsforløbet til at definere iterationerne ud fra og er dermed til at styre projektets leverancer ligesom Use Casene danner udgangspunkt for specifikation af accepttesten.

SPU-UML regel 3: Design før kodning

Denne regel er ved at være almindelig anerkendt blandt de fleste professionelle softwareudviklere. SPU vejledningen præsenterer her nogle generelle og metodeuafhængige regler for designarbejdet.

I SPU-UML anvendes der en objektorienteret metode som designmetode og designet udarbejdes og dokumenteres vha. UML notationen.

SPU-UML regel 4: Planlæg test

Denne regel omhandler V-modellen for test, hvor det fremgår at test skal planlægges sammen med den aktivitet, der senere i forløbet testes. Reglen er samtidig med til at sætte fokus på softwaretest, der længe har været en overset og undervurderet disciplin.

Anvendelsen af Use cases til kravspecifikation letter den tilhørende accepttestplanlægning, da Use Casene netop udtrykker selvstændig funktionalitet, der er nyttige for f.eks. kunden og brugerne af systemet og derfor egner sig til at blive anvendt ved accepttesten.

SPU-UML regel 5: Anvend reviewteknikken

Denne regel fokuserer på anvendelsen af reviews som kvalitetsforbedrende aktivitet i et moderne udviklingsprojekt. Dette er en metodeuafhængig teknik, der kan anvendes på alt skriftligt materiale.

SPU-UML regel 6: Foretag projektstyring

Denne regel sætter fokus på styrings og ledelsesaspektet af et softwareudviklingsprojekt, der er en særdeles vigtig aktivitet for at opnå succes i et projekt.

SPU-UML regel 7: Dokumenter undervejs

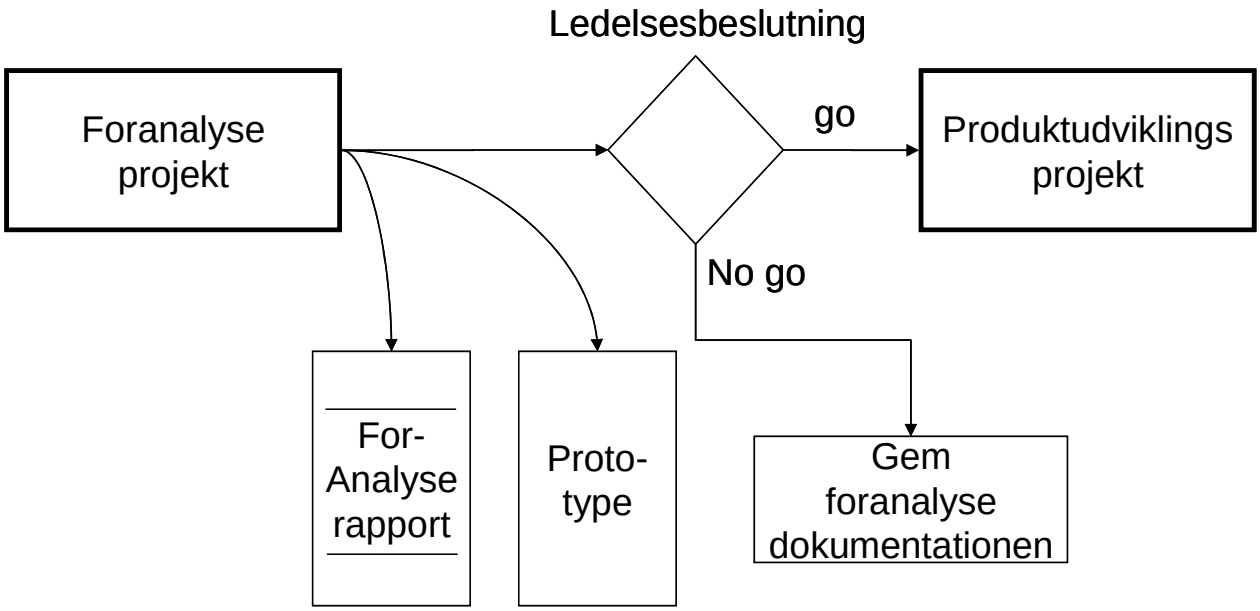
Reglen her sætter fokus på udarbejdelse af softwaredokumentationen for et projekt og præsenterer denne som en løbende proces. Her vil anvendelsen af UML lette dette arbejde idet der nu er en verdensstandard for at beskrive et objektorienteret design.

SPU-UML regel 8: Foretag konfigurationsstyring

Denne regel omhandler det at have styr på sine dokumenter og software og de versioner og konfigurationer denne indgår i. Dette er også en metodeuafhængig og meget vigtig aktivitet.

3.2 Anvendelse af SPU-UML modellen i et produktudviklingsforløb

Figur 2 viser et typisk produktudviklingsforløb for et apparat eller system. Udviklingsforløbet starter med et foranalyse projekt (engelsk: *feasibility study*), der har til formål at afklare vigtige usikkerheder vedrørende projektets gennemførlighed. Dette foranlyseprojekt afsluttes med udarbejdelsen af en foranalyserapport, der f.eks. kan indeholde en foreløbig produktkravspecifikation. Forananalyserapporten vil typisk også indeholde analyse- og foreløbig designdokumentation, der kan udarbejdes vha. UML. Et andet typisk resultat er en prototype af det ønskede apparat eller system. Foranalyseprojektet danner udgangspunkt for en ledelsesbeslutning vedrørende igangsætning af et egentligt produktudviklingsprojekt. Vælger man at stoppe her så vil man typisk gemme resultaterne af foranalyseprojektet sammen med begrundelsen for ikke at gå videre.



Figur 2. Et typisk produktudviklingsforløb

En vigtig pointe er, at SPU-UML udviklingsmodellen kan anvendes både under foranalyseprojektet og under selve produktudviklingsprojektet. Det er en af de store fordele ved den objektorienterede fremgangsmåde og UML notationen at man vha. Nodes og klassebegrebet kan modellere såvel software- som hardwaredelen af et projekt. Der vil være forskel på fokus mellem foranlyse- og produktudviklingsprojektet, hvor man i foranalyseprojektet vil fokusere på at opgaven kan løses (engelsk: *proof of concept)* så vil man i produktudviklingsprojektet lægge mere vægt på et stabilt design samt en udførlig test og dokumentation.

Der findes også produktudviklingsprojekter, der igangsættes ud fra et kort produktoplæg eller produktbeskrivelse. Her vil første aktivitet være at udarbejde en kravspecifikation for det ønskede produkt.

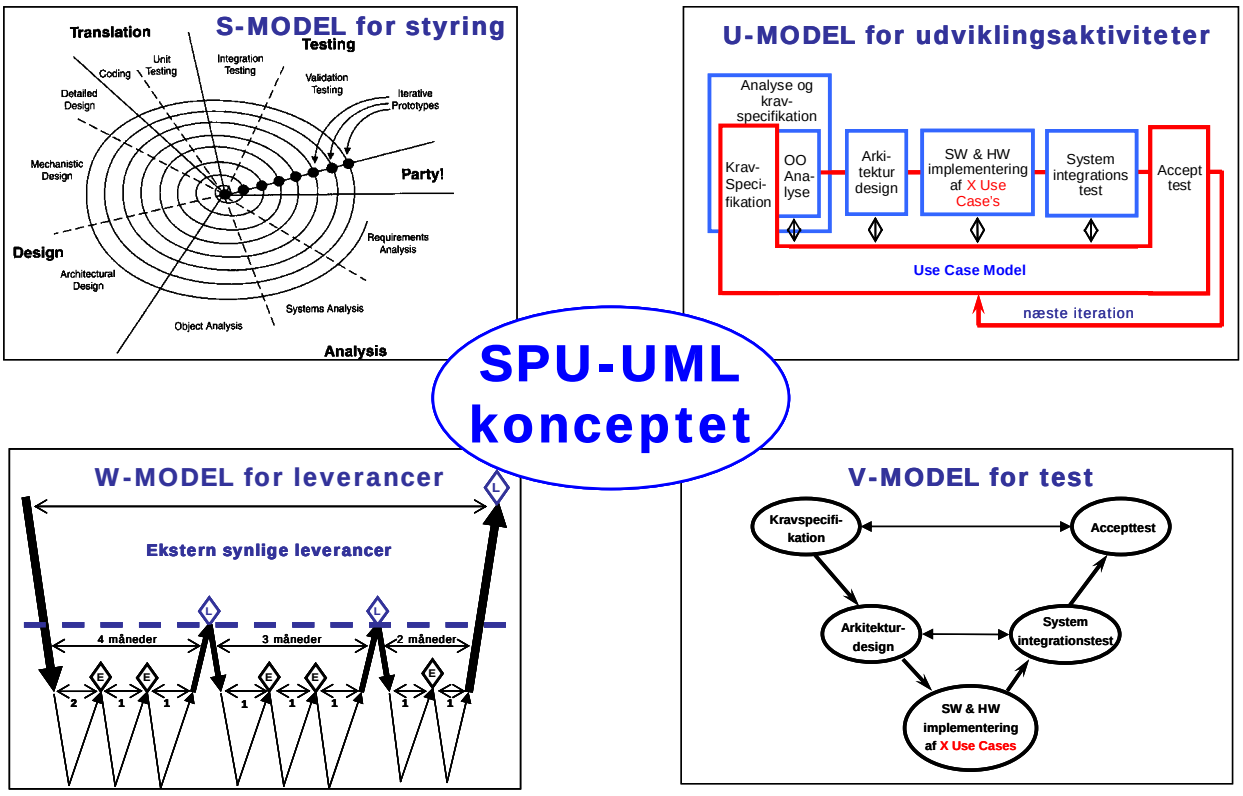
3.3 SPU-UML udviklingsmodellen

Den modificerede SPU-UML udviklingsmodel er først og fremmest baseret på en iterativ udviklingsmodel, der drives ud fra et sæt definerede Use Cases. Use Case teknikken er omtalt i afsnit 4.1 i forbindelse med opdatering af vejledningen i kravspecifikation.

SPU-UML modellen indeholder den traditionelle vandfaldsmodel som et specialtilfælde af en iterativ model, hvor der kun udføres én iteration.

Hvor SPU udelukkende dækkede softwareudviklingsdelen af et projekt så dækker SPU-UML udviklingsmodellen den totale udvikling af produkter eller systemer dvs. både software- og hardwareudvikling.

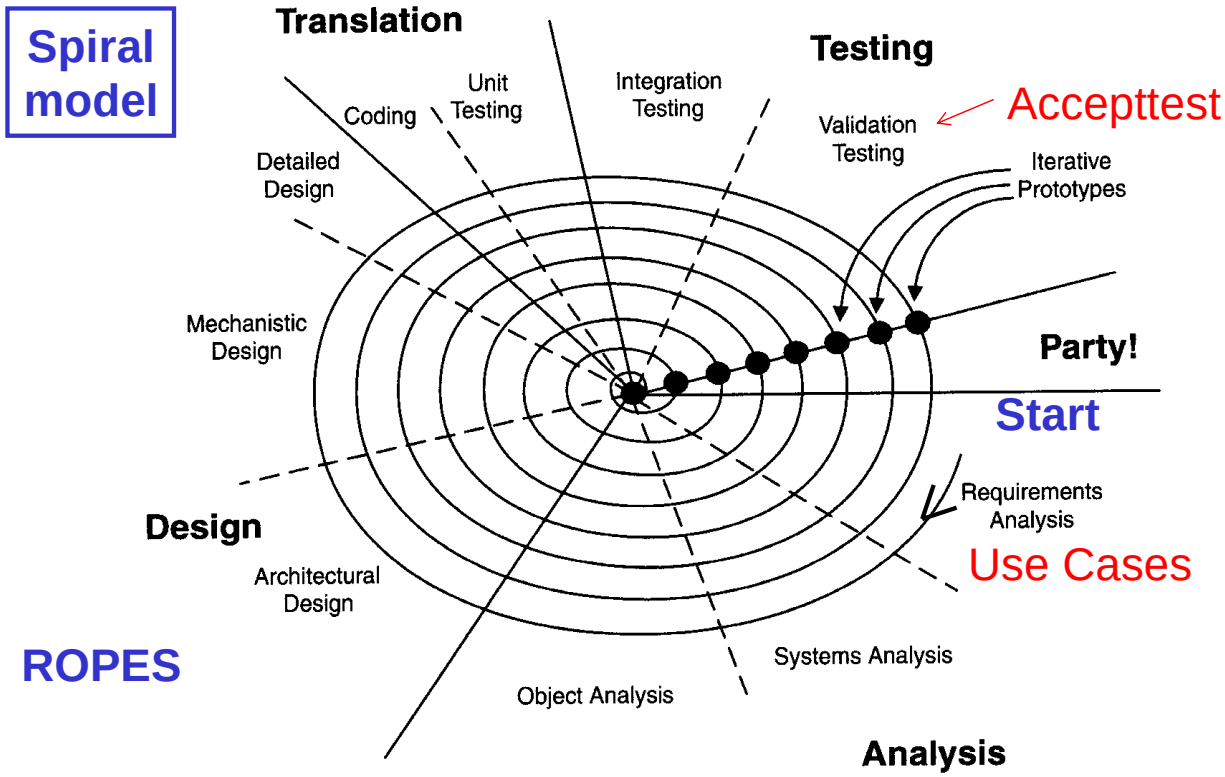
Figur 3 viser SPU-UML udviklingsmodellen beskrevet vha. fire perspektiver, der hver fokuserer på et bestemt aspekt af udviklingsmodellen. Disse fire perspektiver er bundet sammen af SPU-UML konceptets 8 elementer. For at lette referencen til disse perspektiver gives de hvert et kort modelnavn hhv. S-Model for spiralmodel, U-Model for udviklingsaktiviteter, W-model for leverancer og V-model for test.



Figur 3. SPU-UML udviklingsmodellens perspektiver

3.3.1 Spiralmodel (S-model)

Iterative udviklingsmodeller anskueliggøres ofte vha. en spiral, hvilket stammer fra Barry Boehms berømte spiralmodel [Boehm88]. Dette gælder også for udviklingsmodellen ROPES – Rapid Object-oriented Process for Embedded Systems [Douglass99], der er en OO udgave af en spiralmodel, der er specielt beregnet til udvikling af indlejrede systemer.

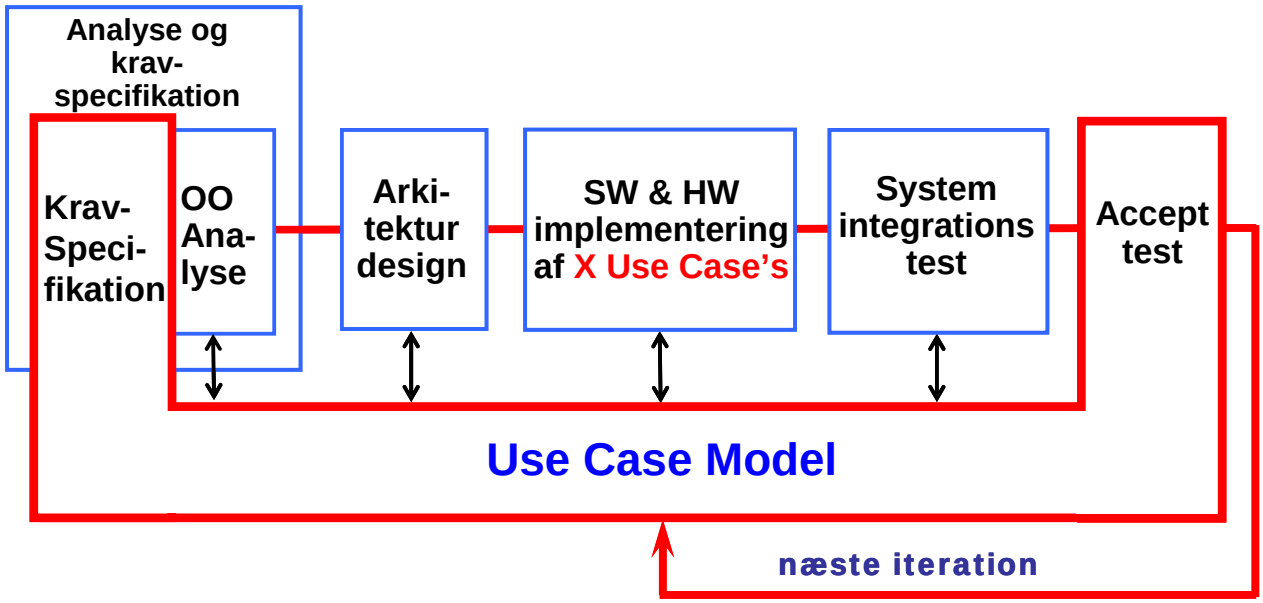


Figur 4. S-model for styring (ROPES)

Anvendelsen af en iterativ udviklingsmodel er vigtig af flere årsager. For det første kan en sådan model være med til at nedsætte risikoen, idet de mest risikobetonede dele af projektet udvikles tidligt, hvor der er tid til at ændre på kritiske faktorer og evt. stoppe projektet i tide, hvis det ikke kan realiseres. For det andet vil det synliggøre udviklingsforløbet, idet hvert iteration er baseret på et kørende delsystem, hvor alle aktiviteter i princippet er gennemført fra kravspecifikation over design til kodning og test. For det tredje vil anvendelsen af en sådan model operere med mange og korte lærercykler, der er specielt vigtige når man tager ny teknologi som f.eks. objekt-orienteret udvikling i anvendelse.

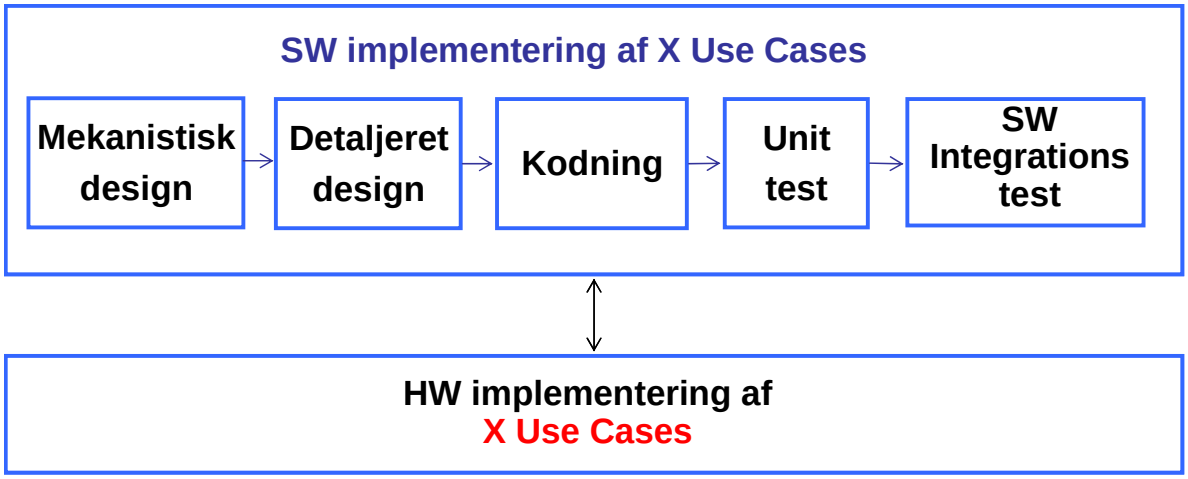
3.3.2 Udviklingsmodel (U-model)

Figur 5 viser hvorledes de traditionelle udviklingsaktiviteter arbejder sammen med en Use Case model, der specificerer de Use Cases systemet er opdelt i. Figuren viser også hvorledes man i en næste iteration kan vende tilbage til en vilkårlig af de tidligere gennemførte aktiviteter inklusiv kravspecifikationen.



Figur 5. U-model for udviklingsaktiviteter

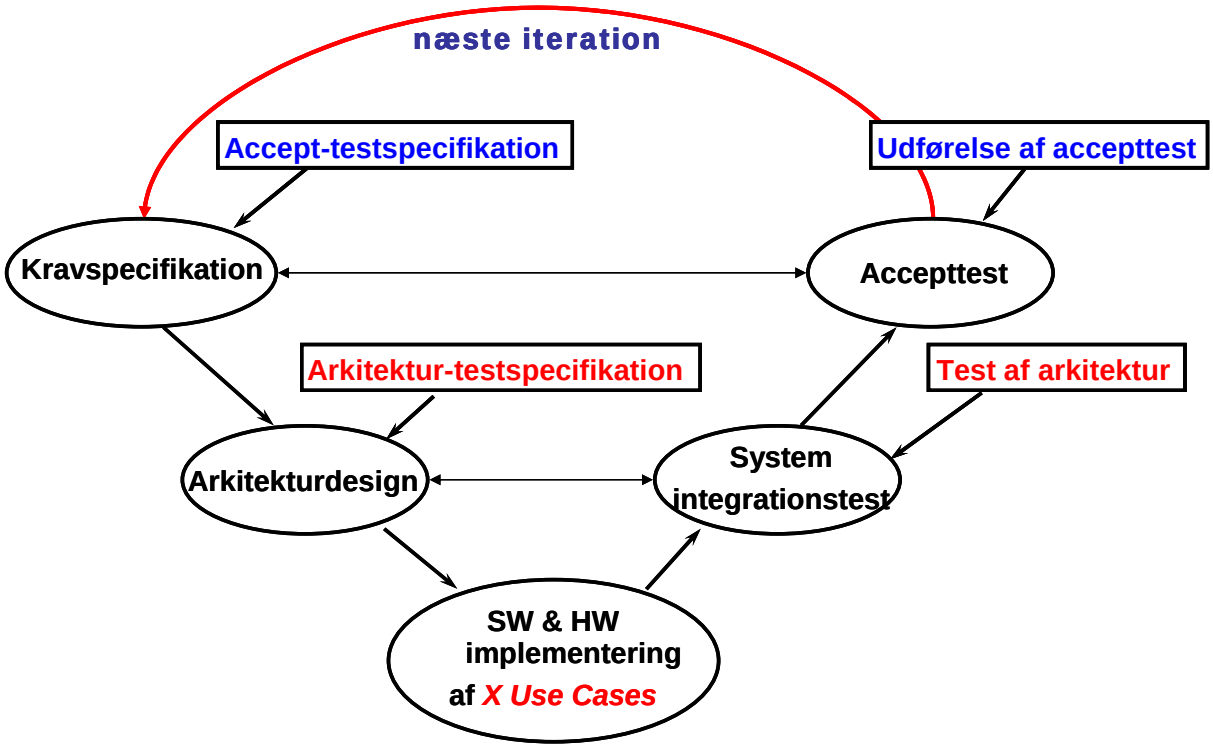
Figur 6 viser hvorledes software- og hardwareudviklingen af den valgte iteration forløber parallelt efter gennemførelse af arkitekturdesignaktiviteten, hvorefter SW og HW integreres i system-integrationstest aktiviteten. SW aktiviteterne er her benævnt som i ROPES modellen.



Figur 6. Oversigt over SW & HW implementeringsaktiviteterne

3.3.3 Testmodel (V-model)

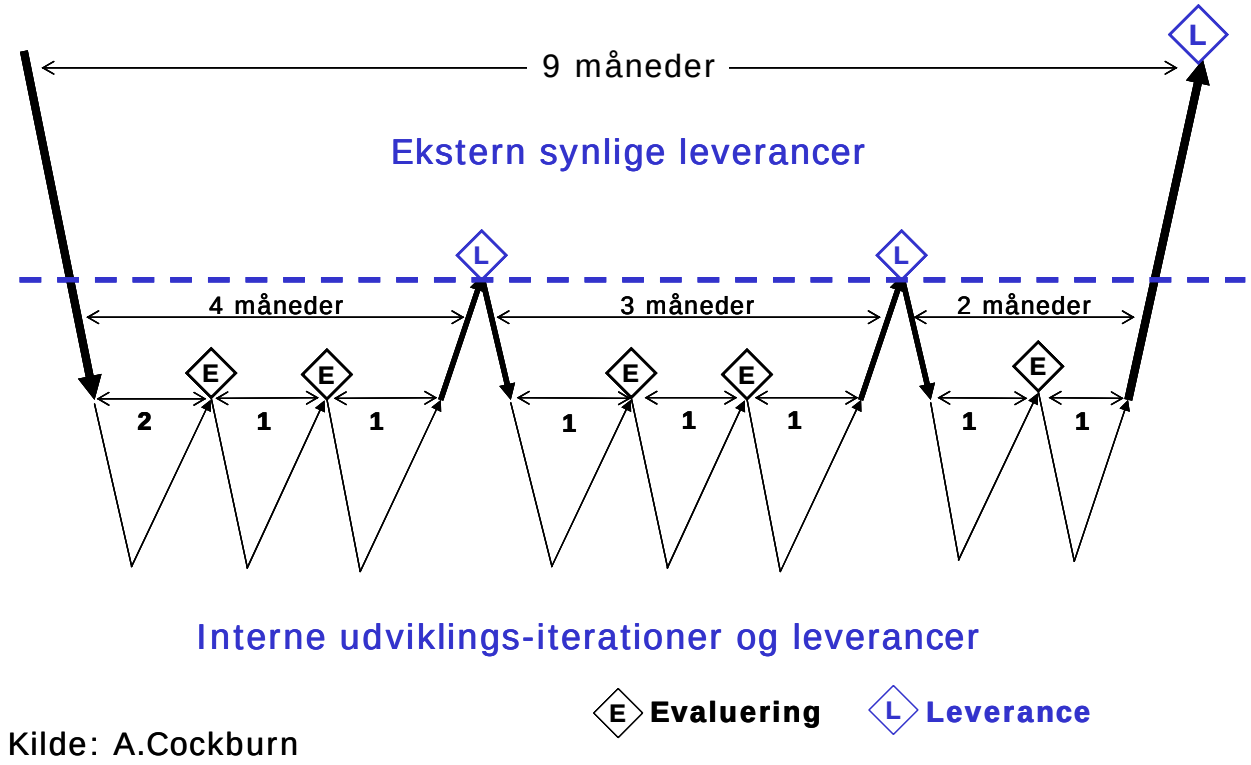
I SPU-UML modellen er det vigtigt at bibeholde V-modellen for test, der giver en god illustration af forholdet mellem tidlig testforberedelse og de senere testudførelse. Figur 7 viser hvorledes denne model også kan anvendes i forbindelse med iterativ udvikling – her gentages V'et blot flere gange.



Figur 7. V-model for test

3.3.4 Leverancemodel (W-model)

Figur 8 viser W-modellen for leverancer, der er en model over de interne evalueringer samt de interne og eksterne leverancer, der kan forekomme i en iterativ udviklingsmodel. Modellen er beskrevet af Alistair Cockburn [Cockburn-W]. Den mindste iteration består i gennemførelse af en V-model, der resulterer i en intern evaluering af et kørende delsystem. Efter et antal af disse kan man have en mere formel og synlig ekstern leverance, der f.eks. kan anvendes til pilottest og pilotforsøg med de rigtige brugere af systemet. Et antal af disse kan så kombineres til en officiel "release" af et produkt.

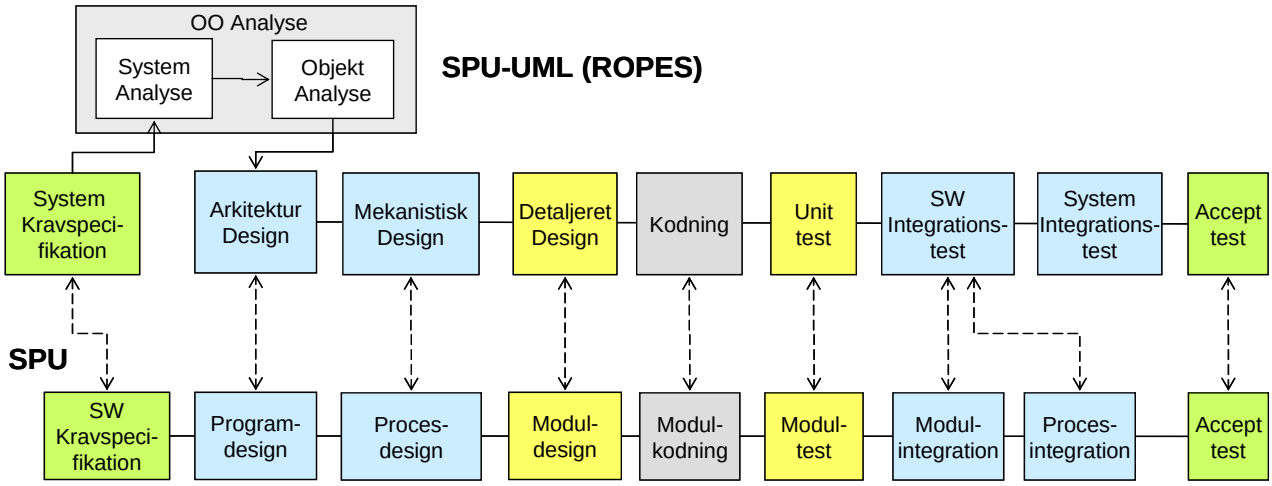


Figur 8. W-model for leverancer

I eksemplet på Figur 8 er der således 8 udviklingsiterationer, hvoraf de tre er ekstern synlige og hvoraf den af sidste af disse fører til en frigivelse af produktet. I dette ni måneders projekt er der således 8 vigtige milepæle at styre efter og vel at mærke milepæle, der demonstrer fremdrift både overfor ledelsen, kunderne og overfor udviklerne selv via kørende systemer eller delsystemer.

3.4 Udviklingsaktiviteter

Figur 9 viser øverst udviklingsaktiviteterne i SPU-UML og nederst som de fremgår af SPU håndbogen. SPU-UML aktiviteterne er benævnt efter ROPES modellen. Som det fremgår af figuren er der i SPU-UML (ROPES) tilføjet OO analyse aktiviteter systemanalyse og objektanalyse. De viste SPU-UML aktiviteter beskrives kort i de følgende afsnit.



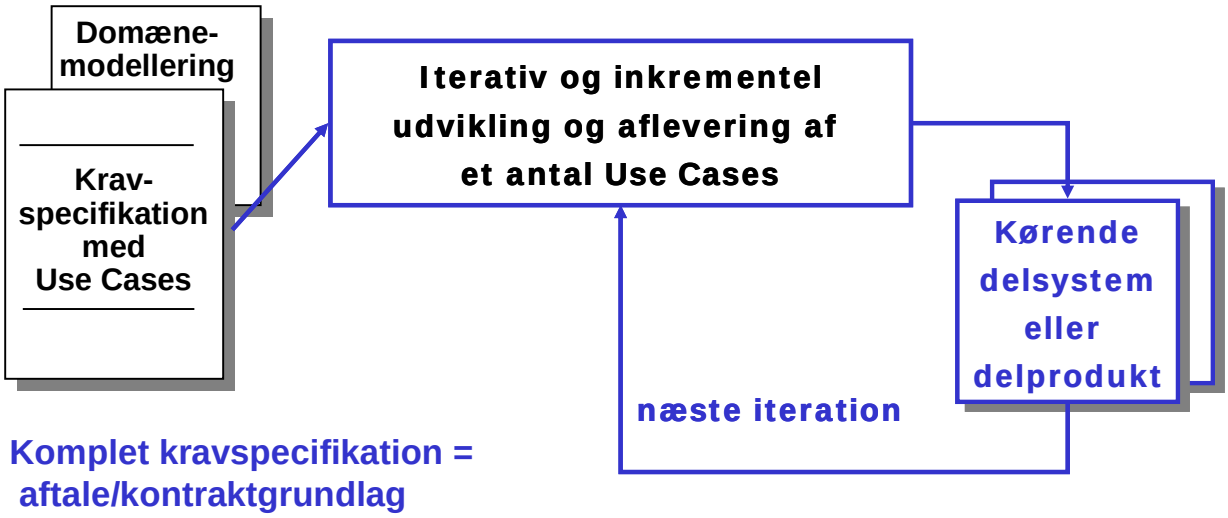
Figur 9. Sammenhæng mellem SPU-UML og SPU aktiviteter

3.4.1 Kravspecifikation

I SPU-UML kan kravspecifikationen dække komplette systemer eller produkter dvs. både software og hardwaredelen af et system. Hvorimod SPU alene omhandlede en SW kravspecifikation. I begge tilfælde vil kan have glæde af at udarbejde en Use Case baseret kravspecifikation.

Der er mange forskellige typer af udviklingsprojekter og udviklingssituationer, hvorfor den udviklingsmodel man anvender til det konkrete projekt, bør tage udgangspunkt i den aktuelle situation.

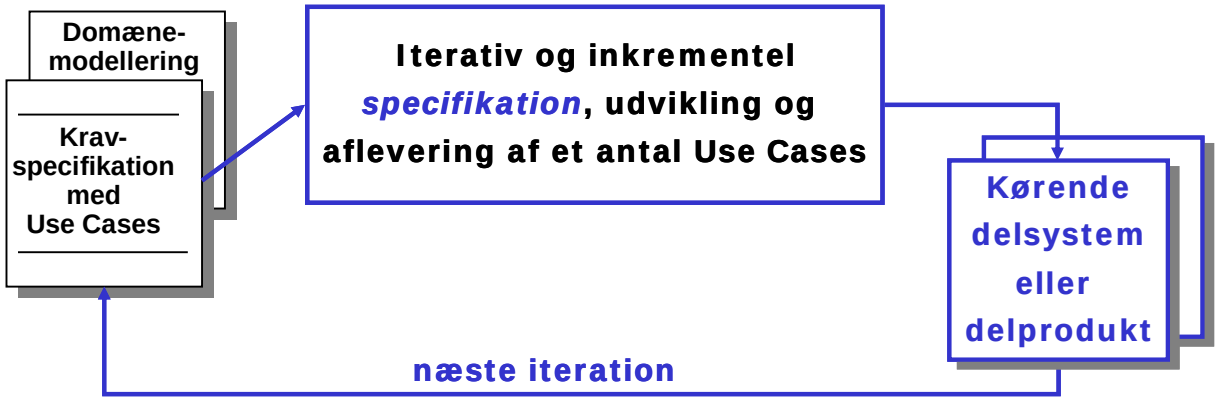
Figur 10 viser f.eks. en udviklingssituation hvor der udarbejdes en komplet Use Case baseret kravspecifikation og evt. en domænemodel før der påbegyndes et iterativt udviklingsforløb, der defineres ud fra de specificerede Use Cases.



Figur 10. Iterativ udvikling ud fra en komplet kravspecifikation

Denne situation kan f.eks. være nyttig i forbindelse med anvendelse af underleverandører til et projekt, hvor kravspecifikationen vil være det formelle aftalegrundlag for definering af priser og tidsplaner. En anden situation, hvor denne model kan anvendes er de tilfælde, hvor kravene er forholdsvis velkendte, f.eks. fordi man tidligere har udviklet tilsvarende produkter eller systemer.

Er der derimod tale om en komplet egenudvikling af et nyt og ukendt produkt så vil der være brug for at anvende en iterativ udviklingsmodel, der også itererer over kravspecifikationen, som vist på Figur 11.

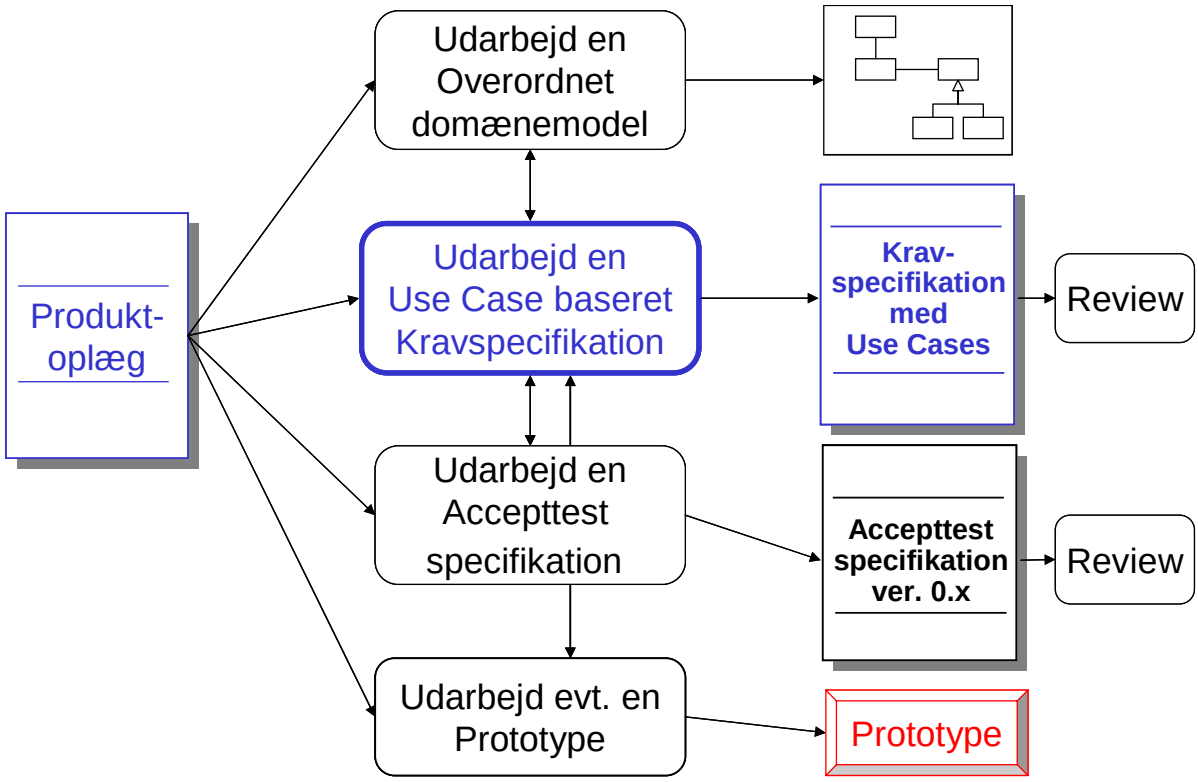


Figur 11. Iterativ udvikling også over kravspecifikationen

Det optimale vil være at projektgruppen sammen med ledelsen ved opstart af et projekt, fastlægger den konkrete udviklingsmodel og dermed definerer hvilke aktiviteter, der skal udføres i det konkrete projekt. Derved opnår man en situationsbestemt udviklingsmodel, der tager højde for projektets specifikke egenskaber.

Den viste domænemodellering består i at udarbejde et eller flere UML klassediagrammer, der beskriver domænets begreber som klasser og viser deres sammenhæng. For et hospitalssystem kan det f.eks. være begreber som patient, afdeling, diagnose.

Figur 12 viser de aktiviteter der er relevante i forbindelse med specifikationsfasen af et projekt. Hovedproduktet er selve kravspecifikationen, der vil være baseret på anvendelse af Use Case teknikken. Parallelt med udarbejdelsen af denne kan man med fordel udarbejde en overordnet domænemodel, der vha. klasser beskriver de objekter, de findes i det aktuelle domæne og viser deres sammenhæng vha. et UML klassediagram. Dette er med til at definere systemets begreber, der så anvendes konsistent i forbindelse med beskrivelse af systemets Use Cases. Parallelt med kravspecifikationen påbegyndes udarbejdelse af en accepttestspecifikation ifølge V-model for test. Dette har den fordel at kravspecifikationen gøres testbart og dermed forbedres kvaliteten af denne.



Figur 12. Specifikationsaktiviteter

For nogle systemer vil det også på dette tidspunkt være meget nyttigt at udarbejde en prototype, det kan være over brugergrænsefladerne eller det kan være over indviklede algoritmer. Til højre på figuren ses de produkter, der er resultatet og hver af disse kan gøres genstand for et review. Specielt bør review af kravspecifikationen være obligatorisk, men det kan også være nyttigt at reviewe accepttestspecifikationen.

For apparatudvikling kan skrivning af en brugervejledning på dette tidlige tidspunkt også være et nyttigt bidrag til at illustrere produktets funktionalitet.

3.4.2 Systemanalyse

Systemanalyseaktiviteten i SPU-UML og ROPES omhandler udførelse af en objektorienteret analyse af et komplet system eller produkt, der både omfatter hardware og software. Aktiviteten udføres på baggrund af en kravspecifikation, der ligeledes er for hele systemet eller produktet. I de tilfælde, hvor det hovedsageligt er et softwareudviklingsprojekt til en standard hardwareplatform som f.eks. en PC så vil denne aktivitet falde ud. Systemet modelleres her vha. klassebegrebet i OO, der diagrammeres vha. UML's klassediagrammer. Hardwaredelen kan modelleres vha. UML's deploymentdiagrammer, der f.eks. kan vise den fysiske opdeling på processorer ligesom UML's klassediagrammer kan anvendes til at diagrammere hardwaren på blokdiagramniveau. For store og eller komplekse systemer kan man her på basis af systemanalysen udarbejde selvstændige software og hardware kravspecifikationer.

3.4.3 Objektanalyse

Objektanalyseaktiviteten i SPU-UML og ROPES omhandler udførelse af en objektorienteret analyse af softwaredelen af projektet. Ud fra kravspecifikationen udarbejdes der her en teknologiuafhængig objektmodel af funktionaliteten uden hensyn til implementeringsteknologien, der tilføjes i de efterfølgende designaktiviteter.

3.4.4 Arkitekturdesign

I SPU-UML og ROPES arkitekturdesignaktivitet fastlægges systemets arkitektur med input fra hhv. systemanalyse og objektanalyse aktiviteterne. Nu skal der vælges konkrete løsninger, der kan leve op til kravene i kravspecifikationen dvs. både de funktionelle krav beskrevet vha. Use Cases og de øvrige ikke funktionelle krav. I denne aktivitet fastlægges såvel hardware- som softwarearkitekturen.

Her anvendes f.eks. UML's deployment-, komponent-, klasse-, tilstands- og sekvensdiagrammer. Ud fra objektanalysen opdeles softwaren i et antal parallelle processer eller task og deres proceskommunikationsform fastlægges.

3.4.5 Mekanistisk design

Mekanistisk design i SPU-UML og ROPES omhandler design af samspillet mellem et antal klasser, der typisk vil indgå i en proces eller et task. Ved denne aktivitet vil man typisk anvende et eller flere designmønstre f.eks. GoF mønstrene [GoF94], der er veldokumenterede løsninger på designproblemer.

3.4.6 Detaljeret design

Detaljeret design i SPU-UML og ROPES omhandler design af en enkelt klasses algoritmer og datastrukturer. Ønsker man at diagrammere algoritmen for en kompliceret operation kan man her anvende UML's aktivitesdiagrammer. Detaljeret design udføres kun for komplicerede klasser hvorimod simple klasser kodes direkte ud fra klassediagrammerne fra den mekanistiske design.

3.4.7 Kodning

Her kodes de klasser, der er identificeret i de tidligere aktiviteter. Er der udført detaljeret design af klassen så omsættes denne til kode og der tilføjes f.eks. funktioner til oprettelse og nedlæggelse af objekterne (C++ constructor og desctructor) ligesom der tilføjes de konkrete datatyper for operationernes parametre og returtyper. Klassens attributter kodens vha. programmeringssprogets datatyper og vha. evt. klassebiblioteker. Klassens associationer til andre klasser kodes og der tilføjes de nødvendige operationer til at vedligeholde f.eks. mange til mange associationer.

3.4.8 Unittest

Unittest i SPU-UML og ROPES består i test af klasser idet en unit er lig med en klasse. En klasse er den mindste enhed hvor det giver mening at udføre en selvstændig test. Ved test af klassen testes klassens operationer og deres samspil gennem klassens attributter. Er der tale om objektorienteret udvikling, hvor der anvendes nedarvning så skal man i testen af de specialiserede klasser tage passende hensyn til de klasser man nedarver fra. Da der kan være såvel meget simple klasser som meget komplicerede klasser vil det ikke give mening at have en udviklingsmetode, der siger at alle klasser skal testes. De simple klasser kan f.eks. testes i forbindelse med en integrationstest eller ved inspektion.

3.4.9 SW-integrationstest

Her integreres og testes softwaren efter en fastlagt teststrategi, der er beskrevet i en testspecifikation. I denne aktivitet testes samspillet mellem de identificerede klasser. Det er her oplagt at teste de enkelte processer eller task hver for sig som sekventielle programmer, før de kobles sammen til et multiprogram bestående af flere processer.

3.4.10 Systemintegrationstest

Her integreres og testes hele systemet dvs. både hardware og software delen af systemet. Dette gøres trinvist efter en fastlagt teststrategi, der er beskrevet i en testspecifikation. Systemintegrationstesten tester det arkitekturdesign, der er udarbejdet under arkitekturdesignaktiviteten.

3.4.11 Accepttest

Accepttesten er den sluttest, der afslutter de iterationer, der har eksterne leverancer. Ved accepttesten demonstreres det, at det delsystem eller delprodukt der omhandles af accepttesten lever op til de krav, der er beskrevet i de Use Cases som accepttesten omhandler. Accepttesten kan derforuden også omhandle de ikke funktionelle krav i kravspecifikationen.

4 Vejledning i kravspecifikation opdateret mht. Use Cases

SPU vejledningen i kravspecifikation er skrevet med udgangspunkt i IEEE-830, der er en standard for udarbejdelse af kravspecifikationer for software. Standarden indeholder en overordnet skabelon for en kravspecifikation med et sæt af forskellige forslag til hvorledes man kan beskrive de funktionelle krav. Den nyeste udgave af IEEE-830 (1999) indeholder desværre endnu ikke Use Cases som beskrivelsesmåde, men dette er bestemt er et spørgsmål om tid før det kommer med.

I SPU-UML vil håndbogens kapitel 3.2 ”Hvordan formuleres kravene” skulle suppleres med Use Case teknikken som her introduceres. Under kapitel 4. ”Indhold af kravspecifikationen” vil afsnit 3. ”Specifikke krav” blive erstattet med specifikationer af de enkelte Use Cases.

4.1 Use Case teknikken

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

Use Case teknikken er udviklet af svenskeren Ivar Jacobson og først beskrevet i bogen ”Object-Oriented Software Engineering – A Use Case driven approach” [Jacobson92].

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

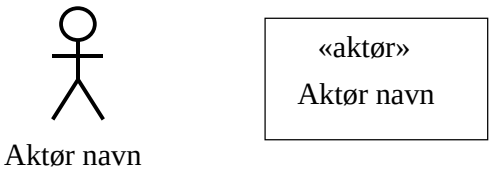
Anbefalingen er her, at man starter med at anvende den basale Use Case notation og beskriver de funktionelle krav til systemet ved hjælp af denne notation. Derefter kan man overveje, om den udvidede notation vil give en bedre model af kravene og derfor kun anvende denne, hvis dette er tilfældet.

4.1.1 Basal Use Case notation

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

En aktør kan enten være en person, et andet system eller en hardware enhed. En aktør er pr. definition udenfor det system, der skal udvikles, men er i samspil med systemet. En aktør beskriver således en grænseflade til det system, man udvikler.

Til at navngive en personaktør anvendes de roller, personen har overfor systemet. Har en given person flere roller, så optræder hver rolle som en aktør. En aktør vises som en tændstiksfigur med aktørens navn påført under figuren. Alternativt kan man vælge at vise aktøren som en firkant med stereotypen «aktør».

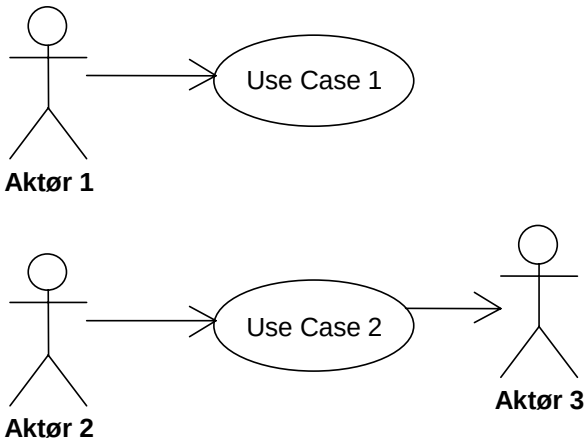


Figur 13. To alternative notationer for en aktør

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

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

Eksempler på Use Cases for et patientovervågningssystem, hvor en aktør med navnet Sygeplejerske f.eks. kan udføre Use Casene: ”Indlæg Patient”, ”Specificer overvågning” og ”Udskriv patientrapport”.

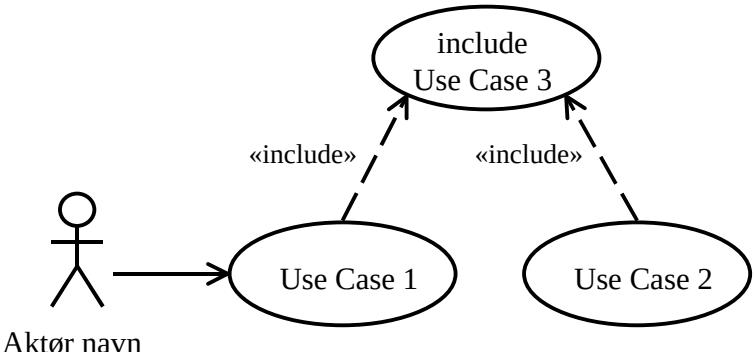


Figur 14. Use Case diagram

For hver Use Case udarbejdes der en specifikation, der præcist specificerer Use Casens funktionalitet med en beskrivelse af normalforløbet og alle de undtagelser, der kan forekomme. I nogle tilfælde kan beskrivelsen være suppleret med forskellige diagrammer (f.eks. sekvens- eller tilstandsdiagrammer).

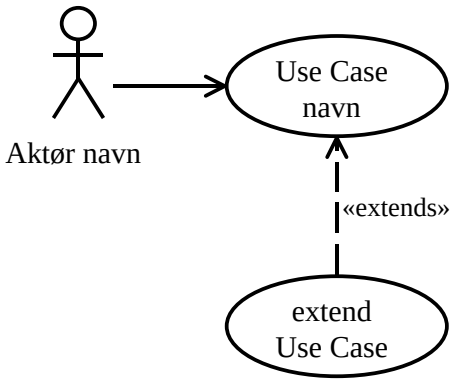
4.1.2 Udvidet Use Case notation

Er der fælles funktionalitet, der indgår i flere Use Cases, så beskrives denne funktionalitet for sig selv som en ”include” Use Case, der er en slags ”hjælpe” Use Case. Det har den store fordel, at man undgår gentagelser i specifikationen.



Figur 15. Use Case diagram med include

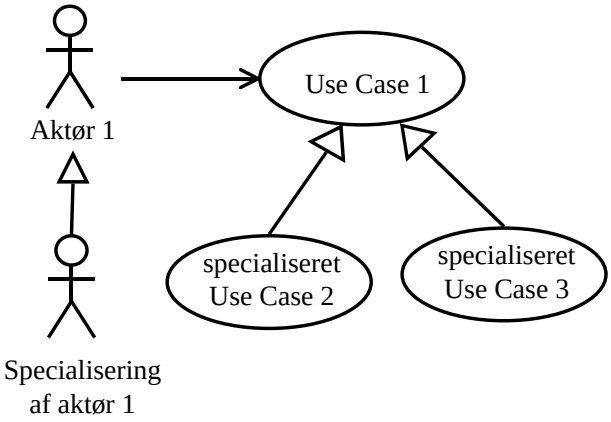
En anden udvidelse til den basale notation beskrives vha. en ”extend” Use Cases, der er en udvidelse til en eksisterende selvstændig Use Case. En ”extend” Use Case kan beskrive enten komplekse fejlforløb eller optionale udvidelser, som alle kunder ikke ønsker, eller beskrive funktionalitet, der først ønskes i senere versioner af systemet.



Figur 16. Use Case diagram med extend

En tredje udvidelse til den basale notation er en specialisering af en Use Case, der beskriver en udvidelse eller ændret funktionalitet i forhold til den Use Case, den er en specialisering af.

En sidste udvidelse består i, at også aktører kan specialiseres.



Figur 17. Use Case diagram med specialisering

4.1.3 Beskrivelse af en Use Case

UML standarden indeholder ikke nogen anbefalinger eller krav til hvordan den enkelte Use Case beskrives. Der kan f.eks. anvendes tekst, grafik som f.eks. UMLs sekvensdiagrammer og i nogle tilfælde også tilstandsdiagrammer (UML statecharts). Anvender man en tekstbeskrivelse er der dog ved at være en slags fælles forståelse for hvad en sådan beskrivelse skal indeholde. Der vil her blive givet et forslag til en sådan beskrivelsesskabelon.

En Use Case specificerer en funktionalitet som systemet udfører for en af systemets aktører og indeholder derfor normalt et antal scenarier. Det vigtigste scenario er normalscenariet hvor alt går lige efter bogen og målet for Use Casen opnås.

En Use Case kan beskrives vha. følgende hovedpunkter:

Use Case navn:
: Der kort beskriver Use Casen vha. et handlingsorienteret udsagnsord, der virker på et navneord f.eks. ”Overvåg temperatur”, ”Konfigurer system”

Målbeskrivelse:
: En målbeskrivelse der kort og præcist beskriver det mål Use Casen opfylder for en af systemets aktører eller interessenter. Dette mål svarer til succesfuld gennemførelse af normalscenariet.

Normal scenario:
: Her beskrives det normale forløb vha. et antal trin, der fører frem til opfyldelsen af målet for Use Casen.
Hver trin nummereres og beskriver en dialog mellem aktørerne og systemet.

Undtagelser:
: I dette afsnit beskrives de undtagelser og afvigelser, der kan forekomme til normalscenariet. Det beskrives her hvordan disse skal håndteres af systemet.

Ud over disse hovedpunkter, kan der medtages flere supplerende beskrivelsespunkter f.eks. forudsætninger og slutbetingelser.

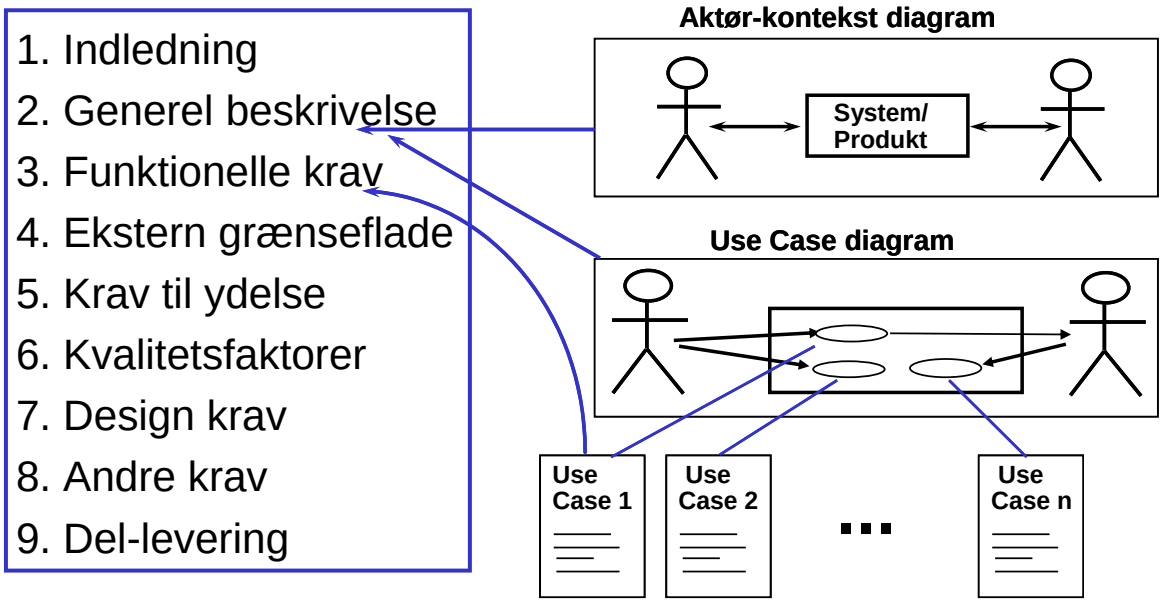
For supplerende læsning og uddybning kan der her henvises til Alistair Cockburns bog ”Writing Effective Use Cases” [Cockburn2000].

4.2 Indhold af kravspecifikation i relation til Use Cases

Anvendelse af Use Case teknikken starter med, at man fastlægger konteksten for systemet ved at definere aktørerne i forhold til det system, man er ved at udvikle. Aktørerne og systemet vises på et aktør-kontekstdiagram. Når dette er på plads, kan man begynde på at finde de Use Cases, som de enkelte aktører er involveret i.

Figur 18 viser, hvorledes aktør-kontekstdiagrammet og Use Case diagrammerne og de tilhørende specifikationer indgår i en standard indholdsfortegnelse for en kravspecifikation [SPU88]. En Use Case baseret kravspecifikation består ligesom den traditionelle kravspecifikation hovedsageligt af tekst, der beskriver de funktionelle krav (kap.3). Det der gør den store forskel, er den tek-

nik og den Use Case drevne synsvinkel, der er anvendt til at finde Use Casene i den Use Case baserede kravspecifikation.



Figur 18. Use Case i relation til et kravspecifikationsdokument

Use Case teknikken kan i øvrigt anvendes, selv om man ikke efterfølgende vil anvende objektbaserede eller objektorienterede udviklingsmetoder. Men så får man til gengæld heller ikke glæde af de metoder, der findes til at gå fra en Use Case baseret specifikation til objektorienteret analyse- og designmodeller.

I forhold til SPU håndbogen er der her indført et ekstra kapitel ”Design krav”, hvor man kan liste de ufravigelige design relaterede krav som projektet skal leve op til. Ved at liste disse eksplicit i sit eget kapitel sættes der fokus på disse ufravigelige krav. Det kan f.eks. være at systemet skal implementeres vha. en PC med Windows operativsystem.

Kapitlet ”del-levering” kan medtages, når der f.eks. er tale om en kundespecificeret opgave, hvor det er vigtigt at systemet leveres i et antal delleveringer. Ved at have dette kapitel med i en skabelon fokuseres der på at udføre iterativ udvikling af et system. Det kan dog argumenteres for at denne information er mere projektspecifik, da den vedrører selve udviklingen af systemet og derfor hører hjemme i en procesorienteret dokumentation.

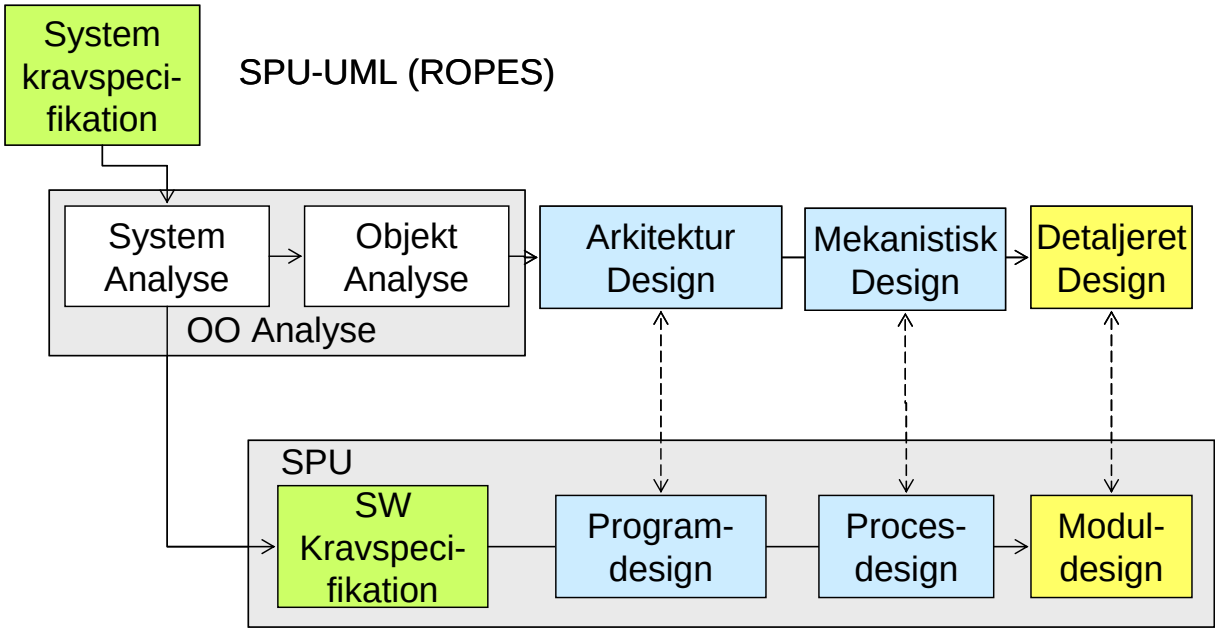
5 Vejledning i design opdateret mht. UML

Design vejledningens kapitel 2. præsenterer 6 generelle designtrin, der kan anvendes i en vilkårlig designmetode og dermed også i forbindelse med Objektorienteret design med UML. Dernæst følger 6 generelle designregler og 7 regler vedrørende arbejdsformer og dokumentation, der ligeledes er universelt anvendelige.

Design vejledningens kapitel 3-5 (s120-162) er den del af SPU håndbogen, der berøres mest af denne SPU-UML opdatering.

5.1 Sammenhæng mellem SPU og SPU-UML terminologi

SPU håndbogen omhandler primært kun softwareudviklingen i et udviklingsprojekt, hvorimod SPU-UML modellen omhandler udvikling af komplette systemer og apparater dvs. både software- og hardwareudvikling. Dette giver anledning til nogle udvidelser i SPU-UML i forhold til SPU som det fremgår af Figur 19, ligesom der også i forbindelse med anvendelsen af en objektorienteret metode til analyse, design og implementering vil medføre nogle ændringer i aktiviteter og terminologi. Sammenhængen mellem SPU aktiviteterne og de øvrige udviklingsaktiviteter i SPU-UML er vist på Figur 9. Som tidligere nævnt er aktiviteterne i SPU-UML benævnt efter de tilsvarende aktiviteter i ROPES udviklingsmodellen.



Figur 19. Sammenhæng mellem SPU-UML og SPU aktiviteter og terminologi

SPU opdeler software designarbejdet i tre designaktiviteter hhv. program-, proces- og moduldesign. Hvor programdesignaktiviteten opdeler et program i parallelle processer (task) og fastlægger deres kommunikation. Hvorefter procesdesign opdeles en proces i moduler, der i OO sammenhæng svarer til klasser/objekter. Dernæst følger moduldesign, hvor modulets funktioner og datastrukturer designes, der i UML svarer til at design en klasses operationer (algoritmer) og at-

tributter (datastrukturer). SPUs modulbegreb kan direkte oversættes til klassebegrebet i UML, hvilket gør sammenhængen enklere.

5.2 Arkitekturdesign (design af et system eller produkt)

Da SPU oprindelig er udviklet til tekniske systemer har det fra starten af været vigtigt at kunne understøtte udvikling af multiprogrammer dvs. programmer, der består af et antal parallelle processer (kaldes også for task og threads), der kan afvikles via et realtidsoperativsystem.

UML har den store fordel, at det er muligt at diagrammere processer idet disse kan udtrykkes som aktive klasser, der er klasser med eget initiativ til at kalde andre passive objekter eller kommunikere med andre aktive klasser. En sådan aktiv klasse har typisk en ”run” operation (metode), der indeholder en uendelig procesløkke og aktiveres af operativsystemet.

5.3 Mekanistisk design (design af et program eller proces)

Ved SPU-UML mekanistiske designaktivitet udføres der design af et program eller af en proces, der er identificeret i den forudgående arkitekturdesign aktivitet. Dette gøres ud fra den forudgående objektorienterede analyse og ud fra arkitekturdesignet. Det er her at samspillet mellem de klasser der indgår i processen designes. Til dette anvendes UML’s klasse-, tilstands- og sekvensdiagrammer. Det mekanistiske design kan ofte forbedres ved at anvende veldokumenterede og afprøvede designmønstre som f.eks. GoF mønstrene [GoF94].

5.4 Detaljeret design (design af en klasse)

SPUs moduldesignaktivitet består i at designe modulet, der i SPU-UML vil sige at foretage detaljeret design af en enkelt klasse, dvs. at fastlægge klassens operationer med parametre og returværdier samt at designe de tilhørende algoritmer samt at designe klassens datastrukturer, der udtrykkes som attributter i UML.

5.5 En letvægts SPU-UML designmetode

Dette afsnit er tænkt som en kort introduktion til hvorledes man kan anvende en objektorienteret letvægtsmetode til design af et sekventielt program dvs. bestående af en proces.

Kravspecifikation:

Trin 1. Udarbejd en domænemodel, der er et klassediagram, der viser de klasser der er i det pågældende domæne og deres sammenhæng. Klasserne kan f.eks. identificeres ud fra navneordene i en opgavebeskrivelse.

Trin 2. Udarbejd en Use Case baseret kravspecifikation, hvor Use Casene beskrives vha. de domænebegreber, der er identificeret på domænemodellen. Definer ud fra Use Casene et antal udviklingsiterationer.

Objektorienteret analyse, arkitekturdesign og mekanistisk design:

Trin 3. Udvælg nu en af de Use Cases, der indgår i iterationen til modellering dvs. analyse og design.

Trin 4. Udarbejd et klassediagram for den udvalgte Use Case.

4.1 Tilføj grænsefladeklasser, én for hver aktør for den valgte Use Case
4.2 Tilføj en kontrolklasse for Use Casen
4.3 Tilføj de relevante domæneklasser ud fra domæneklassediagrammet.
4.4. Forbind klasserne vha. associationer

Trin 5. Udarbejd tilstandsdiagrammer for de klasser, der har en tilstandsafhængig opførsel. Dette vil ofte være selve kontrolklassen, men kan også være grænsefladeklasser til hardwareaktører og til aktører, der repræsenterer andre systemer.

Trin 6. Udarbejd et sekvensdiagram over normalscenariet i Use Casen. Ved udarbejdelsen af dette diagram fastlægges klassens operationer.

Trin 7. Tilføj operationerne til de relevante klasser.

Trin 8. Verificer de tre typer af diagrammer op imod Use Casen.

NB! Trin 4, 5 og 6 kan udmærket udføres parallelt og iterativt, hvor man arbejder med alle tre diagramtyper samtidigt.

Fortsæt nu ved trin 3. med at vælge en ny Use Case indtil alle Use Case i den valgte iteration er analyseret og designet.

Detaljeret design:

Trin 9: De identificerede klasser detaildesignes dvs. deres operationer og attributter skal fastlægges i detaljer. Ved denne aktivitet kan UML's aktivitetsdiagrammer anvendes til at diagrammere komplicerede operationer. For simple klasser kan det detaljerede design udtrykkes vha. pseudokode eller i det anvendte programmeringssprog.

6 Referencer

[Biering2004]:
”*Praktisk it-projektledelse*”, Stephen Biering-Sørensen,
Samfundslitteratur, 2004, ISBN 87-593-1121-5
Stephens nye projektledelsesbog, der meget grundigt og omfattende uddyber og supplerer SPU håndbogens vejledning i projektstyring.

[Boehm88]:
”*A Spiral Model of Software Development and Enhancement*”, B.W. Boehm,
IEEE Computer, May 1988, pp. 61-72.
Artikel der beskriver spiralmodellen.

[Cockburn2000]:
”*Writing Effective Use Cases*”, Alistair Cockburn
En meget fin bog vedrørende beskrivelse af use cases samt niveauer for use cases. Bogen tager dog udelukkende udgangspunkt i systemudviklingseksempler fra adminstrative systemer.
Pearson Education, 2000.

[Cockburn-W]
“*Using "V-W" Staging to Clarify Spiral Development*”, Alistair Cockburn,
http://members.aol.com/acockburn/papers/vwstage.htm
Artikel der beskriver ideen i W-modellen.

[Douglass99]
”*Real-Time UML, Second Edition*
Developing Efficient Objects for Embedded Systems”, Bruce Powel Douglass
Beskriver en objektorienteret analyse- og designmetode for indlejrede systemer, der er baseret på ROPES udviklingsmodellen (en spiralmodel).
Addison-Wesley, 1999.

[Douglass99-2]
”*Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*”, Bruce Powel Douglass
Indeholder en mere dybdegående metodebeskrivelse af ROPES metoden.
Addison-Wesley, 1999.

[Fowler99]
“*UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition)*”, Martin Fowler og Kendall Scott
En god og let læselig introduktion til UML, der også indeholder lidt metode og proces.
Addison-Wesley, 1999.

[GoF94]
”Design Patterns – Elements of Reusable Object-Oriented Software”, E. Gamma, R. Helm, R. Johnson, J. Vlissides
Addison-Wesley, 1994.

[Jacobson92]
"*Object-Oriented Software Engineering*", Ivar Jacobson
Den første egentlige beskrivelse af Use Case teknikken.
Addison-Wesley 1992.

[Krutchen85]
"*Architectural Blueprints—The "4+1" ViewModel of Software Architecture"*,
Philippe Kruchten
En fin artikel, der præsenterer en model for beskrivelse af et arkitekturdesign ud fra et antal view, hvor hvert view har et bestemt formal og en bestemt målgruppe.
IEEE Software, November 1995

[OMG] Object Management Group, home page: www.omg.org
Hjemmesiden for OMG, der er organisationen bag UML standarden.

[SPU88] "*Håndbog i Struktureret Programudvikling*",
Stephen-Biering Sørensen, Finn Overgaard Hansen, Susanne Klim, Preben Thalund Madsen, Teknisk Forlag, 1988.

[UML2003]
Unified Modeling Language, se www.uml.org eller www.omg.org/uml/
Nyeste version af UML er UML ver. 2.0, der kan downloades fra ovenstående links.