Indholdsfortegnelse

1 Indledning 2
1.1 Virksomheden 2
1.2 Host 4
1.3 Service cluster 4
1.4 Applikation 5
1.5 KUDV test model 5
1.6 KUDV test teknikker 5
1.6.1 Uformelle test teknikker 6
1.7 Problem 6
1.8 Modelapparat 8
1.9 Afgrænsning 8
1.10 målgruppe 9
2 Tilstandsdiagrammer 9
2.1 Udarbejdelse 9
2.2 usecase 9
3 Modellering i UPPAAL 12
3.1 Fra tilstandsdiagram til modellering i UPPAAL 12
3.2 Simulering 13
3.3 Tjek properties 14
4 Unit test via junit 15
4.1 Fra UPPAAL trace til JUnit kode 16
4.2 Indlægning af testsuites 18
5 Opsummering 19
5.1 UPPAAL 20
5.2 JUnit 21
5.3 Test-metoden 21
6 Konklusion og perspektivering 21
6.1 konklusion 21
6.2 perspektivering 22
litteraturliste 23
appendix a 24
Appendix b 25
appendix C 27

1 INDLEDNING

Dette projekt har til opgave at afdække, hvilke muligheder der findes for forbedringer i udviklingsprojekternes unit-test af moduler. Ved at følge kurset "Test og verificering af software" har det givet forfatteren nogle ideer til mulige forbedringer i Nykredits IT afdeling Koncernudvikling [1] projekt- og udviklingsmodel og specifik den del af testen der omhandler unit-test.

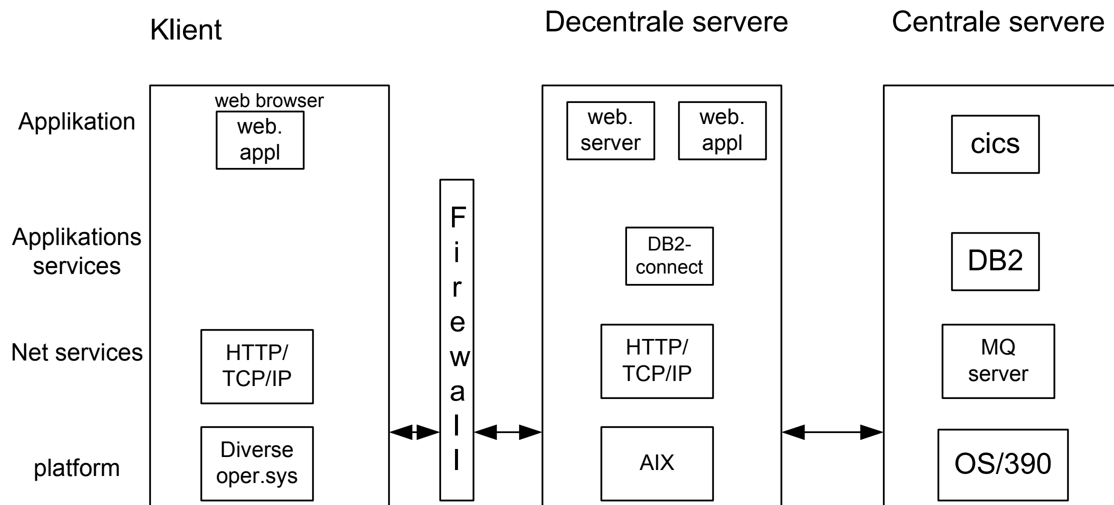
Nykredit startede for fire år siden en proces, der skulle sikre, at it-systemerne skulle understøtte koncernen i at nå de forretningsmæssige mål. Et af resultaterne i denne proces har blandt andet udmøntet sig i en ny udviklingsmetode. En vigtig ingrediens i Nykredits nye projektopskrift er usecases, der bruges til at beskrive funktionaliteten for et kommende system. Forretningsfolk inddrages i høj grad for at sikre validiteten og kravene til det kommende system. Brug af usecases (notation:UML) har også givet it-folk og forretning et bindeled/fælles sprog som et vigtig i kommunikationen og forståelsen af systemet. Den oprindelige model/metodik var Component Based Development (CBD) og det gav ofte problemer for forretning, at vurdere om de komponenter der kom ud af det var det ønskede. Det var så at sige udvikling på it's premisser. Nu bruges usecase også som udgangspunkt til at definere testcases i den funktionelle test.

1.1 VIRKSOMHEDEN

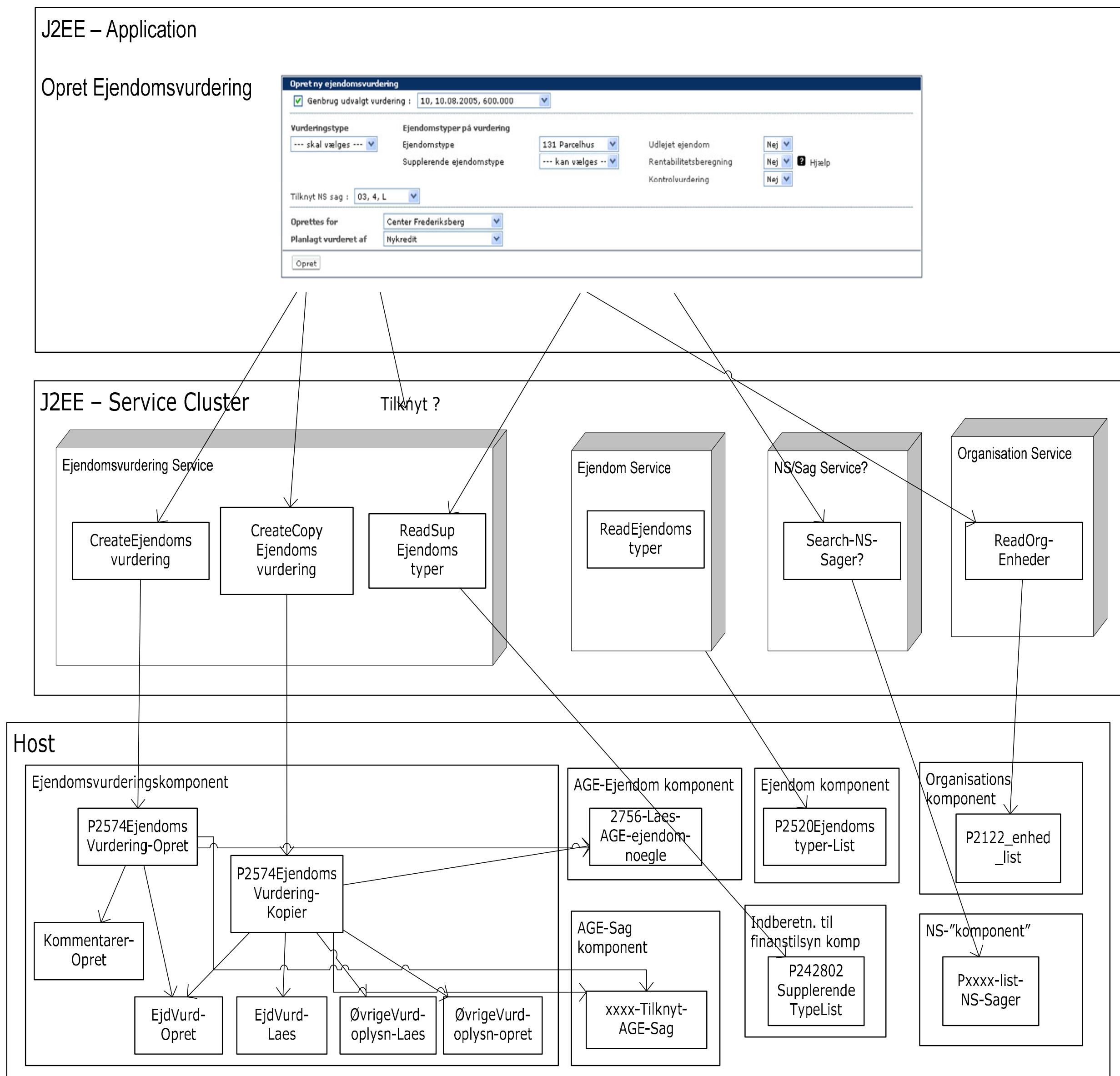
Nykredit koncernudvikling (KUDV) har siden 1980 udviklet systemer til Realkreditvirksomheden Nykredit. I starten af 1990'erne blev de 3270 baserede systemer suppleret med Client/server systemer. Nykredit valgte i 1998 at satse på selvudviklingsstrategien, hvor forretningsområderne Bank og Forsikring på den baggrund kom til. Med teknologien og behovet startede udvikling af Web-baserede systemer midt i 1990'erne, så som Bank og Portal løsninger til enkelte forretningsområder. Gennem alle årene er data gemt centralt i DB2 databaser og udvikling af tynde interfaces er stadig et metodekrav. Endvidere er der krav om at forretningslogik placeres i host moduler. Udviklingsmetoden er løbende blevet ajourført til det aktuelle behov og hvad der på længere sigt er mest hensigtsmæssig for at koncernudvikling kan servicere dets kunder. De små projekter/forvaltning har mulighed for, at benytte den del af det samlede metodeapparat, som de har behov for. Metodeafdelingen har ansvaret for implementering af metoder og servicering heraf. Udvikling af systemer er primært foretaget via CBD, hvor entiteter og relationer er nøgleord. Forretningsanalysen foretages med stor inddragelse af kunden og via metoden UCCBD (UseCase CBD) som der pt. benyttes stor energi på at indføre i alle koncernens udviklingsprojekter.

Figur nedenfor viser overordnet Nykredits driftsmodel fra Client-laget til modellaget til de centrale Host servere.

[1] Forfatteren er ansat som software udvikler i Nykredit Koncernudvikling, Afdeling Aalborg



Følgende figur viser den logiske lagdeling til usecase: Opret ejendomsvurdering.



I det følgende gives der en kort gennemgang af hvad de enkelte lag indeholder.

1.2 HOST

Her forefindes moduler (Server, Toab, Foab Aoab) der understøtter CRUDL operationer, dvs Create, Read, Update, Delete og List operationer. Transaktionsmodul (Toab) der udstiller operationer (f.eks opret entitet), forretningslogik er implementeret i Foab modulet og dataacces finder sted i Aoab modul.

Der kodes en skal omkring Toab modulet og dette modul kaldes Server ledet. Der er mulighed for at kode yderligere logik i Server modulet, f.eks. fejl-kode håndtering men normalt inderholder det kun kald af Toab modulet. Server moduler oversættes til proxy moduler, som kan bruges i det overliggende service cluster lag. Proxy modulet er et Java modul med metoder og kan således implementeres i Cluster laget.

1.3 SERVICE CLUSTER

Oprindelig er det tænkt at der ikke skal være forretningslogik implementeret her, men denne regel kan fraviges hvis det i det enkelte projekt findes hensigtsmæssig.

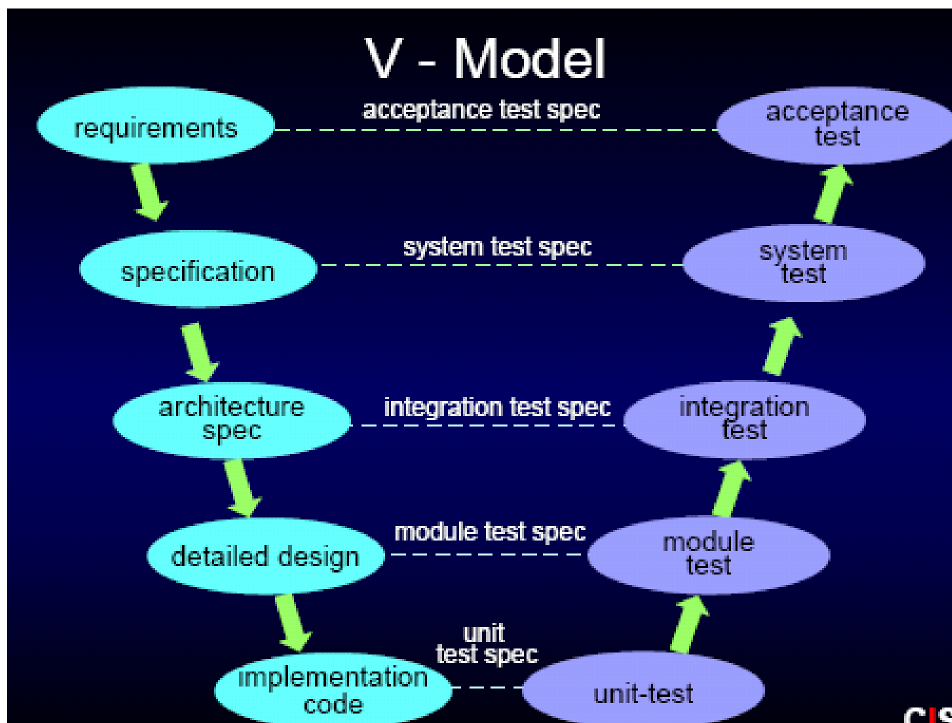
For at dette lag kan benytte operationer fra Host laget genereres proxy. F.eks. server P2574Ejendoms Vurdering-Opret vil kunne tilgås fra dette lag.

1.4 APPLIKATION

Her kodes elementerne til styring af Brugergrænsefladen. Applikationen kodes i Java og strunts.

1.5 KUDV TEST MODEL

For at få et overblik over hvordan KUDV planlægger og gennemfører test, kan det eksemplificeres via figur 1 nedenfor, hvor V-modellen **[2]** er angivet. V-modellen er medtaget her fordi den formelt beskriver den måde som KUDV planlægger test ud fra. Kunden og KUDV udarbejder kravspecifikation (requirements) og herefter foretager KUDV specificering som udmøntes i IT-arkitektur. Denne overordnet arkitektur detaljeres i et design dokument som IT-udvikler benytter til at specificere koden som skal afvikles på et antal servere. Denne fremgangsmåde følger den inkrementale vandfaldsmodel fra det klassiske udviklingsparadigme **[2]**. I KUDV er vandfaldsmodellen kombineret med et iterativt og parallelt element således at eksempelvis design aktiviteten kan starte før den endelige arkitektur er på plads og godkendt. V-modellen beskriver at koden testes som en del af unit-testen endelig testes brugernes krav formelt i accept-tesen.



Figur 1.1: V-model

1.6 KUDV TEST TEKNIKKER

I det følgende beskrives overordnet retningslinierne for hvordan KUDV tester applikationer.

Planlægning af test påbegyndes i afklaringsfaserne. Således udarbejdes allerede i Projektafklaringsfasen forslag til teststrategi, ligesom testaktiviteterne er indarbejdet i grovplanen og ressourceestimaterne for det samlede projekt.

I Kravspecifikationen beskrives kravene, så de er testbare, hvilket sammen med fastlæggelse af accepttestkriterierne er med til at sikre en høj kvalitet af såvel kravspecifikation og i de efterfølgende testaktiviteter i testfaserne: Modultest, Integrationstest, Systemtest og Accepttest.

Ansvaret for den samlede testproces, d.v.s. planlægning og afvikling af testaktiviteterne, herunder overblik over berørte systemer, og relevante testværktøjer, som skal indgå i den samlede test, er ansvarsplaceret hos projektledelsen.

Kunden/faglig ansvarlig har i testaktiviteterne det sædvanlige forretningsansvar for den faglige kvalitet. Forretningsansvaret ved test omfatter specielt:

- Udarbejdelse af relevante testcases med tilhørende 'sandhedsværdier' (specifikation af den korrekte funktionsmåde eller beregningsresultat).

  Test af komplicerede finansielle beregninger kræver stor forretningsviden og overblik, hvorfor de forretningsansvarlige tidligt bør afdække beregningerne på formel niveau, krav til testcases for at dække udfaldsrummet og eventuel anvendelse af testdrivere ('testkanoner')
- Fastlæggelse af accepttestkriterierne f.s.v.a. forretningsfunktionaliteten
- Sikre den relevante bemanding og beslutningskraft til testaktiviteterne.

1.6.1 UFORMELLE TEST TEKNIKKER

Den enkelte udvikler benytter et sæt test teknikker som ikke er nævnt i KUDV metode håndbog. Disse teknikker benyttes afhængig af opgaven og erfaring hos udvikleren. Kodeinspektion/skrivebordstest består i gennemgang af koden af den enkelte udvikler eller kollega som har kendskab til forretningsområdet eller blot er kendt for, at give god feedback ved brug af denne teknik.

Der laves SQL opslag i databasen før og efter afvikling af test og disse udskrifter giver udvikleren et bud på om testen er godkendt.

Der foretages midlertidig kodetilretning, hvor display viser hvilke statement programmet er i færd med at afvikle og indhold af variabel. Endvidere foretages der gennemløb med debugger.

Fælles for de ovennævnte teknikker kan siges, at de ikke indeholder en overordnet systematik og mulighed for, at følge op på hvor godt testen er forløbet. Den enkelte udvikler kan med god ret sige at applikationen er testet, men retningslinierne for hvad der skal foretages før, under og efter testforløbet mangler. Dermed sendes ansvaret og fejlene videre i udviklingsprocessen til gene for andre udviklere og forretningstestere i de efterfølgende test-faser.

Litteraturen **[2]** giver et bredt spektrum af white/blackbox teknikker som med fordel kan benyttes og der er også teknikker som giver et bud på programmets kompleksitet og dermed hvor test indsatsen med fordel kan sættes ind.

1.7 PROBLEM

Kan det ovennævnte KUDV test setup blive bedre ? Problemet består i at metodens test beskrivelser er for generelle og de anvendte test teknikker er for simple. Er der nogle muligheder for på et tidligere tidspunkt at verificere at modellen er korrekt og i overensstemmelse med kundes/brugernes krav.

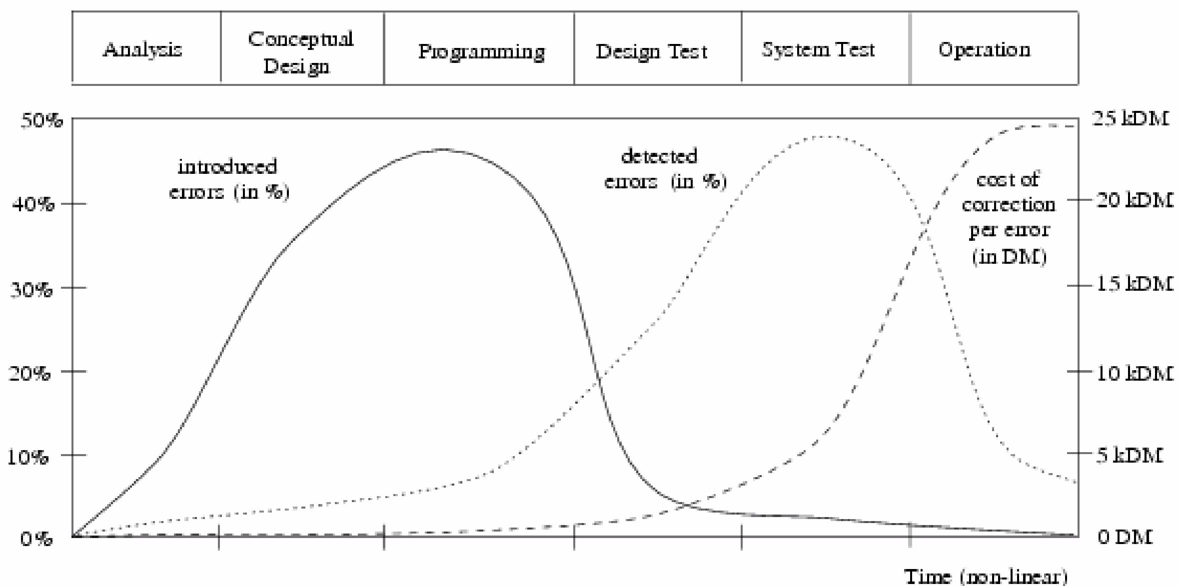
Når kunden i Nykredit er i færd med at specificere krav til det nye system, er der mulighed for, at visualisere systemet via en prototype. Denne prototype indeholder ingen dataacces men værdier og præsentation er hardkodet. Udvikling af Prototypen giver også et overhead, idet ikke alt kan genbruges i det kommende system.

Når KUDV udvikler web-applikationer til brug for Nykredits samarbejdende pengeinstitutter skal denne applikation strategisk basere sig på de eksisterende legacy (host) moduler. Disse moduler indkaples og gøre tilgængelig som services fra Java-platformen. Denne udvikling af services forventes at stille større krav til selvstændige og isoleret test af de pågældende services, end de krav der i dag stilles til test af host moduler. Grundlæggende vil der fremover være flere abonnenter af host modulerne, så kravet til stabilitet øges. Der vil være 3 situationer hvor host moduler skal ændres pga. fejl eller ønske til ny forretningsfunktion: 1) hensyn til legacy 2) hensyn til web applikation 3) både legacy og web skal påvirkes af ændring. Disse 3 situationer skal kunne igangsættes og kvalitetskriteriet er igangsætning uden følgefejl. Et eksempel på følgefejl er rettelse aht. ny web funktionalitet, men host applikation fejler pga. manglende test af denne abonnent.

Behov for automatisk testafvikling: systemerne bliver mere integreret og udvikler kan ikke have viden på alle platforme. Refakturering bliver ikke udnyttet og det bevirker at flere moduler dækker samme funktionalitet. I takt med at flere systemer udvikles i Java giver det et behov for flere java kompetencer generelt i KUDV. Fokus på "Time to market", dvs. udvikling af forretningssystemer med fokus på kundeservicering af produkter giver pres mod minimering af udvikling og testtiden. Dette stiller bl.a. krav til maskinelle testteknikker som kan sikre rapportering om systemets tilstand under udviklingsaktiviteten.

Større indsats mht. udarbejdelse af specifikationsbeskrivelser bør udnyttes i test fasen. Ifølge **[2]** er det vigtigt at finde og rette fejl tidlig i udviklingsforløbet. Fejl er i denne sammenhæng også specifikationsfejl som optræder i kravspecifikation og endnu ikke er designet og kodespecificeret.

Hvorfor er det vigtigt at teste og hvornår opstår fejlene i udviklingsforløbet ? Figur 1.2 **[4]** viser at fejl opstår i de tidlige faser i udviklingsforløbet dvs. under analyse og design og fejlene i høj grad findes under system testen. Endelig viser figuren, at omkostninger til fejlretning stiger ekponentiel ved efter implementering og derfor skal testen søge at finde så mange fejl som muligt, på et tidligt tidspunkt.



Figur 1.2: Fejl introduktion, fejl fangst og fejlløsningsomkostninger

Vedligeholdelse af systemer mangler værktøjer til understøttelse af fejl-retning og test (her ser vi JUit som et vigtigt redskab).

Der bruges megen energi til udarbejdelse af usecases og dokumenter relateret til kravspecifikationen og det er min holdning, at review af bruger og KUDV kun i nogen grad fanger fejl og uklarheder. Der bør derfor på dette tidspunkt af udviklingsprocessen bruges ressourcer på verificering, således at bruger og KUDV er enige om tolkning af materialet og dermed overleveres der færre fejl til testfasen. Verificering er således et område der bør undersøges nærmere mht. brugbarhed i Nykredit udviklingsproces. Et andet og vigtigt element i test-fasen er automatik. I KUDV stilles der krav om automatiserede processer, for derved at undgå menneskelige fejl, men også for at opnå en større effektivitet i test afviklingen.

Disse vurderinger leder frem til rapportens primære undersøgelsesområde og hypotese:
-vil det give øget kvalitet til software produktet, at bruge UPPAAL værktøjet i KUDV til, at verificere, simulere forretningsmodellen og
-vil det give en øget kvalitet i software produktet, hvis der foretages en "automatisk" overførsel af UPPAAL traces til brug for afvikling i JUnit?

Med "automatisk" tænkes der på automatik i så høj grad som muligt. I KUDV viser erfaringer at udvikler efterspørger værktøjer som maskinelt foretager opgaver, som af den enkelte udvikler anses som vigtigt, men dog trivielt.

Hypotese: Specifikationer er input til Modelverificering (her bruges UPPAAL værktøjet), viden ifm af tracelister og betingelser, dvs. kan denne tilstand nås, hvad er den korteste sti ? etc. Endelig bruges tracelister fra UPPAAL simuleringsdel som input til JUnit for at afvikle en (tilfældig) sekvens af metoder. Det at oprette testcases i Junit og afvikle test suiter er i dag en manuel KUDV aktivitet og der er således behov for (maskinel) konvertering mellem UPPAAL og til JUnit.

Hvad opnår vi: Større og tidlig fokus på forretningsprocesserne, hvor kunden og KUDV i samarbejde kan verificere at modellen er korrekt før de efterfølgende fase, med udgangspunkt i specifikationerne, arkitektur, design og kodning tager over og frembringer det endelige system. En højere kvalitet i specifikationerne bør bevirke, at antallet af fejl i test-fasen og ændringer til specifikationen bringes ned. Ved brug af JUnit foretages der validering og systemet tjekkes regelmæssigt at test-cases er ok og hurtig varsling og korrektion kan foretages, hvis der findes fejl i test-cases.

1.8 MODELAPPARAT

Input til UPPAAL er en model af specifikationsmodellen, giver det nogle begrænsninger der bør oplyses her ? I afsnit 5.1 gives der et kort rids af fordele og ulemper ved brug af UPPAAL, men i denne sammenhæng er valget af UPPAAL udelukkende foretaget ud fra interesse og de muligheder som kurset har præsenteret.

I afsnit 5.3 gives der et upsummering over den procedure som denne rapport har frembragt.

Hvad er alternativerne til UPPAAL? VisualState eller UML-Statecharts, dette er ikke analyseret i nærværende rapport.

1.9 AFGRÆNSNING

Vi ser på test af de servere der oversættes til Java proxy'er og tester således ikke de andre lag, dvs. brugergrænsefladen og service-laget. Test af disse lag er også vigtigt men i denne sammenhæng uden for rapportens analyseområde.

Der udvælges usecase fra Ejendomsvurderingsprojektet og som dækker et sæt CRUD operationer.

Der gøres brug af UPPAAL **[3]** men der foretages ikke i nærværende en dybere forklaring af teknikken og tankerne bag dette værktøj, blot nævnes de elementer som (speciel KUDV'er) har brug for at forstå sammenhængen.

En mulig konvertering af output fra UPPAAL til JUnit skal ikke kodes her, men det skal afklares og beskrives hvordan det kan foretages.

Dette projekt omhandler ikke specifikke Java programmeringsteknikker, men der vises udvalgte Junit kode til forklaring af sammenhængen med JUnit og Java koden.

1.10 MÅLGRUPPE

Målgruppen er vejleder på TOV-projektet og Nykredit (KUDV) udviklere og metode specialister der har ansvaret for test beskrivelserne. Der kan forekomme formuleringer som er medtaget aht. KUDV kompetancer.

2 TILSTANDSDIAGRAMMER

Dette afsnit omhandler hvordan de udvalgte forretningsprocesser kan modelleres til tilstandsdiagrammer.

2.1 UDARBEJDELSE

Usecase udarbejdes i smarbejde mellem Kunden og KUDV og det primære formål med usecase er at kunne afklare pre og post konditioner til den enkelte usecase. Ifølge KUDV-metoden skal der forefindes use cases som viser skiftene mellem de definerede tilstande for begreberne i begrebsmodellen.
Begrebsmodel udarbejdes af kunden og består af en opremsning af de primære forretningsentiteter.

Udviklingsmodellen (UCCBD) foreskriver at der udarbejdes tilstandsdiagrammer såfremt der forefindes begreber med en kompleks livscyklus. Denne vage formulering er måske skyld i at der udarbejdes tilstandsdiagrammer i ca. 20% af projekterne.

2.2 USECASE

Figur 2.1 nedenfor viser usecase "Opret Faktuel Beskrivelse". Denne og de øvrige usecases udarbejdes af brugerne i samarbejde med KUDV.

<table>
<tr><td>ID (nr.) :</td><td>140</td><td>Titel:</td><td colspan="7">Opret Faktuel Beskrivelse</td></tr>
<tr><td>Projektnr.:</td><td>1320</td><td>Projektnavn:</td><td colspan="5">Ejendomsvurdering</td><td>Init:</td><td>KWJ</td></tr>
<tr><td>Kategori:</td><td>Komponent</td><td>Status:</td><td>Skitse</td><td>Prioritet:</td><td>Normal</td><td>Prioritetnr.:</td><td>1</td><td>Ansvarlig:</td><td>LAS</td></tr>
<tr><td colspan="2">Formål og beskrivelse:</td><td colspan="8">At kunne oprette faktuel beskrivelse til en ejendomsvurdering.

Definition: En faktuel beskrivelse indeholder de generelle oplysninger til faktuel beskrivelse (hustype, vandfosyning, kloakafløb og offentlige kommentarer) samt opdateringsstemplet.</td></tr>
<tr><td colspan="2">Aktører:</td><td colspan="8">Anvender Applikationen</td></tr>
<tr><td colspan="2">Starthændelse:</td><td colspan="8">Det ønskes at oprette oplysningerne til faktuelle beskrivelse på en ejendomsvurdering.</td></tr>
<tr><td colspan="2">Startbetingelse:</td><td colspan="8">Ejendomsvurdering er oprettet på databasen.</td></tr>
<tr><td colspan="2">Slutbetingelse:</td><td colspan="8">Faktuel beskrivelse på ejendomsvurderingen er oprettet på databasen</td></tr>
<tr><td colspan="2">Input:</td><td colspan="8">evd-vurderings-id, Bruger, felterne til faktuelbeskrivelse: (hustype, vandfosyning, kloakafløb og offentlige kommentarer) samt beskrivelses-type, der f.eks. angiver om oplysninger er fra offentlige ejendomsdata.</td></tr>
<tr><td colspan="2">Output:</td><td colspan="8">evd-vurderings-id + nyt opdateringsstempel med organisatoriske oplysninger</td></tr>
</table>

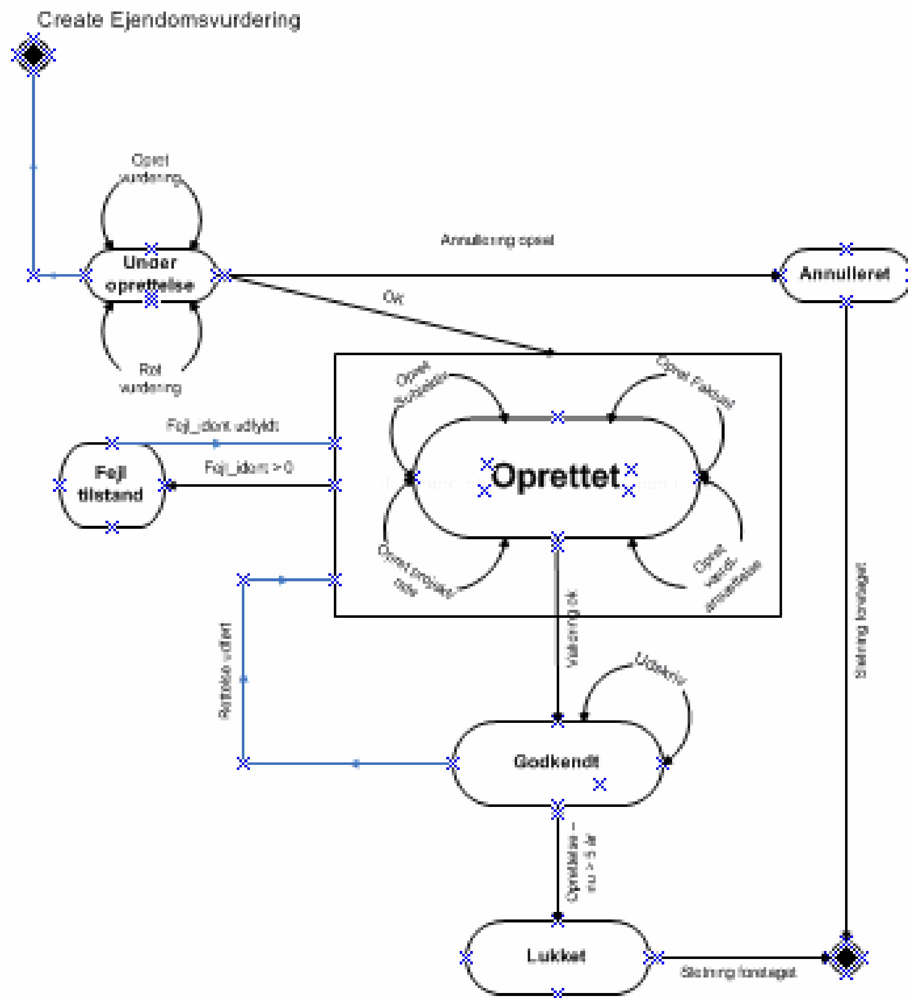
Normalforløb:	1. Valider bruger-id og find navn + evt. org_enhed. (regler beskrives af QJM) 2. Kald ejendomsvurdering læs. Kontroller status er OK til at rette (ikke lukket eller annulleret). 3. Valider enkeltfelter er udfyldt korrekt jvnf formater/regler i billedbeskrivelsen og indhold i typetabeller. Bemærk at hustype kan oprettes med 'ingen'. Hustypen skal valideres op imod ejendomstypen på ejendomsvurderingen. 4. Kald datakomponenten 'faktuel-opret'.	
Normalforløb	Aktør	System
	1. 2.	1. 2.
Undtagelser:		
Fejlbeskrivelser:	Brugerid ej udfyldt Bruger findes ikke vurderingsnr findes ikke formatfejl på feltniveau	
Testcase ref:		
Udestående:	Beskrivelsestype skal vi have den ?	
Bemærkning:	- Hustype skal være udfyldt, når man retter via skærmbilledet, men hvis man sætter hustype til blank fordi, man f.eks. kopierer offentlige ejendomsdata til faktuel, så skal vi kunne gemme forekomsten med ingen.	
Diagrammer:		
Interessenter:		
Dokument ref.:		

Revisionslog:			
Dato	Version	Beskrivelse	Init
23/05/2005	1.0	Første udgave – PA	MSV
07/11 2005	1.1	Tilrettet i PI	KWJ

Figur 2.1: Usecase Opret Faktuel

Når alle usecases er udarbejdet foretages der livscyklus beskrivelse af udvalgte begreber (Læs: klasser) og disse tegnes som tilstandsdiagrammer for bl.a. at få klarlagt forudsætningerne for slettekriterier.

Det følgende tilstandsdiagram (Appendix C viser tegning i fuld størrelse) dækker således over flere usecases og skulle gerne give information til både brugeren og udvikleren om de betingelser det kommende system er underlagt.



Figur 2.2: Tilstandsdiagram

Tilstand Fejl tilstand er med i figur 2.2 og skal blot illustrere at fejl fra server kald skal håndteres, men det er ikke tænkt at modellen skal afspejle denne tilstand. I KUDV har vi erfaring for at denne tegne metodik giver brugeren en store forståelse for sammenhængene.

I det følgende opremses udvalgte datastrømme som er kravene til at en Ejendomsvurdering kan blive oprettet. Dataanalysen er ikke foretaget ifm. med denne rapport og problemstillinger og uklarheder diskuteres ikke her.

Opret Vurdering (input: Instance_id, Ejendom_type, Opdat_timestamp, Output: fejlkode, tilstand)
Ret Vurdering (input: Instance_id, Ejendom_type, Opdat_timestamp, Output: fejlkode, tilstand)
Opret Subjektiv (input: Instance_id, Subjekt_type, Subjekt_kategori, årstal, Opdat_timestamp, Output: fejlkode, tilstand)
Opret Faktuel (input: Instance_id, Hus_type , Byg_anvend_type, Kategori_type, Opdat_timestamp, Output fejlkode, tilstand)
Opret Værdiansættelse (input: Instance_id, Vaerdi_type, V_kategori, Opdat_timestamp, Output: fejlkode, tilstand)
Opret Projektrate (input: Instance_id, Rate_beloeb, Rate_dato, Output: fejlkode, tilstand)

3 MODELLERING I UPPAAL

UPPAAL er tool-box der indeholder modellering og simulering og verifikation af real-tids systemer **[3]**. Det er velegnet til systemer der kan modelleres som en samling af non-deterministiske processer med endelige kontrolstrukturer. Det er designet med fokus på brugervenlighed og effektivitet. For at tilgodese modellering og debugging indeholder UPPAAL model tjekker muligheden for at genere diagnostisk trace der kan forklare hvorfor en property er opfyldt (eller ej) iht. system beskrivelsen. Modellen oprettes i formatet tids automata og efterprøves vha. simulator. Her kan der visuelt foretages debugging og fejl kan fanges her. Endelig efterprøves opnåelighed i model-tjekker.

Nedenfor listes de globale erklæringer som der er fundet behov for i model Ejendomsvurdering.

Clock t er oprindelig tænkt til styring af slette-regel, dvs. Ejendomsvurdering ændre status fra Godkendt til Lukket når tidsenheden 5 år fra oprettelse er opfyldt. I Uppaal modellen i figur 3.1 er dette styret af Brugeren som via Send_til_slet ændre tilstanden.

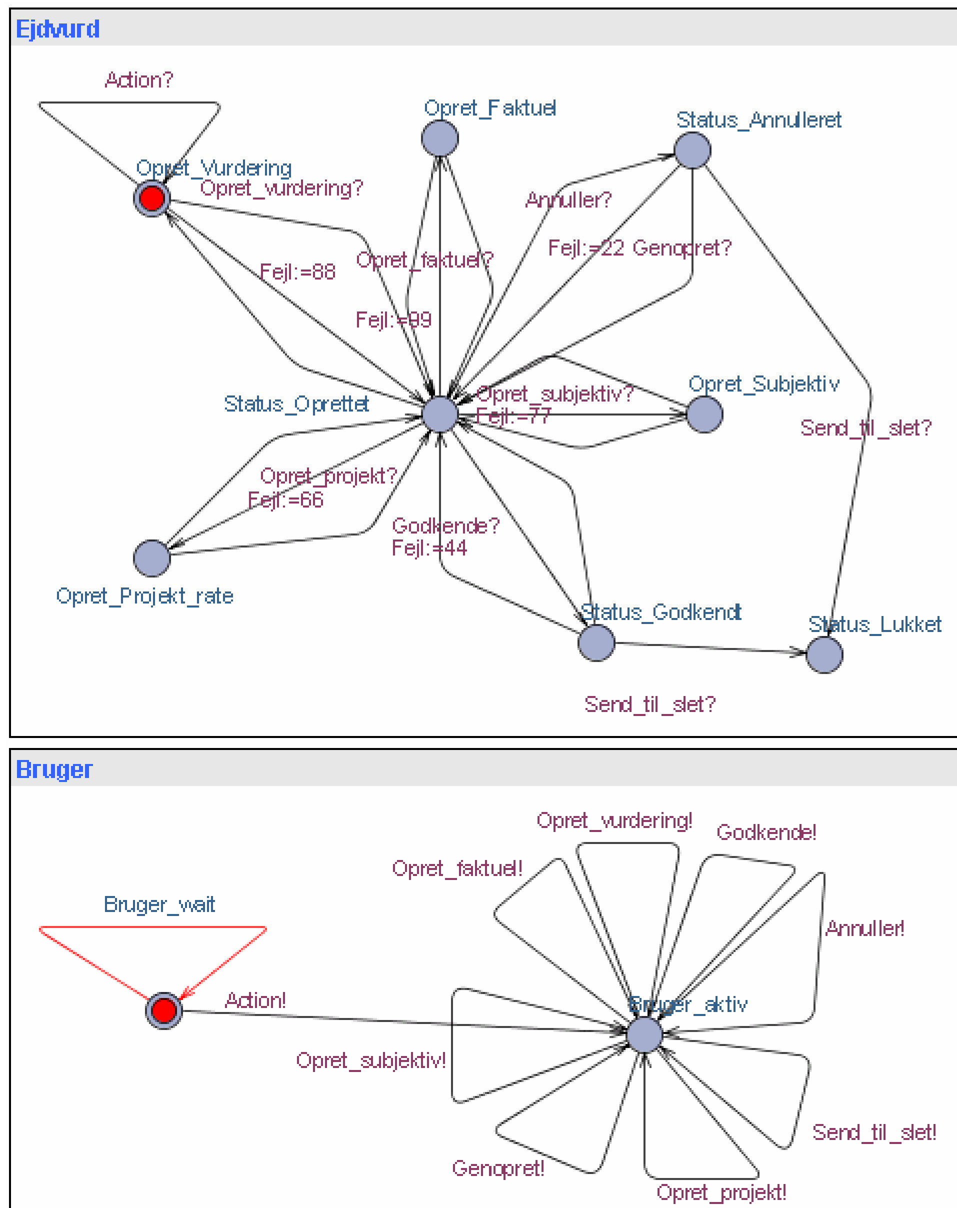
I modellen er opsætning af Fejl styret således at hver har fået en entydig fejl-kode. F.eks. leveres Opret_faktel Fejl 99 retur og Status_Annulleret kan give Fejl 22 retur. Disse fejl (koder) er kun opsat af og angivet i modellen af logiske årsager. I den virkelige verden kan alle KUDV servere levere de samme fejl retur og kun i enkelte tilfælde er der behov for server specifikke fejl koder.

```
clock t;
chan Action, Opret_faktuel, Opret_vurdering, Opret_subjektiv, Opret_projekt,
Annuller, Genopret, Send_til_slet, Godkende;

// Hvis Fejl > 0 så skal den håndteres.
int Fejl;
```

3.1 FRA TILSTANDSDIAGRAM TIL MODELLERING I UPPAAL

Figur 3.1 nedenfor viser hvordan Bruger og Ejendomsvurdering interagere med hinanden, dvs. Brugeren påvirker ejendomsvurderingssystemet og dette skifter tilstand ifølge reglerne.
Det ses af figuren, at Bruger fra tilstand Status_Annulleret og Status_Godkendt kan ændre tilstanden til Status_Lukket.

Figuren søger at illustrere at Brugeren er i vente position (Bruger_wait) og ved handling Action skifter modellen status til en af de mulige, f.eks. Opret_projekt eller Opret_vurdering. Skal medellen være virkelighedstro bør modellen afspejle af Brugeren på et tidspunkt vender tilbage til vente position. Dette er ikke implementeret i modellen.



Figur 3.1 Model af Ejendomsvurdering

3.2 SIMULERING

Her er et eksempel på random trace produceret fra simuleringsbilledet.

```
(Opret_Vurdering, Bruger_wait)
(Bruger.10.Action!, Ejdvurd.21.Action?)
(Opret_Vurdering, Bruger_aktiv)
(Bruger.1.Opret_vurdering!, Ejdvurd.4.Opret_vurdering?)
(Status_Oprettet, Bruger_aktiv)
(Bruger.2.Annuller!, Ejdvurd.16.Annuller?)
(Status_Annulleret, Bruger_aktiv)
(Ejdvurd.20)
(Status_Oprettet, Bruger_aktiv)
(Bruger.3.Opret_projekt!, Ejdvurd.1.Opret_projekt?)
(Opret_Projekt_rate, Bruger_aktiv)
(Ejdvurd.11)
```

```
(Status_Oprettet, Bruger_aktiv)
(Bruger.2.Annuller!, Ejdvurd.16.Annuller?)
(Status_Annulleret, Bruger_aktiv)
(Bruger.6.Genopret!, Ejdvurd.13.Genopret?)
(Status_Oprettet, Bruger_aktiv)
(Bruger.8.Godkende!, Ejdvurd.8.Godkende?)
(Status_Godkendt, Bruger_aktiv)
(Bruger.7.Send_til_slet!, Ejdvurd.2.Send_til_slet?)
(Status_Lukket, Bruger_aktiv)
```

Denne sekvens har formen Linie 1:Tilstand, linie 2:Action, linie 3:Tilstand, linie 4 Action osv.
I afsnit 4.1 bruges denne sekvens som input.

3.3 TJEK PROPERTIES

I dette afsnit beskrives et sæt mulige model tjeks til Ejendomsvurdering og hvad de kan bruges til.

Safety A[] deadlock bruges til at undersøge om deadlock tilstand nåes.

Syntax for Reachability skal læses således: E<> er tilstand P.X true

Eksempler fra modellen:

```
E<>Ejdvurd.Opret_Faktuel
E<>Bruger.Bruger_aktiv and Ejdvurd.Status_Godkendt
E<>Bruger.Bruger_aktiv and Ejdvurd.Opret_Faktuel
E<>Bruger.Bruger_aktiv and Ejdvurd.Status_Annulleret
A[] not deadlock
```

Modellen er udarbejdet således at deadlock opstår med status: Status_Lukket og derfor giver alle de ovennævnte queryes ok til opnåelighed, bortset fra A[] not deadlock.

De nævnte verifikationseksempler kan med god grund siges at være åbenbare, dvs. ved at betragte modellen kan dette hurtig ses ved selvsyn. Dette kan generaliseres til: lille overskuelig model-> giver lille fordel og stor model -> giver stor fordel ved brug af model tjekker. I KUDV sammenhæng vil det betyde, at ca. 20 % af udviklingsprojekterne med fordel kan benytte verifikation.

4 UNIT TEST VIA JUNIT

JUnit **[1]** er udviklet med baggrund (filosofi) i tesen om test først og kode bagefter. Test udarbejdes til hver eneste enhed og afvikling af den enkelte test-case foretages løbende til sikring af fejl-fri program. Den enkelte test-case samles i test-suites som giver mulighed for effektiv afvikling.

Formålet med unit test er at validere de mindste elementers korrekthed i programmet. I JUnit oprettes der typisk en testklasse for hver java klasse, der skal testes. Denne TestCase indeholder metoder, som alle egenudviklede testklasser arver fra. Metoden `setUp()`, initiere inputtet og kaldes før hver testafvikling. Metoden `TearDown()` fjerner testsetuppet og kaldes ved afslutning af hver testafvikling. Ved brug af nøgle ordet `assert` verificeres testen med et boolsk udtryk op i mod testmetode der navngives test<MethodName>. Hvis det boolske udtryk er falsk fejler testen.

Figur nedenfor viser kode der kalder proxy til læs af initialer via server kald. Hvis metoden returnere fejl-ident forskellig fra nul, fejler testen og rapportering foretages automatisk i Junit til brug for opfølgning.
Ud over test på retur svaret er der lagt test ind på status af objektet. Denne status angiver hvor Ejendomsvurdering er i sin livscyklus. Hvis status ikke er som forventet skal forklaring søges
Dette metodekald svare til kald mod FSM (Finite State Machine) **[6]** hvor Ejendomsvurdering betragtes som en blackbox med tilstande. Med et given input til FSM gives et output og Ejendomsvurdering skifter tilstand.

I det nævnte eksempel nedenfor er det en læse operation (PersonidLaesConnection) og Ejendomsvurdering skifter ikke tilstand i denne situation, men ved operationerne Create, Update operationer vil Ejendomsvurdering skifte tilstand, fra ”Under oprettelse” til ”Oprettet” osv.

```
package dk.nykredit.business.organisation.connector.connection;

import dk.nykredit.business.connector.FinderException;
import dk.nykredit.nif.component.monitor.Log;
import dk.nykredit.service.x2005.x11.x24.InitialsType;
import dk.nykredit.service.x2005.x11.x24.OrganisationNoegleIdType;
import junit.framework.TestCase;

/**
 * Place description here.
 *
 * @author Bo Stenvang,BSVN@nykredit.dk)
 */
public class PersonIdLaesConnectionTest extends TestCase {
    private PersonIdLaesConnection con;

    protected void setUp() throws Exception {
        super.setUp();
        con = new PersonIdLaesConnection();
    }

    public void testExistingInitials() throws Exception {
        long id = con.getPersonId("BSVN");
        assertTrue(id > 0);
    }

    public void testStatusInitials() throws Exception {
        long st = con.getStatusId("BSVN");
        assertTrue("Status ikke u.o",st=="Under oprettelse");
    }

}
```

Figur 4.1: eksempel på kald af proxy PersonIdLaesconnection

4.1 FRA UPPAAL TRACE TIL JUNIT KODE

I det følgende beskrives en procedure der beskriver hvordan trace genereret i UPPAAL kan bruges som input til java kode der kan afvikles i JUnit og dermed give status på server kald i Ejendomsvurderingssystemet. Situationen er den, at udvikler har oprettet specifikation I UPPAAL og godkendt den ved at have set simulering og traceliste produceret herfra.

Det er her valgt at bruge template der indeholder de generaliserede rutiner (kode) og reference til håndtering af variabelt input. Det vil være en fordel at lave templates så generelle, at de kan bruges til andre systemer såsom Tinglysning og Omlægningsberegning. Dette vil minimere arbejdet når nye projekter ønsker at benytte denne procedure.

Den manglende generalisering i templates ses i hardkodning af tilstand, handlingsnavne fra modellen som bruges direkte i template. Dette er vigtigt at vide, at der er denne sammenhæng når nye modeller oprettes.

Template oprettes til brug for Create **Vurdering** (SetUp_Vurdering):

```
Protected void setUp()
//Vurderingsnummer,type,Status,ejendomstype,sdd,dsd,oprettelses-dato,timestamp
//
Vurdering vurdering = new Vurdering(123456,"Navn", " ",131,0.34,12.05.2006,11:12:34);
```

Template (Cr_Vurdering) indeholder denne kode:

```
Public class CreateVurderingConnectionTest(string tilstand) extends testCase {
        Private CreateVurderingConnection crv;
        Protected void setup() throws Exception {
        Super.setUp();
        crv = new CreateVurderingConnection();
        }
        Public void testCreateVurdering() Throws Exception {
        Long Retur_kode = crv.createVurdering(123456,"Navn","Oprettet",131,0.34,current
        date,current time);
        assertTrue(Retur_kode > 0);
        }
        Public void testStatusVurdering() Throws Exception {
        Long Status_kode = crv.createVurdering(string tilstand);
        assertTrue("Status ikke Opr",Status_kode==tilstand);
```

```
        }
}
```

Template oprettes til brug for Create **Subjektiv** (SetUp_Subjektiv):

```
Protected void setUp()
//Vurderingsnummer,Beliggenhed, omsættelighed, køkken årstal type,oprettelses-dato,timestamp
//
Subjektiv subjektiv = new Subjektiv(123456,"Navn",131,0.34, 12.05.2006,11:12:34);
```

Template (Cr_Subjektiv) indeholder denne kode:

```
Public class CreateSubjektivConnectionTest(string tilstand) extends testCase {
        Private CreateSubjektivConnection crs;
        Protected void setup() throws Exception {
        Super.setUp();
        crs = new CreateSubjektivConnection();
        }
        Public void testCreateSubjektiv() Throws Exception {
        Long Retur_kode = crs.createSubjektiv(123456,"Navn",131,0.34, 12.05.2006,11:12:34);
        assertTrue(Retur_kode > 0);
        }
        Public void testStatusSubjektiv() Throws Exception {
        Long Status_kode = crs.statusSubjektiv(string tilstand);
        assertTrue("Status ikke Opr",Status_kode==tilstand);
        }

}
```

Template til de øvrige metoder bygges over samme skabelon som til CreateVurdering og CreateSubjektiv.

Vurderingsnummer er entydig nøgle. Beliggenhed kan indeholde værdierne 1 til 5 som via typetabeller oversættes til "Meget god" hhv. "Dårlig" i brugergrænsefladen.

Trace fra UPPAAL simuleringbilledet gemmes I text fil uppaal_til_junit.txt.

Der oprettes og kodes et konverteringsprogram der som input læser uppaal_til_junit. fil og output her fra er ny text-fil (JUnit_tests.java) med Java kode, som kan oversættes og afvikles.

Dette Konverteringsprograms pseudo kode er som følger:

```
//Start program---------------------------------------------------------
Open uppaal_til_junit. txt
While not eof input-fil
   Readline(Action, tilstand)
   Case Action:
   "Opret_Vurdering!": skriv Template SetUp_Vurdering();, skriv template CR_Vurdering(str tilstand);
   "Opret_Subjektiv!" : skriv Template SetUp_Subjektiv( );, skriv template CR_Subjektiv(str tilstand);
   "Opret_Faktuel!" : skriv Template SetUp_Faktuel( );, skriv template CR_Faktuel(str tilstand);
   "Opret_Projekt!" : skriv Template SetUp_Værdian( );, skriv template CR_P_Rate(str tilstand);
   "Godkende!" : skriv Template SetUp_Godkendt( );, skriv template AJ_Godkendt(str tilstand);
```

 ”Send_til_slet!” : skriv //Note… Send_til_slet! ikke implementeret;
 ”Genopret!” : skriv //Note… Genopret! ikke implementeret;
 ”Annuller!” : skriv //Note… Annuller! ikke implementeret;

 Otherwise: skriv //”Fejl --- ukendt Action ”.
 End-Case;
End-while;

Close input-fil.
//Slut program ---

Bemærk at Readline læser input og de to variable Action og tilstand indeholder værdierne fra dette input. Tilstanden er input til dannelse af template så der kan testes på denne ved afviklingen af den enkelte testcase.

Efter konverteringskørsel skal java-fil redigeres og tilstand opsættes som angivet i modellen.

Hver template afsluttes med tearDown() pr. klasse-navn, således at data kan oprettes uden fejl (dublicate violation) ved afvikling af den efterfølgende test-case.

Appendix B viser resultatet efter afvikling af konverteringsprogrammet på sekvenser fra afsnit 3.2.

Dette konverteringsprogram opretter public class pr. template og det er ikke vurderet i detaljer om det er mere hensigtsmæssig at samle en simulering i en samlet Klasse. Det skal dog nævnes at samling af lange traces givetvis vil virke uoverskueligt, hvis der er behov for at kigge i koden. Ideen med opbygning i mindre enheder kan derfor gøre overblikket større.

I Appendix A ses resultatet af random sekvensen fra afsnit 3.2. Det 2. eksempel indeholder kald til CreateFaktuel som fejler og fejl rapportering dannes.

4.2 INDLÆGNING AF TESTSUITES

I det følgende beskrives fremgangsmåden til indlæggelse af test i JUnit.

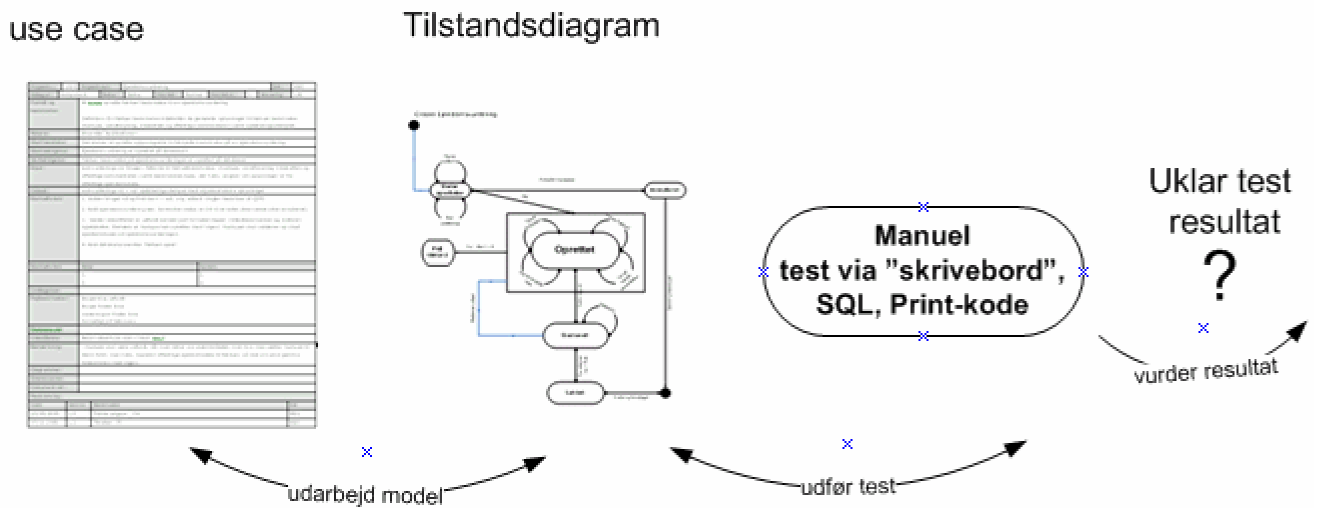
Den manulle måde foretages som i dag, dvs. udvikleren der kender program strukturen indlægger tests på den ”semi-maskinelle” måde ved, at enkelte trace-lister fra UPPAAL gennemgåes og udvalgte sekvenser udvælges (udplukkes, da der er uendelige mange) og indlægges i JUnit. I figur 4.2 nedenfor er der vist et eksempel hvor kun hovedklasser er valgt. Et andet eksempel på sekvens kan være:
4 gange PersonidLaes, 12 OpretSubjektiv, 1 OpretVaerdiansaettelse, 14 OpretFaktuel, 2 gange PersonidLaes, 3 OpretSubjektiv, OpretFaktuel osv.

```
TestSuite suite= New TestSuite();
suite.addtest(new PersonIdLaesConnectionTest("BSVN");
suite.addtest(new PersonIdLaesConnectionTest("ABAL");

suite.addtest(new OpretSubjektivconnectionTest(" ");
suite.addtest(new OpretFaktuelconnectionTest(" "));
suite.addtest(new OpretProjektconnectionTest(" "));
suite.addtest(new OpretVaerdiansaettelseconnectionTest(" "));


//Data slettes så suite kan køres igen.
TearDown();

TestResult result= suite.run();
```

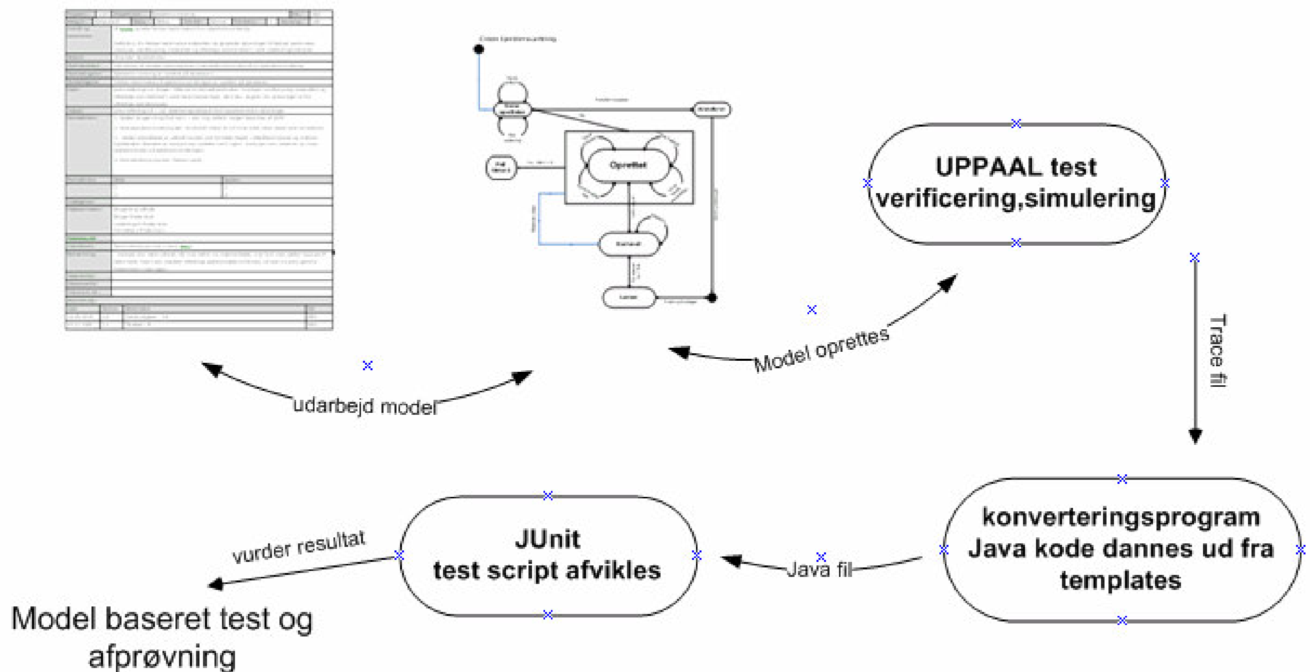
Figur 4.2: Kode til oprettelse af Testsuite

5 OPSUMMERING

I dette afsnit opsummeres fordele og ulemper ved brug af hhv. UPPAAL, JUnit og test-metoden angivet nedenfor i figur 5.2.



Figur 5.1 Eksisterende KUDV test model



Figur 5.2 Fremtidig KUDV test model

5.1 UPPAAL

Fordele:

-Tilstandsdiagrammer kommer mere i fokus og validering foretages på et tidligere tidspunkt. Der er mulighed for at få visualiseret hvordan tilstandene i systemet forventes at ændre sig.

- Verifikationsdelen er god til at få afklaret og forstå tilstandsændringer. Ved design af selv små systemer kan overblikket mistes, men her er der mulighed for via query forespørgsel, at verificere at forventelige tilstande kan nåes.

- Der er mulighed for at udvælge en delmængde af modellen f.eks. efter vigtighed og en komplet model er ikke nødvendig at efterprøve bestemte conditioner.

- simpel installation og mulighed for tilmelding til brugergrupper med hjælpefaciliteter/vidensdeling.

Ulemper:

-i princippet udviklet til Realtids systemer, dvs. mange funktioner er ikke nødvendig til Nykredits brug

-Et traceforløb kan ikke erstatte prototype eller anden visuel visning af hvordan systemet tænkes at virke.

- endnu et værktøj til den i forvejen store KUDV værktøjskasse

- Lange traces er omstændigt at indlægge manuel i JUnit. Dette er dog ikke værktøjets skyld, men der bør være mulighed for (evt. delvis) maskinel konvertering fra UPPAAL til JUnit

- Verifikationen er af system modellen og ikke det aktuelle system (produkt/prototype)

- Verifikation er kun så god som kvaliteten af modellen af systemet

5.2 JUNIT

Fordele:
- Refakturering i større grad bliver muligt, da JUnit understøtter regressionstest og dermed testes koden hver gang suiten sættes til at køre. Funktionaliteten burde være ok efter et fejlfrit gennemløb, dog afhænger dette at hvor godt de indlagte testcases dækker funktionaliteten.
- Der skabes et overblik over det aktuelle systems tilstand (antal ok/ikke ok JUnit tests)
- Maskinel rutine erstatter den nuværende manueller indlægning af JUnit testcases. En opgave som fravælges hvis der er knaphed med ressourcer.

Ulemper:
- Opgaven med at udtænke kreative test situationer er forankret hos udvikleren og validering af returkode fra server-kald som eksempel er blot toppen af isbjerget, men det er muligvis det, der kan give udvikleren tid i kalenderen.
- Modellens templates skal vedligeholdes til nye krav fra anvender

5.3 TEST-METODEN

Metoden har en fordel i at være semi automatisk (set i forhold til den nuværende manuelle metode i KUDV) og når modellen er oprettet og godkendt i UPPAAL bliver de sidste rutiner udført automatisk. Modellens achilleshæl er UPPAAL som er et stærkt værktøj, men integrationen med KUDV's øvrige værktøjer er ikke eksisterende. Der er ikke nogen mulighed for automatisk import af KUDV's tilstandsdiagrammer i vision til UPPAAL. Der kan være interesse i metodeafdeling for UPPAAL, men en mere generel brug i KUDV er ikke forventelig med det nuværende setup.

Ved at vælge UML-statecharts i stedet for UPPAAL kan implementering lettes, men hvis verificering i KUDV er et stort ønske er dette ikke muligt.

I KUDV er det sat ressourcer ind fra metode afdeling på, at JUnit skal udnyttes mere effektivt. Dette betyder at test-metoden's sidste del med brug af JUnit vil blive forsøgt implementeret ud fra selvgenereret trace eller manuel input.

Det manulle indgreb der skal foretages efter konverteringsprogrammet, hvor tilstanden skal hardkodes ind i sourcekoden virker ikke hensigtsmæssigt, men mit bud er, at det kan accepteres i en Beta version og en mere automatiske løsning burde kunne udvikles på kort sigt.

Test-proceduren er ikke fuldt implementeret og afprøvet pt. De enkelte dele er ok hver for sig, men integrationen med konverteringsprogrammet er ikke kørende.

6 KONKLUSION OG PERSPEKTIVERING

6.1 KONKLUSION

I denne rapport har vi bl.a. undersøgt, om vil det give øget kvalitet til software produktet, at bruge UPPAAL værktøjet i KUDV til, at verificere, simulere forretningsmodellen. Dette kan ikke besvares entydigt, men vi vil hævde at fokus på modellen i de tidlige faser vil fange mange uklarheder og misforståelse og dermed vil det endelige software produkt få en øget kvalitet.

Vi har endvidere udarbejdet en procedure der som input tager UPPAAL trace og semi-automatisk danner Java-kode der kan afvikles. Dette vil vi hævde også har en positiv effekt på software kvaliteten, idet tidlig afvikling af software med indlagte test gør, at fejl findes tidligere og ikke giver afledte fejl. Den semi-

automatiske procedure vil også betyde at der sættes større fokus på tidlig test af software. Med dette menes at test bør foretages struktureret, før og under programmering og ikke og som nu, hvor unit-testen foretages ud fra den enkelte udviklers erfaring.

Der er også i rapporten sat fokus på forretningsprocesserne og verificering heraf, hvor kunden og KUDV i samarbejde kan verificere at modellen er korrekt på et tidlig tidspunkt. En højere kvalitet i specifikationerne bevirker, at antallet af fejl i test-fasen og ændringer til specifikationen bringes ned.

Endelig vil et mere udbredt brug af JUnit i KUDV giver en større kvalitet i softwaren generelt, da det åbner op for refakturering. Ved at afvikle regelmæssige JUnit-test sikres det at systemerne og programmerne er valide og hurtig indgriben kan foretages, hvis der rapporteres fejl i test-cases.

Den udarbejdede test-procedure giver et forslag til indsats både på det forretningsmæssig plan, dvs. større indsats af forretningselementer tidlig i udviklingsprocessen, men også at der mere fokus på test afvikling og især med sigte mod en for for automatik. Disse elementer er for mig ikke modstridende, men også i overensstemmelse med IT-strategien i Nykredit og derfor vil implementering af rapportens ide have gode muligheder.

6.2 PERSPEKTIVERING

KUDV metode og it-udvikler må sammen finde en model for den fremtidige brug af verifikationsværktøj, da det giver gode muligheder for at højne softwarens kvalitet.

I den nære fremtid vil systemerne blive mere og mere integrerede og derfor er der større behov for model specifikation og verifikation heraf, således at Kunde og KUDV på et tidlig tidspunkt i projektforløbet bliver enige om kravene til det kommende system. Dermed undgås ressourcekrævende tilbageløb med tab for kunden og KUDV.

Brug af JUnit er blot et element som der skal bruges ressourcer på, men der er også mulighed for en standardisering af tesk-teknikkerne i KUDV. Her tænkes der på teknikker som kan integreres i de nuværende compilere og giver et bud på hvor test ressourcerne skal sættes ind.

LITTERATURLISTE

[1] http://junit.sourceforge.net

[2] Software Engineering – A proctitioner's Approach By Roger S. Pressman 2005

[3] http://www.uppaal.com

[4] Principles of model checking, Joost-Pieter Katoen 2002

[5] Java Development: Idea http://www.intellij.com

[6] Design and Validation of Computer Protocols, Gerard J. Holzmann

APPENDIX A

Class dk.nykredit.bsvn.MEjdvurdTest

Name	Tests	Errors	Failures	Time(s)
MEjdvurdTest	4	0	0	0.078

Tests

Name	Status	Type	Time(s)
testCreateVurdering	Success		0.000
testCreateProjektRate	Success		0.000
testStatusGodkendt	Success		0.000
testStatusLukket	Success		0.000

Class dk.nykredit.bsvn.MEjdvurdTest

Name	Tests	Errors	Failures	Time(s)
MEjdvurdTest	**5**	**0**	**1**	**0.078**

Tests

Name	Status	Type	Time(s)
testCreateVurdering	Success		0.000
testCreateProjektRate	Success		0.000
testCreateFaktuel	**Failure**	**expected:<0> but was:<99>** junit.framework.AssertionFailedError: expected:<0> but was:<99> at dk.nykredit.bsvn.MEjdvurdTest.testCreateFaktuel(Unknown Source) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)	**0.016**
testStatusGodkendt	Success		0.000
testStatusLukket	Success		0.000

APPENDIX B

```
Protected void setUp()
//Vurderingsnummer,type,Status,ejendomstype,sdd,dsd,oprettelses-dato,timestamp
//
Vurdering vurdering = new Vurdering(123456,"Navn", " ",131,0.34,12.05.2006,11:12:34);

Public class CreateVurderingConnectionTest(string tilstand) extends testCase {
        Private CreateVurderingConnection crv;
        Protected void setup() throws Exception {
        Super.setUp();
        crv = new CreateVurderingConnection();
        }
        Public void testCreateVurdering() Throws Exception {
        Long Retur_kode = crv.createVurdering(123456,"Navn","Oprettet",131,0.34,current
        date,current time);
        assertTrue(Retur_kode > 0);
        }
        Public void testStatusVurdering() Throws Exception {
        Long Status_kode = crv.createVurdering(string tilstand);
        assertTrue("Status ikke Opr",Status_kode==tilstand);
        }
}


//Note… Annuller! ikke implementeret;

Protected void setUp()
//Vurderingsnummer,Projekt_type, oprettelses-dato,timestamp
//
Projekt  projekt = new projekt(123456,"P-Navn",12.05.2006,11:12:34);


Public class CreateProjektConnectionTest(string tilstand) extends testCase {
        Private CreateProjektConnection crp;
        Protected void setup() throws Exception {
        Super.setUp();
        crp = new CreateProjektConnection();
        }
        Public void testCreateProjekt() Throws Exception {
        Long Retur_kode = crp.createProjekt(123456,"P-Navn",12.05.2006,11:12:34);
        assertTrue(Retur_kode > 0);
        }
        Public void testStatusProjekt() Throws Exception {
        Long Status_kode = crp.statusProjekt(string tilstand);
        assertTrue("Status ikke Opr",Status_kode==tilstand);
        }

}
```

//Note… Annuller! ikke implementeret;
//Note… Genopret! ikke implementeret;
//Note… Godkende! ikke implementeret;
//Note… Send_til_slet ikke implementeret;

APPENDIX C