Bachelor projekt

Dipl.Ing.IT

Dokumentationssystem til understøttelse af udvikling og vedligeholdelse af produktkonfigureringssystemer

Anders Degn

Kongens Lyngby, 29. april 2006
IMM-B.Eng-2006-17

Technical University of Denmark
Informatics and Mathematical Modelling
DTU – Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-B.Eng-2006-17

Forord

Dette bachelor projekt er skrevet hos Institut for Matematisk Modellering hos Danmarks Tekniske Universitet DTU i forbindelse med erhvervelsen af bachelor graden på diplomingeniøruddannelsens IT-retning.

Opgaven omhandler udviklingen af et dokumentationssystem til understøttelse af udvikling og vedligeholdelse af produktmodeller. Der er hovedsageligt sat fokus på systemets brugerinterface idet dette skal overholde krav opstillet af samarbejdsinstituttet IPL.

Opgaven består af denne rapport, sourcecode på CD-Rom, en eksekverbar version af programmet samt en test-produktmodel ligeledes på CD-Rom samt to research papers der forventes udgivet andetsteds i Juni 2006.

Projektet er udført i perioden 2. februar 2006 til 1. maj 2006.

Kongens Lyngby, 29. april 2006

Anders Degn

Inkluderede dokumenter

Anders Haug and Lars Hvam - The modelling techniques of a documentation system that supports the development and maintenance of product configuration systems [Haug & Hvam (2006a)]

Anders Haug and Lars Hvam - CRC-cards for the development and maintenance of product configuration systems Anders Haug and Lars Hvam [Haug & Hvam (2006b)]

Anerkendelser

Jeg takker min vejleder Bjarne Poulsen på IMM ligesom min vejleder Anders Haug på IPL for godt samarbejde under hele forløbet og gode råd mht. denne rapport.

Desuden retter jeg en tak til Jesper Riis og Jesper Nielsen hos GEA Niro A/S, der tog sig tid til at forestille deres nuværende dokumentationssystem og gav feedback på min løsning.

Indhold

Forord iii
Inkluderede dokumenter v
Anerkendelser vii
Indhold ix

Kapitel 1

Introduktion 1
1.1 Indledning 1
1.2 Fremgangsmåden 3
1.2.1 Analysemodellen 3
1.2.2 Designmodellen 5
1.3 Status 5
1.4. Problemstilling 5
1.5 Krav 6
1.6 Afgrænsning 7
1.7 Tidsplan 8
1.8 Risici 8

Kapitel 2

Teknologi og værktøjer 10
2.1 Microsoft .NET Framework 10
2.2 Microsoft Visual Studio 2003 12

Kapitel 3

Design og arkitektur 13
3.1 Datamodellen – PMData 13
3.2 Præsentationskomponenten - PMDocTool 15
3.2.1 Design 16
3.2.2 Arkitektur 17

Kapitel 4

Implementering 23

4.1 PMData 23
4.1 PMDocTool 26

Kapitel 5

Test 34
5.1 Test af funktionelle krav 35
5.2 Analyse af mulige fejl 37
5.3 Test af analyserede fejl 37
5.4 Unit test 38
5.5 Løsning på fejl fundet i tests 41

Kapitel 6

Konklusion 42
6.1 Fremtidige forbedringer 43
6.1.1 Arkitektur 43
6.1.2 Implementering 44
6.2 Demonstrationsmøde med GEA Niro A/S 45

Appendiks A

Referencer 47

Appendiks B

Brugervejledning 48

KAPITEL 1

Introduktion

Dette eksamensprojekt er opstået på baggrund af forskning udført ved Institut for Produktion og Ledelse (IPL) vedrørende skabelse af et dokumentationssystem til understøtning af udvikling og vedligeholdelse af produktkonfigureringssystemer. Mens tidligere forskning [Hvam & Malis, 2002; Hvam et al., 2005] primært har fokuseret på funktionelle krav, er der i forskningen af Anders Haug opstået en mere detaljeret specifikation af, hvorledes et sådant system kan udformes [Haug, 2006a; Haug, 2006b].

Formålet med dette eksamensprojekt er at udvikle en prototype på et dokumentationssystem på baggrund af disse specifikationer for herved at afklare i hvor stor grad, det specificerede koncept [Haug, 2006a; Haug, 2006b] er muligt at opfylde, samt hvilke supplerende definitioner eller tilpasninger, der eventuelt må være behov for.

1.1 Indledning

Igennem mange år har virksomheder, der har beskæftiget sig med produkter, der bliver fremstillet stykvis – eller i små serier – tilpasset produkterne kundens individuelle ønsker. I de senere år er også masseproducerende virksomheder blevet berørt af kunders individuelle ønsker.

Når en virksomhed skal tilpasse et produkt til en kundes individuelle ønsker, er flere afdelinger involveret for at kunne kalkulere en pris og give en forventet leveringstid. Der skal oprettes styklister og produktionen skal forberedes m.m. Denne proces er ofte meget tidskrævende og kan vha. en

såkaldt produktmodel effektiveres. Produktmodellen indeholder viden om produktet (f.eks. materialer, mål, farver og sammensætningsmuligheder) og om fremstillingen af produktet.

En produktmodel implementeres i et produktkonfigureringssystem, som benyttes til at understøtte specifikationer for kundetilpassede produkter. Gevinsten ved at benytte et sådant system kan være enorm og effekten kan iagttages i flere forskellige virksomheder, der har implementeret produkt-konfigurerings systemer og benytter disse:

Sandviken – en svensk virksomhed, der producerer stålholdere, benytter [Hvam et al., 2005] konfigureringssystemer til kalkulation af pris, produktspecifikationer og produktionsgrundlag for stålholdere, der er tilpasset kundernes individuelle ønsker. Indtil indføringen af systemerne, har virksomhedens medarbejdere brugt 2-3 uger på udarbejdelsen af tilbud, specifikationer og produktionsgrundlag for enhver ny model af en stålholder. Med konfigureringssystemet kan sælger nu indtaste parametre, der beskriver kundens krav. Konfigureringssystemet udarbejder selv styklister, tegning og produktionsgrundlag – her f.eks. CNC-kode – ligesom prisen på stålholderen. Fra kunden tager kontakt til sælgeren til kalkulationen af prisen, specifikationerne og produktions-grundlaget samt tilbuddet til kunden er udarbejdet, går der nu ca. 10 minutter.

F.L.Smidth er en dansk entreprenør virksomhed, der har specialiseret sig i fremstilling af cement-fabrikker og udstyr til cementfremstilling. På verdensplan er virksomheden den største af sin slags og sidder på over halvdelen af markedet. F.L.Smidth har [Hvam et al., 2005] implementeret et konfigureringssystem til understøttelse af udarbejdelsen af budgettilbud. Et sådant tilbud blev tidligere udarbejdet over 3-5 uger. Under udarbejdelsen var udover sælgeren flere specialister involveret til at bidrage med viden og erfaringer om udviklingen af cementfabrikker. Denne viden er nu implementeret i konfigureringssystemet og sælger kan hermed udvikle budgettilbudet på 1-2 dage.

Udover tidsbesparelsen har begge virksomheder kvalitetsmæssige fordele af produktmodellerne.

Mange andre virksomheder benytter konfigureringssystemer. Dell Computers adskillige individuelt kunde-tilpassede systemer, med manuel specifikation af de kundetilpassede produkter ville være en næsten umulig opgave (i alt fald meget tidskrævende) uden konfigureringssystemet. Af andre virksomheder, der benytter konfigureringssystemer kan nævnes American Power Conversion (APC), Aalborg Industries, NEG-Micon, GEA-Niro og IBM-SMS. Flere bilproducenter som VW, BMW, Ford og Volvo tilbyder mulighed for at konfigurere sin egen kommende bil i et online konfigureringssystem.

Produktmodeller og konfigureringssystemer finder altså på nuværende tidspunkt anvendelse i virksomheder verden over.

1.2 Fremgangsmåden

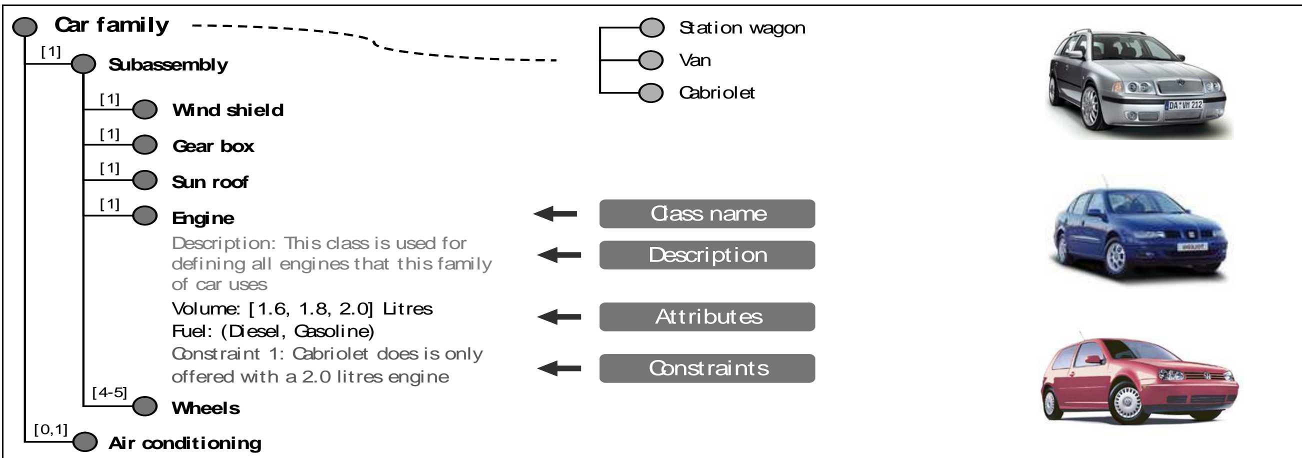
Fremgangsmåden for opbygning af konfigureringssystemer bygger på den objektorienterede projektlivscyklus (Unified Process) og indeholder bl.a. metoder til analyse af de forretningsprocesser, der skal understøttes ligesom analyse og modellering af et produktsortiment. Fremgangsmåden består af syv faser [Hvam et al., 2005]:

1.: Udvikling af specifikationsprocesser
2.: Analyse af produktsortiment
3.: Objektorienteret analyse
4.: Objektorienteret design
5.: Programmering
6.: Implementering
7.: Vedligeholdelse og videreudvikling

I de følgende afsnit belyses resultaterne af faserne 2-4 – Analysemodellen og Designmodellen. Disse modeller er baggrund for udviklingen af det dokumentationsværktøj, der skal udvikles i dette projekt. Værktøjet vil desuden kunne benyttes som understøttelse til fase 7 - vedligeholdelse og videreudvikling af de udviklede modeller.

1.2.1 Analysemodellen

Analysemodellen afspejler produktets elementer og struktur – dvs. f.eks. en kombination af materialer, dimensioner, sammensætningsmuligheder, farver osv. Således kan alle varianter af et produkt beskrives med ét diagram - f.eks. en produkt variant master (PVM). Denne består af to dele, part-of struktur, der beskriver de dele, der indgår i hele produktserien og kind-of struktur, der beskriver de udskiftelige dele i produktet. I OO terminologi svarer part-of til aggregering og kind-of til generalisering.



Figur 1.2.1.1 - Produktmaster [Hvam et al., 2005b]

De enkelte dele tegnes som en forgrening med et symbol og beskrives med attributter. Streger mellem modulerne beskriver hvilke dele, der kan sammensættes. Disse markeres med beskrivelser af de regler, der gælder for sammensætningen af delene.

Udover at kunne beskrive produktet, kan en PVM også benyttes til at beskrive produktets livsfaser - såsom f.eks. produktionssystemet.

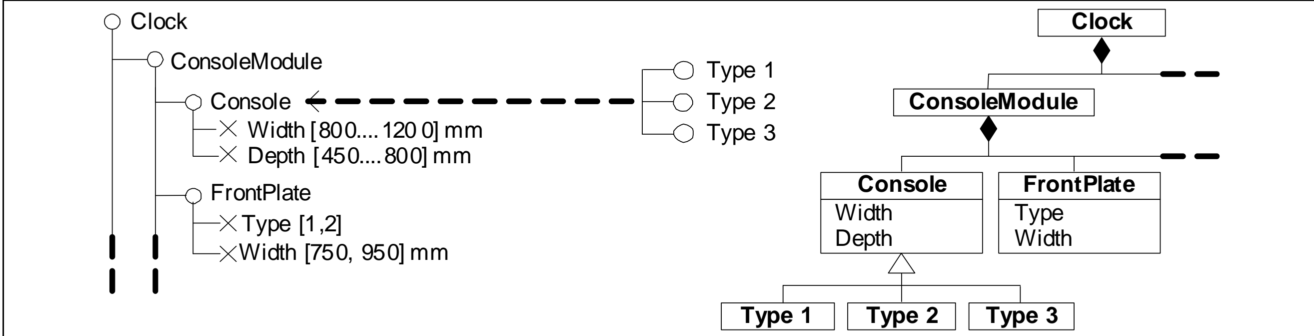
Klasserne, der indgår i PVM-diagrammet, kan beskrives yderligere på CRC-kort.

Class name			Status		Change requsts	Version
▷ Basic information						
Created by:		Date:	Card responsible:		Product responsible:	
Responsibilities:						
▷ Relationships						
Aggregation	Superparts:			Subparts:		
Generalisation	Superclasses:			Subclasses:		
Dependency	Sources:			Clients:		
Association	Classes:					
▷ Sketch/Picture						
Type 1	Type 2	Type 3		Width	Height	Depth
▷ Product knowledge						
Attributes	**Name**	**Values**	**Unit**	**Type**	**Inherited**	**Comments**
	Height	30; 35; 40	mm	[Integer]		
	Width	30; 40; 50	mm	[Integer]		
Constraints	**Name**	**Description**			**Inherited**	**Comments**
	§1	Heigth = Chair.Height + 300				
	§2	Type2 -> Legs.Type1 or Legs.Type3				
Methods	**Name**	**Description**			**Inherited**	**Comments**
	CalcArea ()	Height * Width				

Figur 1.2.1.2 –Design af CRC kort – sammensat af figurer fra [Haug & Hvam, 2006b]

PVM-diagrammet kan umiddelbart transformeres til en objektorienteret beskrivelse af analyse-modellen i form af et UML klassediagram. Klassediagrammet indeholder de samme klasser og attributter som PVM-diagrammet. Ved overførsel af klasser fra PVM til klassediagram optimeres modellen, evt. ved at slå flere klasser sammen. [Riis, 2003; Hvam et al., 2005].

PVM diagrammet er resultatet af fase 2 imens klassediagrammet er resultatet af fase 3.



Figur 1.2.1.3 –Transformationen fra PVM til klassediagram

1.2.2 Designmodellen

Informationer fra analysemodellen benyttes i den næste fase (fase 4) - en objektorienteret designmodel i form af et UML klassediagram. Klassediagrammet skal senere benyttes til implementeringen af konfigureringssystemet. Det indeholder udover komponenterne PVM-diagrammet eventuelt klasser, der skal bruges til udvikling af et userinterface/brugergrænseflade.

1.3 Status

Der findes forskellige værktøjer, der bliver benyttet til at tegne CRC-kort og UML med. På nuværende tidspunkt benyttes hertil oftest programmer som MS Visio og Excel til tegning af UML diagrammer. CRC kortene skrives oftest i et almindeligt tekstbehandlingsprogram – f.eks. MS Word. På nuværende tidspunkt findes der ikke nogen værktøjer, der er beregnede til at tegne PVM-diagrammet. Til vedligeholdelse af CRC kort har virksomheden Niro udviklet et værktøj i form af en Lotus Notes skabelon. Skabelonen indeholder et træ af CRC kort, som ligesom produktmasteren er opbygget som en whole-part struktur. Opbygningen af træstrukturen giver imidlertid ikke mulighed for at vise generaliseringer - dvs. kind-of sammenhænge. Heller ikke afhængigheder kan vises i træstrukturen. Attributter, samt andre symboler og relationer fra klassediagrammer kan ligeledes ikke illustreres. Manglerne skyldes begrænsningerne i et standard Datapool system som Lotus Notes.

1.4. Problemstilling

På IPL er der gennem en årrække blevet udført megen forskning i, hvordan produktkonfigurerings-systemer (PKS) kan opbygges. Forskningen har indtil nu ført til, at der eksisterer metoder til udvikling af produktkonfigureringssystemer ved hjælp af CRC kort, produktmaster og UML diagrammer. Metoderne benyttes på nuværende tidspunkt i industrien. Et integreret værktøj til

understøtning af disse teknikker er aldrig blevet udviklet. Det har derfor været nødvendigt for produktudvikleren at beskrive de samme informationer (data) i flere modeller hvilket implicerer manuelle overførsler af information mellem modeller.

Problemerne i indføringen af produktmodeller og konfigureringssystemer i virksomheder ligger ved hånden [Hvam, L., Mortensen, N.H. & Riis, J. (2005b)]:

1. Udviklingen af produktmodellen er tidskrævende og efter færdigudvikling skal modellen løbende vedligeholdes – ligesom virksomheden løbende opdaterer sit produktprogram. Opdateringen af produktmodellen skal implementeres i konfigureringssystemet for at virksomhedens medarbejdere kan arbejde med de ny produktvarianter. Det kan let ske, vedligeholdelsen bliver forsømt og systemet forældet – og dermed ikke kan bruges
2. Systemet bliver udviklet uden at tage stilling til virksomhedens procedurer. Herved skal procedurer, der gennem mange år er blevet indført pludseligt omlægges. Dette medfører oftest stress og vrede blandt medarbejderne og uvildighed til brugen af systemet. Et system, der har kostet lang udviklingstid og fungerer godt i teorien, risikerer aldrig at blive benyttet i praksis.

Punkt 2 kan løses, idet der forud for udviklingen af produktmodellen og implementeringen i konfigureringssystemet tages stilling til de nuværende procedurer og systemet dermed tilpasses medarbejdernes krav.

Det første punkt kan løses ved at vedligeholdelsen af produktmodellen og implementeringen i konfigureringssystemet gøres lettere og at modellen og systemet dokumenteres omfattende. Dokumentationen og vedligeholdelsen af produktmodellen kan lettes betydeligt idet der benyttes et specifikt vidensbaseret værktøj til dokumentation og udvikling af produktmodeller. Værktøjet skal kunne håndtere de forskellige sprog / diagrammer, der benyttes til udviklingen af produktmodellen. Således skal CRC-kort, PVM og UML diagrammer kunne udvikles og opdateres. En opdatering i ét af diagrammerne skal automatisk afspejle sig i de andre. Ændringer skal fastholdes, så det er muligt at gå tilbage til en tidligere version af produktet. At der er tale om et vidensbaseret værktøj betyder, at viden om produktmodeller skal være implementeret som regler i systemet. Hermed opnås muligheden for at produktmodellens konsistens valideres. På nuværende tidspunkt findes et sådant værktøj ikke, men der har været gennemført megen research omkring hvilke funktioner at sådant værktøj bør have. Denne research har dog ikke ført til nogen konkrete forestillinger om, hvordan et brugerinterface kan opbygges og endnu vigtigere: hvordan de forskellige diagrammer og CRC kortene hænger sammen.

1.5 Krav

Anders Haug specificerer i [Haug & Hvam, 2006b] hvordan CRC kort i en integreret applikation kunne se ud. Haug og Hvam (2006a) beskriver , hvordan et dokumentationsværktøj til produktmodeller kunne opbygges.

I dette projekt skal der udvikles en prototype af et dokumentations system til produktmodeller, der implementeres iflg. specifikationerne fra Haug og Hvam (2006). Projektet er dermed tænkt som et ”proof of concept”.

1.6 Afgrænsning

Der skal udvikles en fælles datastruktur for de forskellige diagrammer og beskrives transformationerne mellem diagramtyperne. Jeg vil beskrive mulighederne i forbindelse med eksport af den færdige produktmodel, så denne kan indlæses i et konfigurationssystem. Herved begrænses nødvendigheden af den viden en produktudvikler skal have inden for udvikling af IT-systemer, hvorved denne kan koncentrere sig om sit eget domæne - dvs. produktudvikling.

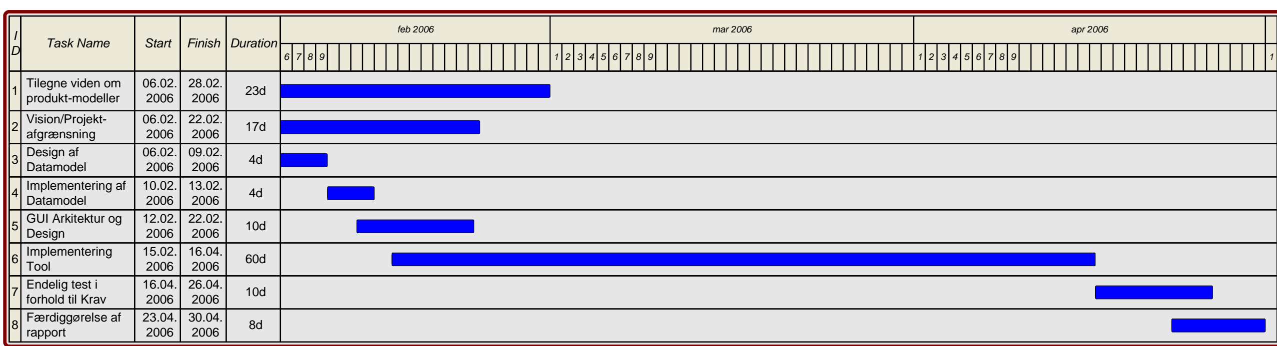
Der skal udvikles en prototype af et værktøj, der kan benyttes til at udvikle CRC-kort og tegne UML- og PVM-diagrammer.

- Programmet skal benytte fælles data til CRC-kortene og de to diagramtyper.
- Det skal være muligt at ændre på data i alle tre forskellige diagramtyper. Når der skiftes til en anden visning, skal ændringerne afspejles i de andre diagramtyper.
- Programmet skal være opbygget, så brugeren frit kan skifte imellem de tre visninger. Det skal være muligt at se mere end én visning på skærmen ad gangen. Ved ændring af data i én visning, skal de andre viste visninger med det samme opdatere sig (observer).
- Det skal være muligt at arbejde med diagrammer, der er større end skærmbilledet. Dette gøres ved at muliggøre scrolling og evt. zoom.
- Programmet skal have en undo funktion, der giver mulighed for at fortryde den sidste ændring. Undo skal være muligt tilbage til programstart.
- Der skal være mulighed for udskrift af diagrammerne på en printer.
- Diagrammet skal kunne gemmes på harddisken til senere brug og videreudvikling.
- Det skal være muligt at eksportere produktmodellen til f.eks. en xml-fil. En eksporteret fil kan f.eks. benyttes af produktkonfigurationssystemer.
- Lagring af data skal i prototypen ske i form af en projektfil, der gemmes på harddisken. Der vil ikke være mulighed for lagring i en database.

1.7 Tidsplan

Projektet påbegyndes den 6. februar 2006 og skal afleveres per 1. maj 2006.

I nedenstående Gannt Chart er en tidsplan estimeret.



ID	Task Name	Start	Finish	Duration
1	Tilegne viden om produkt-modeller	06.02.2006	28.02.2006	23d
2	Vision/Projekt-afgrænsning	06.02.2006	22.02.2006	17d
3	Design af Datamodel	06.02.2006	09.02.2006	4d
4	Implementering af Datamodel	10.02.2006	13.02.2006	4d
5	GUI Arkitektur og Design	12.02.2006	22.02.2006	10d
6	Implementering Tool	15.02.2006	16.04.2006	60d
7	Endelig test i forhold til Krav	16.04.2006	26.04.2006	10d
8	Færdiggørelse af rapport	23.04.2006	30.04.2006	8d

Figur 1.7.1 – Tidsplan for projektforløbet

Grundet opgavens karakter – udviklingen af en software til dokumentation af produktmodeller ud fra en given specifikation, der udelukkende specificerer det færdige brugerinterface, er implementeringen af det grafiske brugerinterface den vigtigste del af opgaven. Det vil af samme grund også være denne del, der vil blive brugt mest udviklingstid på. Til implementeringen af præsentations-komponenten er der derfor afsat $^2/_3$ af den samlede tid, der er afsat til hele projektet. Til design af funktion og udseende på samme del er der afsat forholdsvist lidt tid. Dette valg er foretaget fordi dette design er specificeret på forhånd i [Haug & Hvam (2006a & 2006b)]. Der skal derfor kun udvikles en software arkitektur inden implementeringen.

Der vil gennem implementeringsfasen løbende blive foretaget tests op imod kravene i [Haug & Hvam (2006a & 2006b)]. Udviklingen er planlagt som en tilnærmet iterativ udviklingsproces. En iterationsplan bliver ikke lavet. I stedet gennemgås de enkelte udviklingstrin med Anders Haug. På denne måde vil det være muligt, hurtigt at finde frem til fejl og misforståelser, der ligeledes hurtigt kan rettes.

Ved implementeringen af GUI'et implementeres de tre views i rækkefølgen CRC view, PVM view og ClassDiagram view.

1.8 Risici

Som i alle andre udviklingsprojekter er der forskellige risici at tage højde for. Der fokuseres på tre områder. Her beskrives hvad der kan gå galt, samt hvorledes problemer undgås eller tackles i opløbet.

Person risici omhandler hvad der kan gå galt på det personlige niveau.

Skulle jeg under udviklingsforløbet blive ramt af sygdom, er der ingen mulighed for at overdrage dette projekt til andre. Det er derfor i dette tilfælde nødvendigt at søge om dispensation til udsættelse af opgavens afleveringstidspunkt.

Data risici omhandler risici og reduktion af selv samme i forbindelse med håndtering af dokumenter og sourcecode.

Designdokumenter, sourcecode, rapportudkast og alle andre dokumenter lagres på en Pc's harddisk. Efter hver afsluttet arbejdsdag pakkes alle filer og uploades til en ftp server hos b-one.dk. Af denne server tages dagligt backup af provideren. Det antages at være usandsynligt, at både lokal harddisk, ftp-server og backup af ftp-server går tabt under normale forhold.

Program risici diskuterer risici i forbindelse med kompleksiteten af produktet.

Skulle kompleksiteten af omfanget af implementeringen føre til risiko for at deadline ikke overholdes, kan kravene til softwaren skæres ned i form af afgrænsning. Den tilnærmede iterative udviklingsproces betyder, at afgrænsning af krav ikke vil føre til et program, der ikke kan køre.

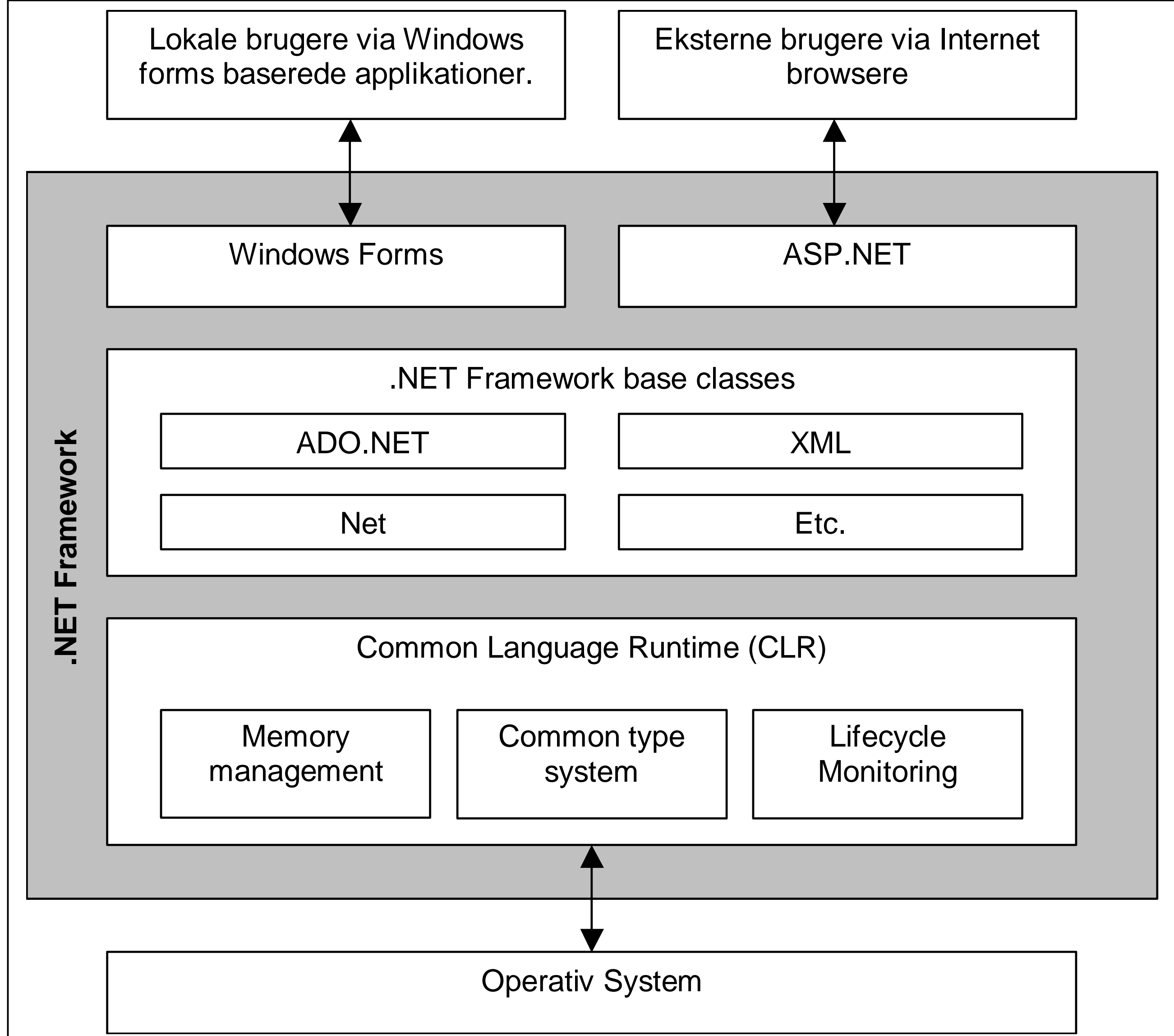
KAPITEL 2

Teknologi og værktøjer

2.1 Microsoft .NET Framework

Som teknologisk platform er løsningen bygget på Microsofts .NET Framework. Denne platform må betragtes som værende blandt de mest moderne på markedet, og kan benyttes til såvel webapplikationer, som traditionelle applikationer.

.NET frameworket er struktureret på følgende vis:



Figur 2.1.1 – Strukturen af .NET frameworket. Kilde: Professional VB.NET (Wrox).

Løsningen er primært bygget op omkring de klasser der findes i Windows Forms (System.Windows.Forms namespace), samt .NET reflection komponenterne. Frameworket er bygget op således at koden - hvad enten den er skrevet i C# , VB.NET eller et af de mange andre understøttede sprog, bliver kompileret til et assembly lignende sprog kaldet MSIL (Microsoft Intermediate Language). Denne kode JIT compiles først senere til x86. Det vil sige, at først når en applikation forsøges eksekveret, bliver den lavet til ”gammeldags” x86 instruktioner.

.NET er et såkaldt *managed environment* ligesom f.eks. Java. Det er derfor ikke er muligt at tilgå memory områder vilkårligt. Derudover er der ligesom i Java automatisk garbage collection, hvilket også betyder at det er svært at holde øje med en applikations memoryforbrug under test.

2.2 Microsoft Visual Studio 2003

Microsofts Visual Studio bliver benyttet som udviklingsværktøj igennem projektet. VS tilbyder, udover en god editor, en user interface designer, der letter og simplificerer skabning af GUI.

VS byder yderligere på muligheder, der letter debugging, bl.a. med debug output, breakpoints og ”watches” til at læse værdien af variable at-run-time.

KAPITEL 3

Design og arkitektur

Programmet skal opbygges af to komponenter – en præsentationskomponent og en datamodel.

Præsentationskomponenten skal benyttes til visning og redigering af data i datamodellen og skal tilbyde tre forskellige diagrammeringsformer.

Datamodellen skal rumme alle informationer, der skal benyttes til visning og lagring af informationerne i de tre forskellige diagrammeringsformer.

3.1 Datamodellen – PMData

I [Haug & Hvam, 2006a] beskrives nødvendigheden for en transformation mellem CRC kort, PVM diagram og klassediagram. Denne har hidtil været udført manuelt. [Haug & Hvam, 2006a] lægger op til, fortsat at udføre en transformation diagrammerne imellem. Dette ville betyde for dokumentationsprogrammets datamodel, at data i CRC kort skulle kopieres til PVM diagrammet og videre til klassediagrammet og i [Hvam et. al 2005b] endda mellem analysemodellens klassediagram og designmodellens klassediagram.

I stedet for at benytte denne fremgangsmåde har jeg besluttet, at genbruge data fra CRC kort i både PVM diagrammet og i klassediagrammet. Datamodellen er derfor opbygget, som en objektorienteret datamodel, der kan afspejle en klasse i både PVM diagrammet og klassediagrammet ligesom i analyse- og designmodellen med alle dens attributter. Heri indgår alle

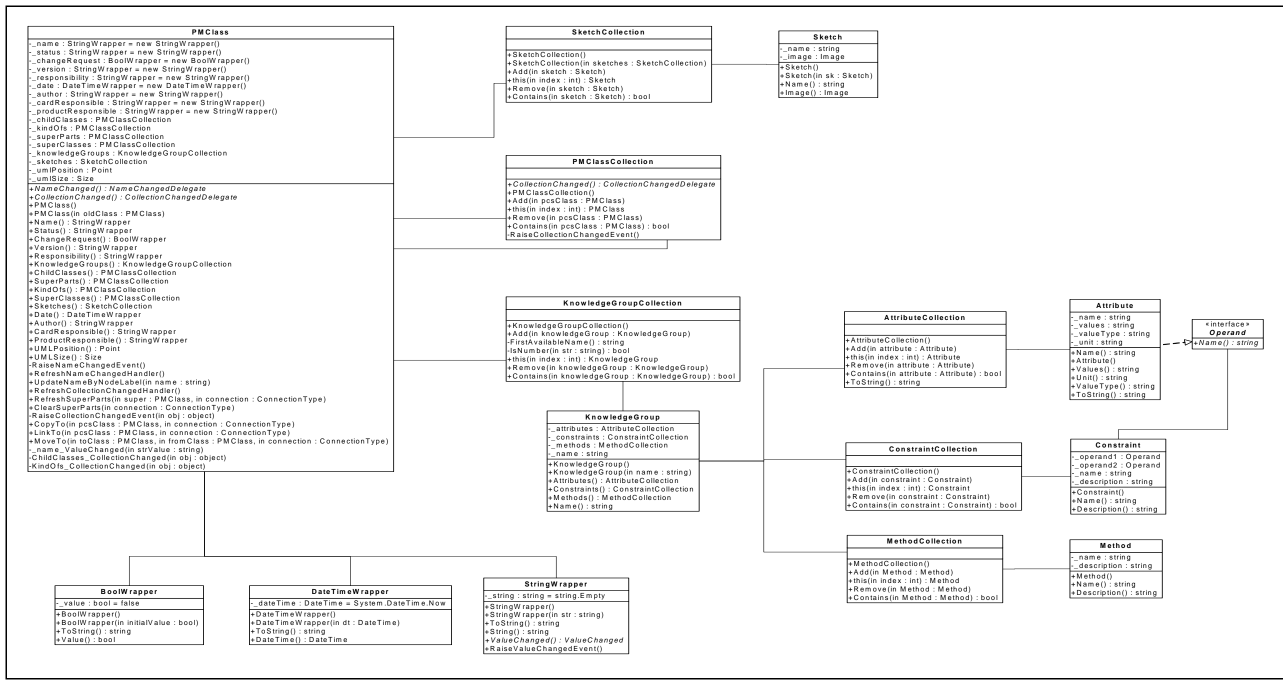
de attributter, der er beskrevet i [Haug & Hvam, 2006a og 2006b]. Udover disse attributter er det nødvendigt at lagre informationer om klassernes fysiske placering i klassediagrammet. Dette diagram er det eneste, hvori det er nødvendigt for brugeren, selv at kunne placere klasserne fysisk på en tegning. PVM diagrammets struktur gør muligheden for manuelt at positionere klasserne unødvendig, da positioneringen af klasserne her er fastlagt. [Hvam et al., 2005b]

Der er kun enkelte iøjnefaldende ting ved datamodellens UML (Figur 3.1.1):

1. Datamodellen er forberedt på at kunne håndtere en række af de krav, der i [Haug & Hvam, 2006a] stilles til denne type systemer. To af disse krav vil ikke blive implementeret i prototypen, men alligevel er datamodellen forberedt på disse:

   a) Prototypen vil kun give mulighed for én Knowledgegroup på hvert CRC-kort. Opbygningen af PMData med en én til mange relation mellem PMClass og KnowledgeGroup – her givet ved en Collection af KnowledgeGroup objekter på PMClass objektet – muliggør let en udvidelse af præsentationskomponenten, så dette krav imødekommes.

   b) I den endelige version skal der henvises til de involverede klasser og attributter i form af et hyperlink i en klasses Constraints. I prototypen skal ikke henvises til de involverede klasser. I stedet skal en Constraint blot indtastes i form af en tekst, der beskriver hvilke klasser og attributter, der er involveret. Realiseringen af kravet er forberedt, idet Attribute implementerer Operand interfacet, der benyttes af Constraint. Iflg. kravet skal der dog også være mulighed for at såkaldte ”KindOf” klasser skal indgå i Constraints. En KindOf er blot en PMClass. Dermed bør PMClass i senere versioner også implementere Operand.



Figur 3.1.1 – UML Diagram for datamodellen – PMData

2. Der ses tre wrapper klasser, der blot indpakker hhv. String-, DateTime- og Bool-typer i objekter. Data i PMClass skal kunne ændres i præsentations komponenten. Ændringerne skal kunne foretages direkte på PMClass attributter. Det vil dermed være nødvendigt at kunne referere direkte til PMClass' attributter.

   Hertil er det nødvendigt at kigge på frameworkets måde at overføre parametre med ValueType semantik til metoder. Typer, der nedarver fra `System.ValueType` passes i frameworket by value. Dvs. der oprettes en kopi af værdien på heapen. Denne kopi passes til metoder med ValueType-variable i argumentet.

   Præsentationskomponenten skal kunne udføre ændringer på PMClass' attributter og ikke på kopier af disse. Selvfølgelig findes der metoder til dette:

   - Variablen kan passes som reference vha. keyword'et `ref`. Det er imidlertid ikke muligt at passe et objekts Properties by ref. Dermed er denne metode ikke brugbar i denne situation.

   - Variablen kan boxes. Ved boxing castes ValueType variablen til et object, der ligger på heapen og kan passes som en reference. I metoden, der har variablen som argument kan denne igen unboxes til den oprindelige ValueType type. Boxing og unboxing kan ske både implicit og eksplicit.

   - Variablen kan pakkes ind i en wrapper klasse. Herved ligger *referencen* til ValueType variablen på heapen og kan passes uden at der oprettes en kopi af variablen.

   Sidstnævnte to metoder er begge brugbare. Jeg har dog valgt den sidste metode, da denne muliggør yderligere funktioner, der kan være brugbare. Bla. indeholder StringWrapper klassen en Event, der fyres af når værdien af strengen ændres.

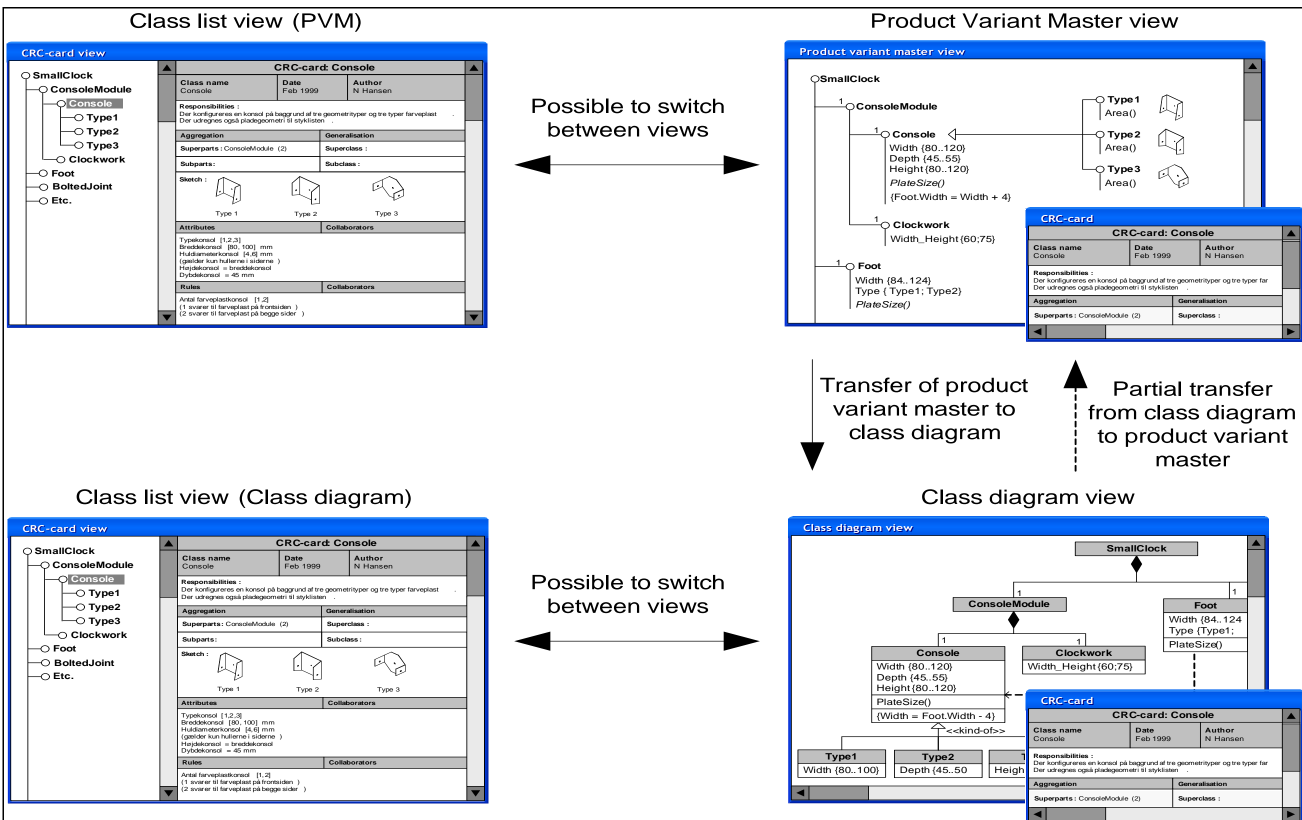
   `String` typen er ikke en ValueType, men den behandles som en ValueType, idet samtlig manipulation af en string returnerer en ny instans af String-objektet. Det vil altså til enhver tid være en manipuleret kopi af objektet, der returneres og ikke det oprindelige objekt med udførte manipulationer, der returneres. Derfor har denne – ligesom Bool og DateTime en ValueType semantik. Det er af denne årsag nødvendigt at behandle den som en ValueType.

3.2 Præsentationskomponenten - PMDocTool

I de følgende afsnit beskrives brugergrænsefladens design og arkitektur. Mange steder er arkitekturen ikke optimal. Mange af de afsnit, der omhandler klasser hvis arkitektur kan forbedres afsluttes med en kort konklusion, der indeholder idéer til en evt. refactoring, der vil føre til bedre arkitektur og dermed bedre maintainability. Arkitekturen illustreres med UML diagrammer.

3.2.1 Design

Softwarens præsentationskomponent er den grafiske brugergrænseflade (GUI). GUI'ets design er beskrevet i [Haug & Hvam, 2006a og 2006b].



Figur 3.2.1.1 – Eksempel på, hvordan brugerinterfacet kan se ud i de tre forskellige visninger[Haug & Hvam (2006a)]

Figur 3.2.1 viser et eksempel på. hvordan brugerinterfacet skal se ud. Det er ønsket, at der findes tre forskellige views i softwaren:

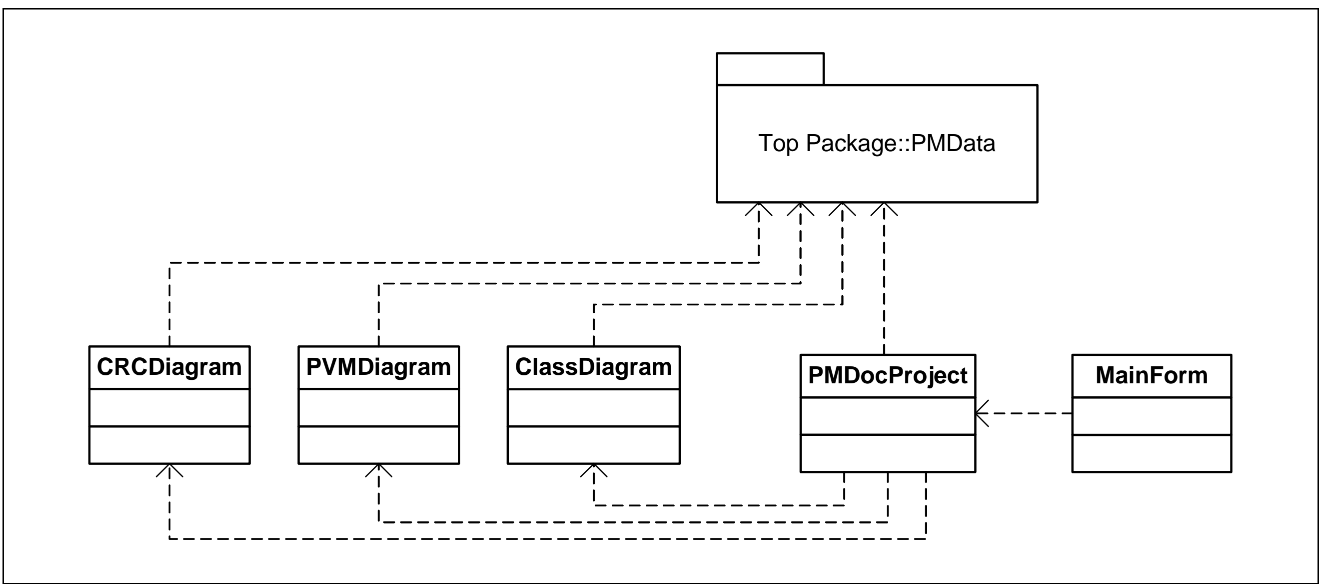
1. Et CRC kort view. Her skal i venstre side vises en træstruktur, hvori alle CRC kort findes. I højre side ses det i træet aktuelt valgte CRC kort. Det er muligt at udføre ændringer på CRC kortet i højre side.
2. Et Produkt Variant Master (PVM) view. Her ses et PVM diagram (som beskrevet i afsnit 1.2.1) i programmets hovedvindue. Dobbeltklikkes der på et element i diagrammet, åbnes et dialogvindue. I dialogvinduet ses det til det valgte element tilhørende CRC kort. Det er muligt at udføre rettelser i CRC kortet.
3. Et Klassediagram View. Her ses et klassediagram (som beskrevet i afsnit 1.2.1)i programmets hovedvindue. Dobbeltklikkes der på en klasse i diagrammet, åbnes et dialogvindue. I dialogvinduet ses det til det valgte element tilhørende CRC kort. Det er muligt at udføre rettelser i CRC kortet.

Pilene mellem de forskellige visninger illustrerer muligheden for at skifte fra et view til et andet, hvorved de samme data vises i forskellige diagramtyper.

3.2.2 Arkitektur

GUI'et skal altså opbygges af flere komponenter – et CRC view, et PVM view og et ClassDiagram view. De tre diagramtyper samles i et projekt objekt, for at opbygge en logisk gruppering af de tre diagrammer. Derudover skal der laves en Windows Forms applikation, der kan rumme GUI'ets elementer.

Windows applikationens opgave skal blot være at stille menuer og ikoner til rådighed og håndtere kliks på disse samt at holde styr på projekt-objekter. Projektet skal initialisere de tre diagramtyper. Projektet såvel som de tre diagrammer skal have kendskab til klasserne i produktmodellen. Desuden skal projektet kunne gemme et projekt som en projektfil og indlæse projektfilen igen.

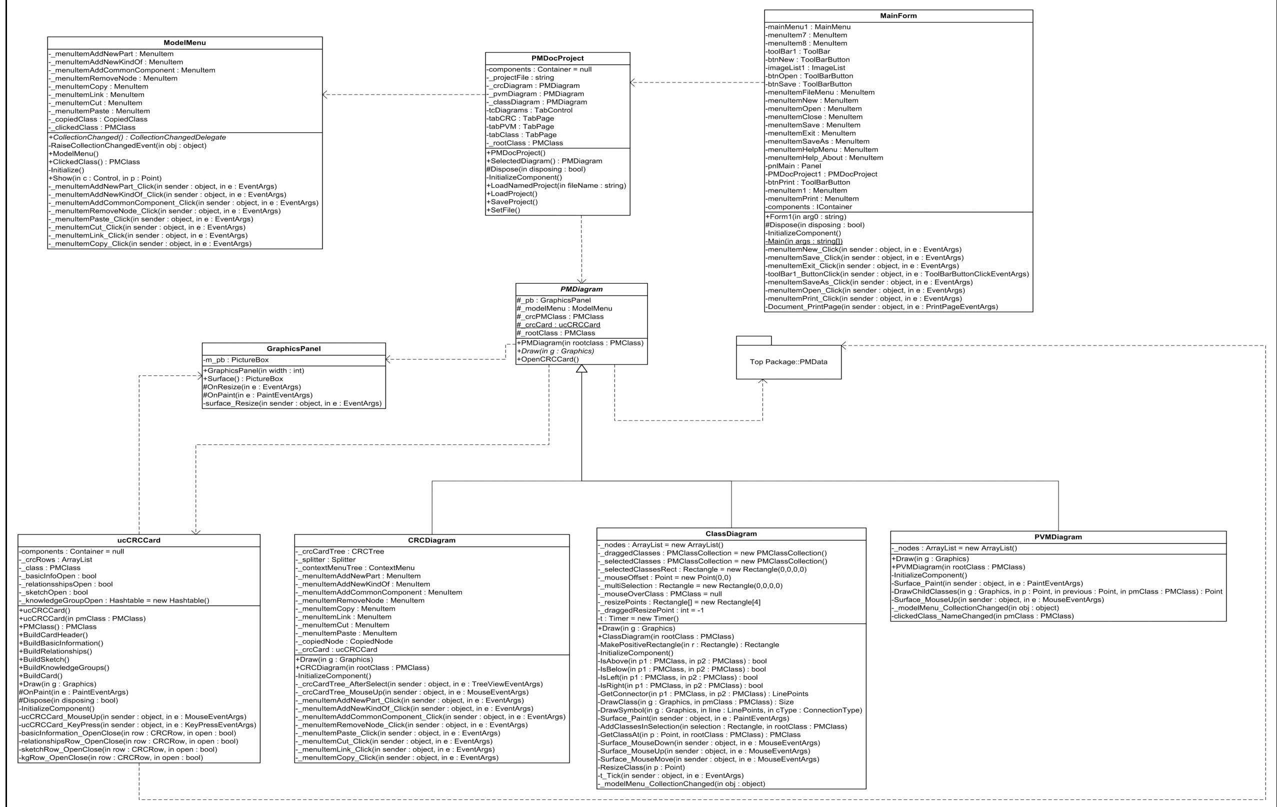


Figur 3.2.2.1 – Udkast til UML Diagram for præsentationskomponenten PMDocTool

UML diagrammet i Figur 3.2.2.1 viser et udkast til opbygningen af dokumentationsværktøjet. De tre diagramtyper har dog en del til fælles. F.eks. skal produktmodellen være den samme i alle tre, alle skal have en context menu og alle skal have en Draw metode, der tegner diagrammet. For at undgå runtime-fejl skal det ikke være muligt at instantiere et diagram uden at angive en indgang til en produktmodel – en rootklasse. Der indføres derfor den abstrakte klasse PMDiagram, der indeholder nogle attributter og metoder, der skal være fælles for de tre diagramtyper samt en constructor, der ikke giver risiko for at de nedarvende klasser ikke angiver en rootklasse. De tre diagrammer skal nedarve fra den abstrakte PMDiagram, som nedarver fra UserControl.

Alle diagramtyperne skal kunne åbne et CRC kort. I CRC-viewet skal CRC kortet vises i hovedvinduets højre side, når der klikkes med musen i træet. I de andre views skal CRC kortet vises

i et nyt vindue når der dobbeltklikkes på en klasse i diagrammet. CRC kortet skal derfor være en selvstændig komponent, der kan bruges af alle views. Denne skal laves som en UserControl.



Figur 3.2.2.2 – UML Diagram for præsentationskomponenten PMDocTool

Diagrammerne skal tegnes på en PictureBox. PictureBox giver ikke umiddelbart mulighed for at scrolle. Derfor tegnes alle diagrammer på en PictureBox, der ligger oven på et Panel. GraphicsPanel stiller dette til rådighed for alle diagrammer.

Desværre fører den strenge opdeling af userinterface (UI) og datamodellen til en høj kobling idet UI'ets opgave netop er at præsentere data fra datamodellen for brugeren. Især metoderne `Draw(g)`, der skal tegne data fra datamodellen skal kende alle attributter på produktmodellens klasser. Alternativt kunne man lægge forskellige tegne metoder i PMClass. En metode DrawAsPVMNode på PMClass-objektet kunne f.eks. tegne klassen som en Node i PVM diagrammet. Dette ville give lavere kobling, men føre til en mindre streng opdeling af UI og datamodel.

I denne løsning er den høje kobling og dermed ingen UI-metoder i PMClass benyttet. PMClass er dog løst fra viewet som det kendes fra design pattern'et Model-View-Controller.

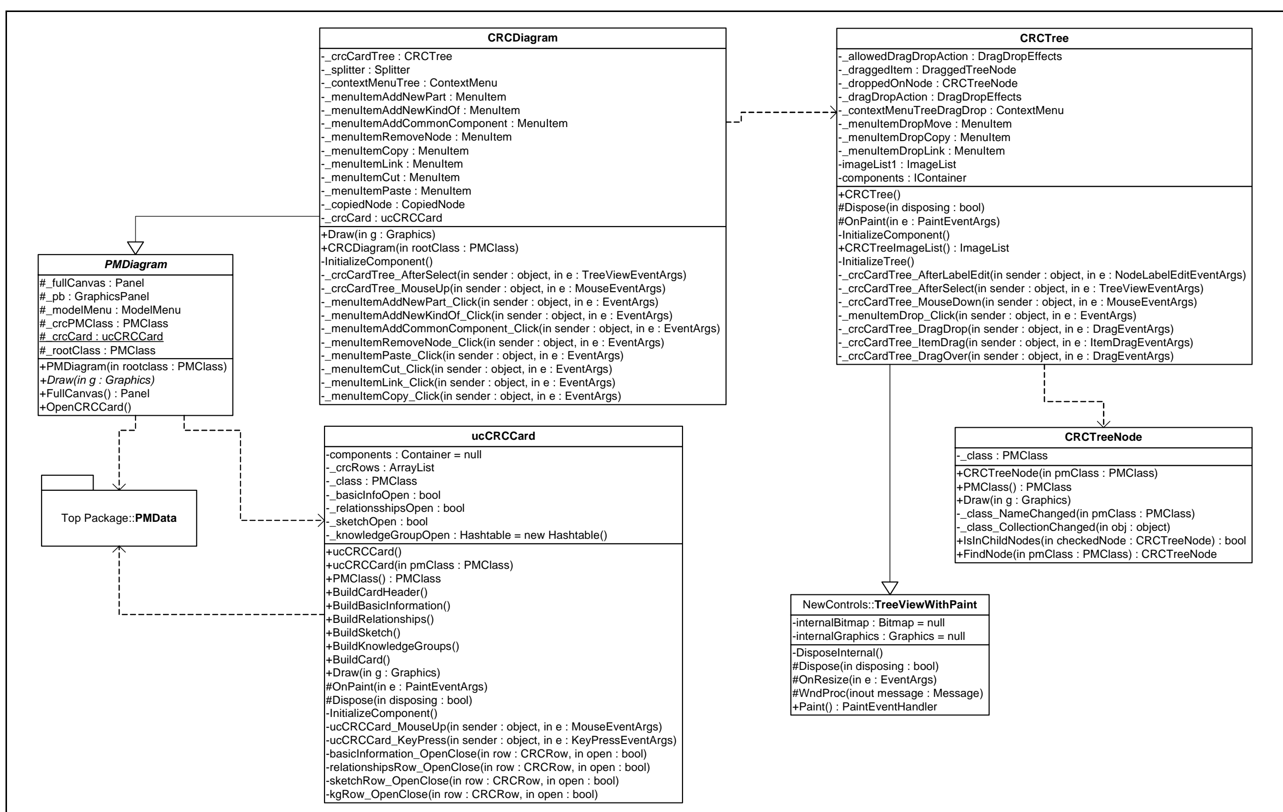
I de følgende afsnit beskrives CRCDiagram, ucCRCCard, PVMDiagrams og ClassDiagram arkitektur og design.

CRCDiagram

Som det ses i designet i Figur 3.2.1.1 adskiller CRC Viewet sig fra de to andre views på to områder. For det første har det ikke kun et område, hvorpå der tegnes men derimod to områder – en træstruktur og et område hvorpå CRC kortet tegnes. De andre views har blot et område hvorpå diagrammet tegnes. For det andet skal CRC kortet i de to andre views åbnes i et separat dialogvindue – i CRC view vises CRC kortet i hovedområdet når der klikkes på et element i træstrukturen.

Til træstrukturen skal der laves en nedarvet version af System.Windows.Forms.TreeView – CRCTree og træets nodes skal bestå af en nedarvet version af System.Windows.Forms.TreeNode – CRCTreeNode.

CRCTreeNode udvider TreeNode med en enkelt attribut, der giver mulighed for at tilknytte en PMClass til en TreeNode. Desuden tilføjes funktioner til søgning efter en speciel TreeNode udfra den tilknyttede PMClass ligesom funktioner til at genoptegne en speciel gren hvis der tilføjes ny klasser som aggregerer eller generaliserer en klasse.



Figur 3.2.2.4 – UML for CRCDiagram

CRCTree udvider blot TreeView med metoder, der skal muliggøre ændring af en PMClass' navn ved at ændre navnet på en node i træet samt metoder, der giver mulighed for drag and drop. Drag and drop i træet skal ændre data i datamodellen idet klasser skal kunne kopieres, flyttes og linkes.

Det skal også være muligt at kopiere, flytte og linke klasser via en context menu, der derudover skal kunne oprette nye klasser og fjerne klasser.

Da jeg ikke er tilfreds med den mulighed TreeView stiller til rådighed for at tegne symboler ud for TreeNodes i TreeView, benyttes en nedarvet version af TreeView. Normalt har TreeView ikke en OnPaint metode, da TreeView iflg. [Young J. (2003)] i virkeligheden blot er en .NET-wrapper til Windows kontrollen af samme navn og denne ikke stiller OnPaint til rådighed. TreeViewWithPaint er skrevet af J. Young og hentet fra codeproject.com.

ucCRCCard

Denne klasse skal bruges til at vise en klasses CRC kort og give mulighed for redigere attributter i klassen. ucCRCCard skal laves som en UserControl. Kontrollen skal tegne et CRC kort som det er afbilledet på Figur 1.2.1.2. Der skal være mulighed for at ”åbne” og ”lukke” dele af kortet. Dermed menes, at ved klik på en gruppes overskrift, skal en gruppe af felter synliggøres. Klikkes igen på samme overskrift, skal gruppen gøres usynlig.

ucCRCCard er opbygget som et view, der viser en model – iflg. princippet fra design pattern’et Model-View-Controller (MVC). Dog benyttes kun to af delene i dette pattern – model og view. Brugen heraf indebærer at modellen (PMData.PMClass) er afkoblet fra viewet idet viewet på intet tidspunkt referenceres i PMClass. Omvendt referenceres PMClass adskillige gangei viewet.

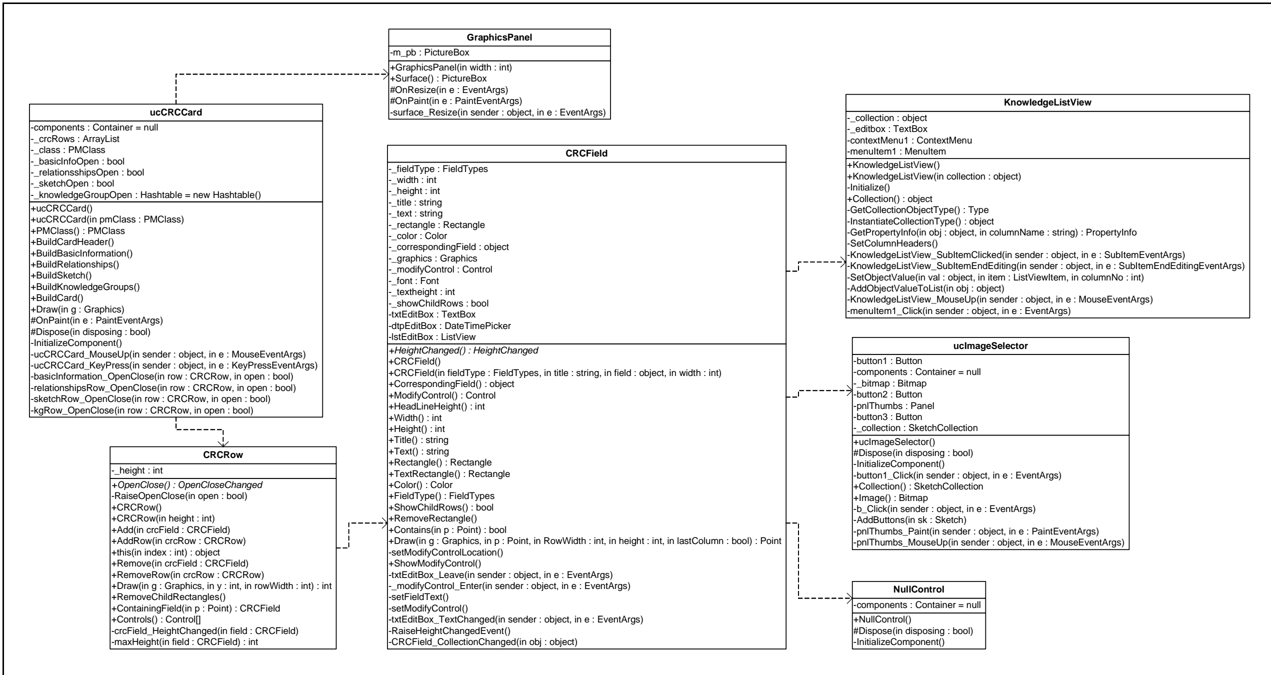
CRC kortet kan opdeles logisk i tre dele – CRC kort indeholder et arbitrært antal rækker, der hver især indeholder et arbitrært antal felter. Dette realiseres ved at indføre klasserne CRCRow og CRCField.

Tegning af CRC kortet skal ske oven på et GraphicsPanel og skal gøres manuelt af kontrollen ved at override kontrollens OnPaint metode.

Redigeringen af felter skal ske ved at lægge forskellige typer af kontroller oven på ucCRCCard. Skal en streng redigeres benyttes et TextField. Til redigering af en dato benyttes DateTimePicker. Bool kan redigeres med en CheckBox. Disse kontroller findes allerede i Windows.Forms.

Nogle felter skal slet ikke kunne redigeres. For at opretholde en fælles metode til alle felter skal der alligevel tilknyttes en kontrol til denne type felter. Dertil laves en kontrol, der intet kan – NullControl.

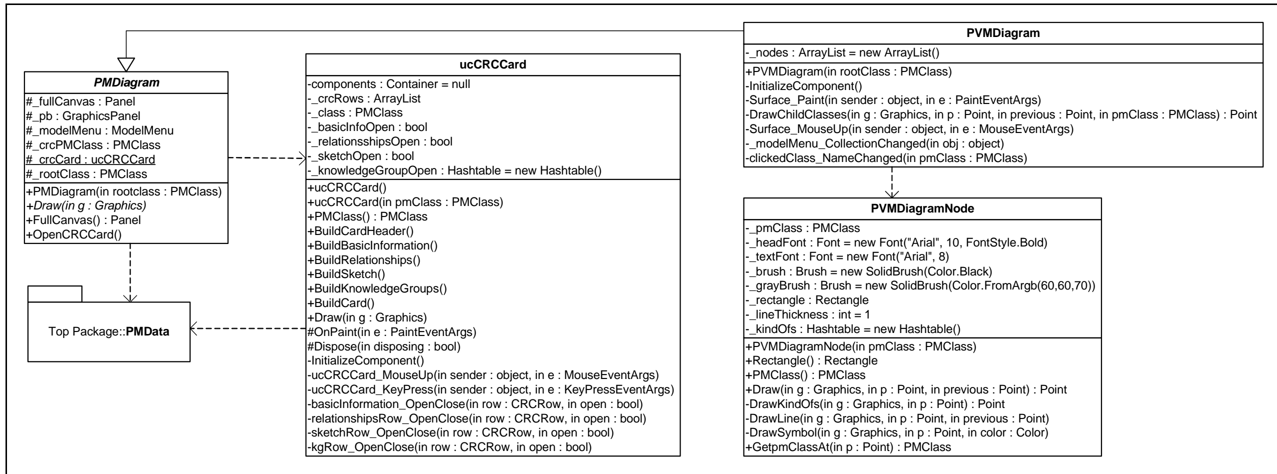
Andre felter er ikke mulige at redigere med de kontroller, der allerede findes i frameworket. Til disse felter skal der opbygges særlige kontroller. Der er her tale om Attributes, Constraints og Methods (KnowledgeListView) samt Sketches (ucImageSelector) i CRC kortet. I afsnit 4 beskrives implementeringen af disse kontroller.



Figur 3.2.2.4 – UML for ucCRCCard

PVMDiagram

PVMDiagram er det simpleste af alle diagrammerne. Det skal blot kunne vise træstrukturen og give mulighed for at redigere i den idet klasser skal kunne tilføjes, fjernes, kopieres, flyttes og linkes. PVMDiagram skal derfor være en simpel implementering af PMDiagram.

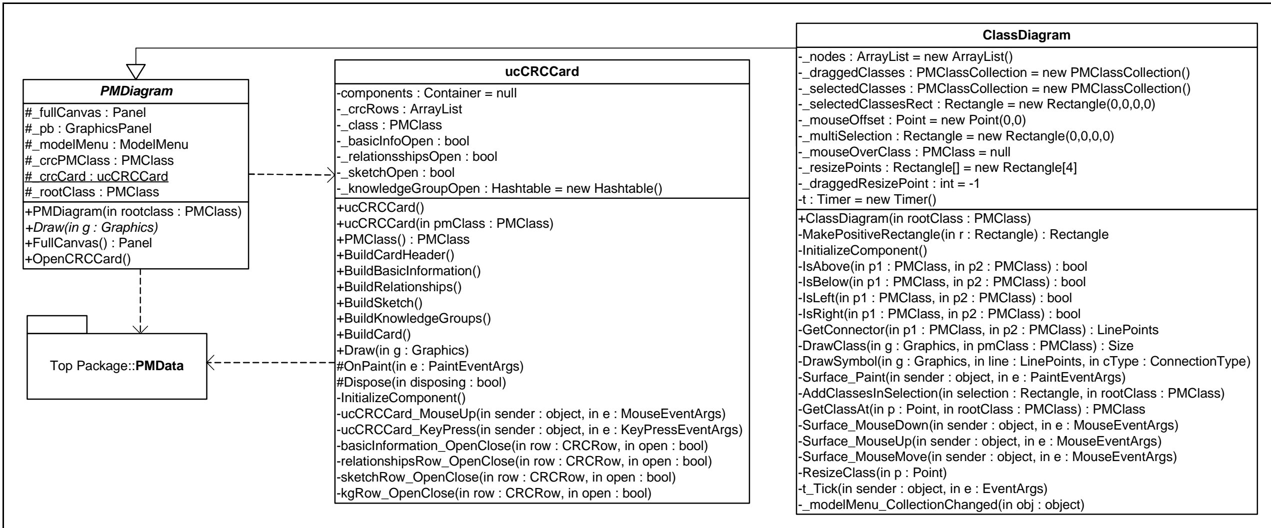


Figur 3.2.2.5 – UML for PVMDiagram

ClassDiagram

ClassDiagram er simpelt at tegne, men brugeren skal have mulighed for at flytte rundt på klasserne ligesom det skal være muligt at ændre størrelsen. Der skal desuden være mulighed for at redigere idet klasser skal kunne tilføjes, fjernes, kopieres, flyttes og linkes.

ClassDiagrams arkitektur skal ligesom PVMDiagram være en simpel implementering af PMDiagram.



Figur 3.2.2.6 – UML for ClassDiagram

KAPITEL 4

Implementering

I de følgende afsnit beskrives detaljer om implementeringen af klasserne PMData og PMDocTool. Enkelte specielt interessante metoders implementering vil blive beskrevet vha. kodeeksempler.

4.1 PMData

PMData er et class library (DLL fil) indeholdende de klasser og interfaces, der er nødvendige i datamodellen. Den består af følgende klasser og interfaces:

PMClass og PMClassCollection

PMClass repræsenterer en klasse i produktmodellen. Den indeholder alle attributter som produktmodellens klasser har. Desuden er der properties på alle attributterne, så disse kan tilgås udefra.

De to Collections KindOfs og ChildClasses svarer til generaliseringer og aggregeringer. Disse collections vil til enhver tid indeholde de klasser, der i træstrukturen ligger under klassen. Omvendt er der ingen relation til klasser, der ligger over klassen i træstrukturen (Superklasser og Superparts). Det er derfor nødvendigt at vedligeholde denne information. Metoderne `RefreshSuperParts` og `ClearSuperParts` muliggør opdatering af Superklasser og Superparts. De fungerer ved tree-traversing.

På PMClass ligger en observer, der informeres når ChildClasses eller KindOfs ændres. Metoden `RefreshCollectionChangedHandler` sørger for at alle ChildClasses og KindOfs informerer observeren om ændringer i deres collections. Objekter, der abonnerer på denne observer vil blive informeret om

ændringer i produktmodellen. Metoden fungerer vha. tree-traversing. Tilsvarende findes en anden observer, der informerer hvis navnet på en klasse i produktmodellen ændres.

Til sidst findes der på PMClass objektet metoder, der giver mulighed for at kopiere, linke og flytte nodes fra én node til en anden. Med link menes at der i en anden klasses collections oprettes en reference til den linkede klasse. På denne måde kan en klasse genbruges flere steder i produktmodellen. Ændringer det ene sted er samtidig ændret det andet sted. Ved kopiering forstås at der oprettes en kopi af en klasse. Kopien lægges i den klasses collections, der kopieres til. Ændringer i original eller kopi medfører ikke ændringer i modstykket.

PMClassCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde PMClass objekter. Tilføjes eller slettes klasser fra PMClassCollections InnerList, fyres en event af. Til denne event er PMClass' observer CollectionChanged hooket op. Ændres en PMClassCollection i grenene på et træ, vil denne event rejse op gennem træet til rodklassens CollectionChanged-observer.

PMClassCollection benyttes i PMClass til at lagre ChildClasses og KindOfs – dvs. aggregeringer og generaliseringer.

Sketch og SketchCollection

Sketch repræsenterer et billede af f.eks. et produkt, en produktvariant eller en komponent. Sketch objektet indeholder et Image-objekt (en bitmap) og et navn på billedet.

SketchCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde Sketch objekter.

KnowledgeGroup og KnowledgeGroupCollection

KnowledgeGroup er en nyere opfindelse, der oprindeligt ikke var tiltænkt som en del af opgaven. Pga. nye krav fra IPL blev denne klasse tilføjet som en gruppering af produktviden. Produktviden er Attributes, Constraints og Methods. Disse lå før grupperingen direkte i PMClass. KnowledgeGroup er altså blot en gruppering af AttributeCollections, ConstraintCollections og MethodCollections og indeholder derfor udover et navn også disse tre attributter.

KnowledGroupCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde KnowledgeGroup objekter. Collectionens Add metode tester om en nyligt added KnowledgeGroup har et navn. Har den ikke det, benyttes metoden `FirstAvailableName` til at finde et navn til gruppen. Et automatisk genereret navn vil aldrig være magen til et andet navn i gruppen. Navngivningen foregår efter princippet "product knowledge " + første fri heltal mellem 1 og ∞.

Attribute og AttributeCollection

Attribute repræsenterer i produktmodellen klasses attribut. En attribut er opbygget af et navn, mulige værdier, en type og en enhed.

Hver Property – Name, Values, Unit og ValueType har attributten Width(int). Tallet angiver i pixel hvor bred en spalte der er brug for for at kunne redigere i den. Attributten bruges af UI'et til at sætte bredden på en spalte i et ListView. Dette beskrives senere i dette afsnit.

AttributeCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde Attribute objekter.

Constraint og ConstraintCollection

Constraint repræsenterer i produktmodellen én af en klasses constraints. En Constraint er opbygget af et navn og en beskrivelse i form af tekst. I en senere udgave af Constraint skal der refereres direkte til involverede operander i stedet for brugen af en tekst.

Hver Property – Name og Description har ligesom i Attribute attributten Width(int).

ConstraintCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde Constraint objekter.

Method og MethodCollection

Method repræsenterer i produktmodellen klasses metode. En metode er opbygget af et navn og en beskrivelse i form af tekst.

Hver Property – Name og Description har ligesom i Attribute og Constraint attributten Width(int)

MethodCollection er en speciel collection – nedarvet fra CollectionBase. Denne collection kan udelukkende indeholde Method objekter.

StringWrapper, BoolWrapper og DateTimeWrapper

De tre wrapperklasser pakker blot typer med ValueType semantik ind, så de originale værdier i stedet for kopier kan refereres fra andre klasser. StringWrapper har yderligere en event ValueChanged, der fyres af når den indpakkede streng ændrer værdi.

Width

Width er en Custom Attribute og nedarver fra System.Attribute. Width benyttes til at angive bredden på en kolonne i f.eks. et ListView. Brugen af Width vises i beskrivelsen af KnowledgeListView klassen i brugerinterfacet.

4.1 PMDocTool

PMDocTool er en Windows .NET Application (EXE fil), der indeholder de klasser, der stiller den grafiske brugeroverflade til rådighed, opretter et projekt og laver tre views, der viser projektets produktmodel.

Applikationen består af en lang række klasser. Efterfølgende beskrives alle klasser. Implementeringen af en enkelt klasser er speciel og derfor særligt interessant. Denne klasse vil blive særligt fremhævet og beskrevet vha. kodeeksempler.

MainForm

Klassen nedarver fra System.Windows.Forms.Form og repræsenterer Windows Forms Applikationen. Når den instantieres kaldes metoden `InitializeComponent`, der initialiserer de statiske UI-elementer som Menuer og Toolbar. Desuden instantieres et PMDocProject – et produktmodel dokumentations projekt. Instantieres klassen med et argument (string) forskellig fra null, loades en fil med et serialiseret PMDocProject.

MainForm instantieres af den statiske metode `Main`, som også ligger på MainForm klassen. Denne metode er indgangspunkt til applikationen.

Klassen indeholder desuden en række event handlers, der håndterer klik på menupunkter og toolbar-ikoner.

PMDocProject

Klassen nedarver fra System.Windows.Forms.UserControl og instantieres af MainForm. Den repræsenterer et Produktmodel dokumentations projekt – dvs. en produktmodellen og de tre views, der kan vise produktmodellens klasser. Når den instantieres, initialiseres de tre faneblade og de tre views, der skal ligge på hver sit faneblad. Klassen kender kun én del af produktmodellen – rodklassen, der er ”indgang” til produktmodellen, der eksisterer som en træstruktur. Alle klasser i produktmodellen kan nås vha. tree-traversing.

Klassen indeholder metoder til håndtering af produktmodel filer

Desuden indeholder klassen en Property - `SelectedDiagram`, der returnerer det aktuelt valgte diagram. Denne benyttes f.eks. af Print metoden i MainForm.

PMDiagram

PMDiagram er en abstrakt klasse, der repræsenterer et view til visning af en produktmodel. Den nedarver fra System.Windows.Forms.UserControl og indeholder attributter og metoder, der med fornuft kan genbruges i alle implementeringer af views. Det er meningen, at PMDiagram i senere

versioner skal overtage endnu flere funktioner fra views'ne end den på nuværende tidspunkt gør. Lige nu begrænser den muligheden for instantiering af views, der nedarver fra den til at disse kun kan instantieres med en henvisning til en produktmodel i argumentet. Dette gøres for at undgå runtime exceptions under kørslen. Henvisningen er blot en reference til modellens rodklasse. Denne gemmes ved instantiering i en membervariable i PMDiagram.

PMDiagram stiller også muligheden for at åbne et CRC kort i et nyt vindue til rådighed vha. metoden `OpenCRCCard` Sættes inden kaldet til denne metode `_crcPMClass` til den ønskede PMClass, åbnes denne i et nyt vindue ved kald til metoden. Det åbnede vindue indeholder til enhver tid den samme instans af ucCRCCard. Dermed bevares dennes interne tilstand, hvilket muliggør at kortet, der vises altid har de samme grupper åbne som det sidste kort, der blev set på. Udelukkende de data kortet viser er forskellige ved åbning af de forskellige klassers CRC kort.

Til sidst har alle views brug for et område, der kan tegnes på – et GraphicsPanel, som beskrives senere.

CRCDiagram

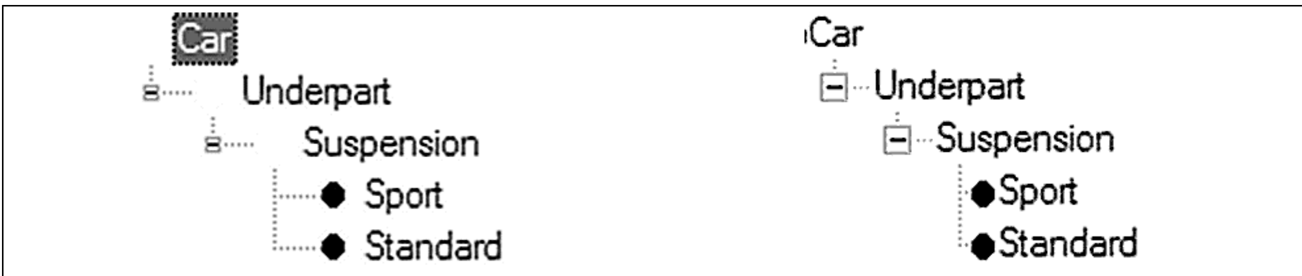
CRCDiagram er det første view der blev lavet. Klassen nedarver fra PMDiagram. Udover et område hvorpå der tegnes indeholder dette view i venstre side af billedet en træstruktur, der viser alle CRC-kort i produktmodellen. I højre side tegnes et CRC kort. Klikkes i træet på et CRC kort, vises dette i højre side.

Klassen er implementeret inden der blev gjort tanker om contextmenuen ModelMenu. Af denne grund indeholder klassen sin egen contextmenu. Denne bør i fremtidige versioner flyttes ud af klassen.

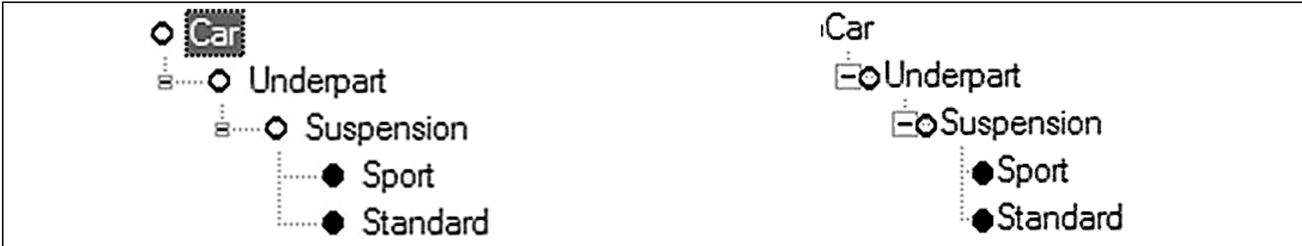
CRCTree

CRCTree nedarver fra NewControls.TreeViewWithPaint, der igen nedarver fra System.Windows.Forms.TreeView. CRCTree indeholder eventhandling til drag and drop. TreeViews manglende OnPaint kald er genereret af TreeViewWithPaint. I OnPaint kaldes Draw metoden i TreeViewets rodnode.

Muligheden for brug af ikoner i et TreeView var ikke tilfredsstillende idet det under implementeringen af træet endnu ikke var fastlagt om både dele og varianter skulle have symboler. Senere blev det besluttet at markere dele med en åben cirkel og varianter med en lukket.



Figur 4.1.1 – Dele har ingen symboler imens varianter har symboler. Til venstre ses standard TreeView. Til højre ses det samme tegnet med TreeViewWithPaint. Symbolerne er her placeret manuelt, hvorved "hullerne" er væk.



Figur 4.1.2 – Både dele varianter har nu symboler. Til venstre ses standard TreeView. Til højre ses det samme tegnet med TreeViewWithPaint. Her er forskellen mindre synlig.

Derfor ville jeg selv tegne symboler ovenpå TreeView'et ved siden af hver TreeNode. TreeView besidder ganske vidst en OnPaint metode, der kan overrides da den nedarver fra System.Windows.Forms.Control, men forsøg på, inden i denne metode at tegne på kontrollen mislykkedes. Dette skyldes iflg. [Young J. (2003)], at TreeView ligesom ListView blot er en wrapper omkring ComCtl kontrollerne af samme navn og disse ikke normalt invoker eventen Paint. Løsningen er, at generere et Graphics-objekt baseret på en bitmap og tegne ComCtl kontrollen ovenpå. Derefter genereres det manglende kald til OnPaint.

CRCTreeNode

CRCTreeNode nedarver fra System.Windows.Forms.TreeNode og udvider blot denne med en enkelt attribut – en PMClass. Noden hooker sig op til klassens NameChanged og CollectionChanged events. Når disse affyres, ændres nodens tekst sig hhv. cleares childnodes og fyldes op igen. På denne måde sikres at træet under den aktuelle node altid svarer til klassen.

Draw metoden tegner et symbol lige til venstre for nodens tekst. Derefter kaldes Draw metoden på alle childnodes.

CRCTreeNode indeholder desuden en metode FindNode, der vha. tree traversal kan finde en CRCTreeNode med en bestemt PMClass.

ucCRCCard

ucCRCCard er en UserControl, der tegner et CRC kort og giver mulighed for at redigere i det.

Et CRC kort består af et kort, hvorpå der er en række felter. Felterne er opdelt i grupper og inden i grupperne i rækker som det ses i Figur 1.2.1.2. Jeg har valgt at nøjes med at opdele CRC kortet i rækker og felter. Rækker kan dog have underrækker og dermed fungere som grupper. Rækker og Felter er realiserede som objekterne CRCRow og CRCField, der kan tegnes ovenpå CRC kortet. Dermed er ansvaret for kort, rækker og felter fordelt for bedre overskuelighed og større samhørighed. ucCRCCard kan bygge CRC kortet op af CRCRow og CRCField objekter og kan holde styr på hvilke grupper der er åbne og lukkede. Derudover kan det registrere klik på musen og derved aktivere felter for indtastning.

CRCRow

CRCRow nedarver fra CollectionBase og er dermed en speciel collection. Der kan add'es CRCField og CRCRow objekter i dens liste. Ved at lægge CRCRow objekter i listen opnås muligheden for at gruppere rækker af felter. Metoden Draw kalder Draw metoden på CRCField og CRCRow objekter i listen. CRCRow holder bla. styr på hvor høj en række er – dvs. hvor højt det højeste felt er. Rækken er hooket op til underliggende fields HeightChanged-event. Fyres denne event af på et af felterne i collectionen, ændres højden på rækken til den nødvendige højde.

CRCField

CRCField repræsenterer ét felt på CRC kortet. Klassen indeholder metoder til tegning af feltet ligesom til aktivering af redigering af feltets indhold. Kræver f.eks. en ændring af indhold, at højden på feltet og dermed på rækken, hvori feltet befinder sig ændres, fyres en HeightChanged event af. Redigering foregår ved at en egnet kontrol (f.eks. TextBox, DateTimePicker, KnowledgeListView osv.) lægges ovenpå feltet når der klikkes på feltet. Når kontrollen forlades, gemmes ændringen i det objekt feltet repræsenterer.

NullControl

NullControl er en UserControl, der intet kan. Den er udelukkende lavet fordi alle CRC felter *skal* have en kontrol – også de, der ikke skal kunne redigeres. Sådanne felter får en NullControl.

KnowledgeListView

KnowledgeListView nedarver fra ListViewEx [mav.northwind (2004)], som igen nedarver fra ListView. ListViewEx udvider ListView med funktioner, der muliggør In-Place Editing i et ListView. Dette fungerer ved at lægge en kontrol – f.eks. en TextBox ovenpå ListView'et på samme måde som ucCRCCard gør det.

KnowledgeListView fungerer generisk med forskellige Collections. Titlen på hver kolonne i ListView'et bestemmes vha. Reflection på følgende måde:

Først skal et egnet ListView opsættes med de rigtige kolonner. Hertil kaldes i konstruktøren metoden `SetColumnHeaders`. Her skal først findes typen på objekter i en specialiseret collection. Dette gøres i metoden `GetCollectionObjectType`.

```
        private Type GetCollectionObjectType()
        {
            Type t = _collection.GetType();
            MethodInfo mI = t.GetMethod("Add");
            ParameterInfo[] pI = mI.GetParameters();
            return pI[0].ParameterType;
        }
```

En collection vil ikke have nogen mening hvis det ikke er muligt at add’e objekter til dens liste. Dette sker i metoden Add på collectionen. Som argument tager en specialiseret collection som regel en enkelt parameter – et objekt af den type, der skal ligge i den specialiserede collection. `GetCollectionObjectType` benytter klasser fra namespacet System.Reflection til at finde typen på en collection. Derefter findes metoden Add og dennes parametre gemmes i et ParameterInfo-Array. Objekterne i den specialiserede collection har alle typen på den første parameter ParameterInfo-Array’et (pi[0]). Denne type returneres.

Nu er det muligt at lave det korrekte antal kolonner og give dem navne. Kolonnerne vil svare til de properties, der findes på et objekt af den opløste type.

```
    PropertyInfo[] properties = t.GetProperties();
```

PropertyInfo-Arrayet indeholder nu PropertyInfo på alle objektets properties. Kolonnerne kan nu addes til ListView’et vha. en for-løkke:

```
    for (int i = 0; i < properties.Length; i++)
        this.Columns.Add(properties[i].Name);
```

Men hvor brede skal kolonnerne være? Et Navn er f.eks. givetvist en del kortere end en beskrivelse. Her kommer Custom-attributten Width ind i billedet. Width attributten er lagt på alle de objekter, der benyttes i de specialiserede collections, som i dette program vises vha. KnowledgeListView – f.eks. `Method.Name` og `Method.Description` hvor bredden skal være 110 hhv. 355 pixel.

```
    [Width(110)]
    public string Name
    {
        get {return _name;}
        set {_name = value;}
    }

    [Width(355)]
    public string Description
    {
        get {return _description;}
        set {_description = value;}
    }
```

Nu kan koden i for-løkken udvides så kolonnernes bredde angives. GetCustomAttribute kan kaste en Exception hvis en Property ikke har attributten Width. Derfor er der tilføjet exception handling –

dog blot for at undgå at programmet fejler. Hvis Width attributten ikke er sat defaulter bredden til 80 pixel.

```
for (int i = 0; i < properties.Length; i++)
{
    int width = 80;
    try
    {
        object[] CustomAttributes = properties[i].GetCustomAttributes(typeof(PMData.Width), false);

        if (CustomAttributes.Length > 0)
            width = ((PMData.Width)CustomAttributes[0]).Value;
    }
    catch{}
    this.Columns.Add(properties[i].Name, width, HorizontalAlignment.Left);
}
```

Elementer i KnowledgeListView kan redigeres ligesom der kan tilføjes nye elementer til en collection vha. denne. ListViewEx har to eventhandlere, der benyttes her:

SubItemClicked - kaldes når der klikkes på et item eller et subitem. Heri startes et felts redigerings-tilstand og feltets værdi kan redigeres.

SubItemEndEditing – kaldes når et felt, der editeres forlades og editeringen dermed afsluttes. Er der tale om et nyt objekt, der skal addes til den specialiserede collection, skal et objekt af den rigtigt type først instantieres.

```
    private object InstantiateCollectionType()
    {
        Type t = GetCollectionObjectType();
        ConstructorInfo ci = t.GetConstructor(new Type[0]);
        return ci.Invoke(new object[0]);
    }
```

Dette gøres igen vha. Reflection. Først bestemmes typen på objektet. Derefter findes objektets default-constructor og et objekt instantieres og returneres. Objektet gemmes i et nyt ListViewItem's Tag og dette addes til ListView'et.

Derefter sættes værdien på den første Property i objektet til det indtastede med `SetObjectValue(e.DisplayText, e.Item, 0)`:

```
    private void SetObjectValue(object val, ListViewItem item, int columnNo)
    {
        object obj = item.Tag;
        PropertyInfo pi = GetPropertyInfo(obj, this.Columns[columnNo].Text);
        pi.SetValue(obj, val, null);
    }
```

Hermed findes den Property på objektet i Tag'en, der har samme navn som kolonnen der lige er indtastet en værdi i. Værdien af denne property sættes til det indtastede.

Ved eksisterende objekter kaldes blot `SetObjectValue(e.DisplayText, e.Item, e.SubItem)`.

ucImageSelector

Klassen nedarver fra System.Windows.Forms.UserControl og kan bruges til at vise og redigere en SketchCollection. Klikkes på et billede, åbnes dette i et nyt modal vindue – en PictureViewer. Ovenpå hvert billede tegnes en lille knap med et rødt kryds i. Trykkes på denne knap, bliver det underliggende billede slettet fra SketchCollection'en. Nye billeder kan tilføjes ved at trykke på knappen "Add".

PictureViewer

PictureViewer klassen nedarver fra System.Windows.Forms.Form. PictureViewer benyttes til at vise et Sketch-objekts bitmap-grafik i et separat vindue.

PVMDiagram

Dette view viser produktmodellen i et PVM Diagram. PVMDiagram nedarver fra PMDiagram. Implementeringen af Draw starter en traversering ned gennem produktmodel-træet og tegner alle klasser. Den egentlige tegning af klasserne foregår i PVMDiagramNode.Draw. Klikkes dobbelt på en klasse, åbnes klassen i et CRC kort så den kan redigeres. Højreklikkes på en klasse vises en contextmenu, der giver mulighed for redigering af produktmodellen.

PVMDiagramNode

PVMDiagramNode repræsenterer en klasse i PVM diagrammet. Denne klasse tegner klassen som den skal se ud i et PVM diagram samt linierne, der forbinder klasserne. Har klassen KindOf's, tegnes de også her. Desuden findes en metode GetpmClassAt(point), der returnerer den klasse, der ligger på et bestemt punkt. Denne metode fungerer vha. tree traversal og benyttes ved klik på en klasse i diagrammet.

ClassDiagram

Dette view viser produktmodellen i et klassediagram, der følger UML notationen. ClassDiagram nedarver fra PMDiagram. Implementeringen af Draw starter ligesom i PVMDiagram en traversering ned gennem produktmodel-træet og tegner alle klasser. Klikkes dobbelt på en klasse, åbnes klassen i et CRC kort så den kan redigeres. Højreklikkes på en klasse vises en contextmenu, der giver mulighed for redigering af produktmodellen.

Den egentlige udfordring i implementeringen af klassediagram viewet ligger i at brugeren skal kunne flytte rundt på klasserne i diagrammet – og der skal være mulighed for at ændre størrelsen (resize) på klasserne.

Tools

Klassen skal bruges til forskellige tools, der kan benyttes flere steder i programmet. På nuværende tidspunkt ligger der blot en metode, der kan generere thumbnails i en given størrelse.

GraphicsPanel

GraphicsPanel benyttes i alle views til at tegne ovenpå i stedet for blot at benytte en System.Windows.Forms.PictureBox. PictureBox understøtter ikke umiddelbart scrolling. Det gør derimod et Panel. Tegnes der derfor i en PictureBox anbragt ovenpå et Panel således at den fylder hele Panel'ets område, kan man ved at scrolle i Panel'et samtidig scrolle i PictureBox'en. GraphicsPanel nedarver fra Panel. Når den instantieres, lægges en PictureBox oven på den og base klassens muligheder for scrolling aktiveres.

ModelMenu

ModelMenu er den sidste klasse, der blev implementeret i prototypen og den er endnu kun delvist færdig. ModelMenu nedarver fra ContextMenu. Den kan betegnes som en specialiseret context-menu idet alle menu punkter er defineret inde i klassen og lægges på menuen når den instantieres. Menuen er beregnet til at redigere en produktmodel fra de views, der benytter den. Der er implementeret mulighed for at tilføje nye parts og kinds til modellen. Kopiering, linking, flytning og sletning er endnu ikke implementeret. ModelMenu benyttes af PVMDiagram og ClassDiagram. En færdig version af menuen skal desuden kunne bruges på CRCDiagram, der endnu har sin egen contextmenu.

KAPITEL 5

Test

Programmet består af to dele – en datamodel og et grafisk brugerinterface (GUI).

Datamodellen indeholder kun meget få funktioner. Disse funktioner kan testes vha. fornuftige Unit tests.

Størstedelen af programmet består af GUI'et og netop denne del er iflg. [Gerrard, P. (1997)] vanskelig at teste idet det ikke kan udelukkes at brugeren klikker på enhver pixel på skærmen og ikke følger en fastlagt procedure for brug af softwaren. GUI'et er eventdriven og ethvert klik på skærmen eller en indtastning kan medføre en event. Følges normale principper for tests, bør for ethvert view i enhver tilstand alle mulige events i enhver rækkefølge testes. Dette vil medføre uoverkommeligt store tests og evt. kræve udviklingen af specielle test-drivere.

I testen her udføres flere typer af tests.

Først undersøges om de funktionelle krav er opfyldt idet en produktmodel implementeres i programmet.

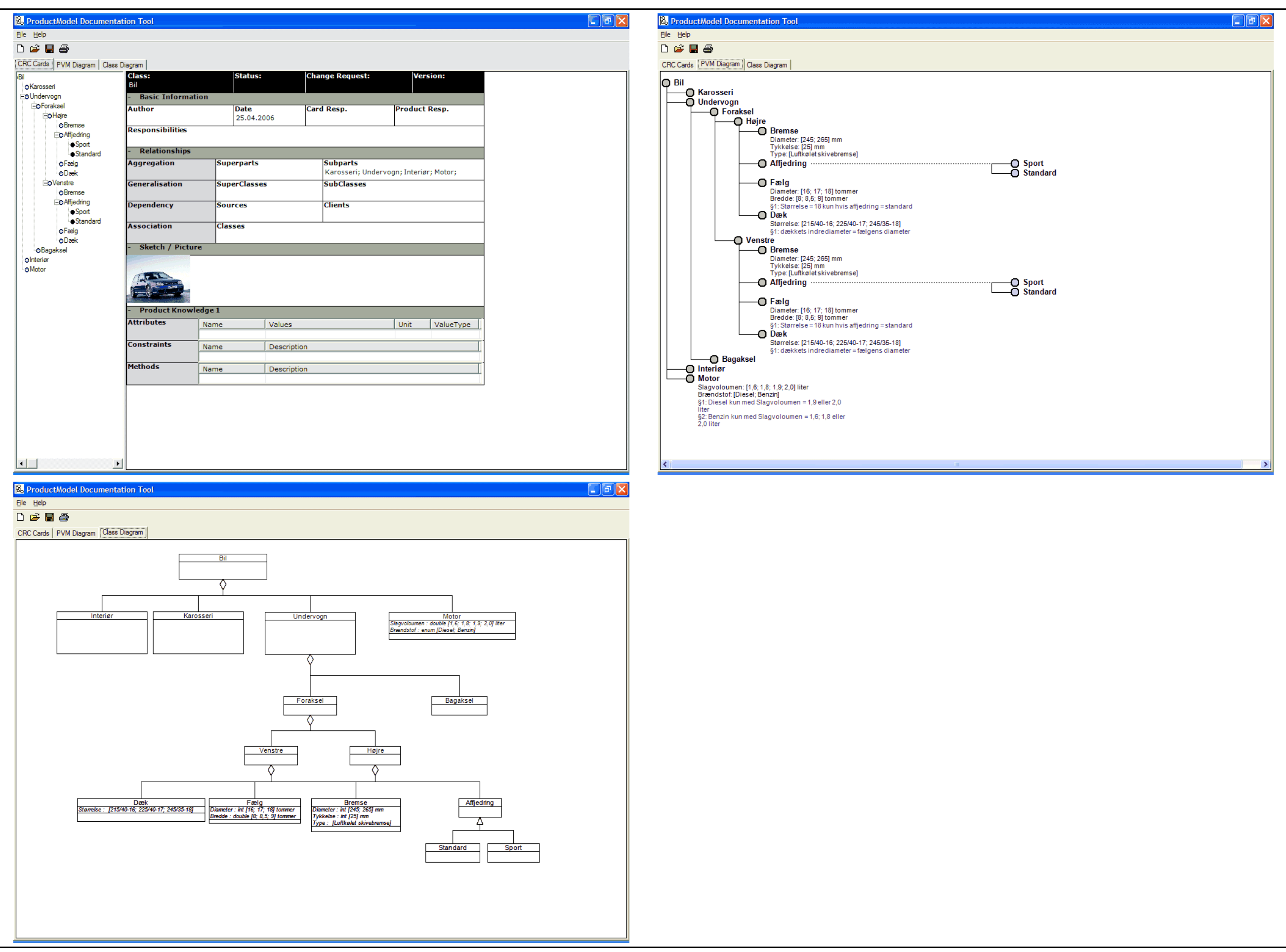
En blackbox test kan også udføres. Chancen for at finde fejl ved denne type test er imidlertid ikke så høj som ved analyse af programkoden idet der er mange muligheder for fejl, der evt. overses og dermed ikke testes. Derfor analyseres programkoden for hvor fejl kan opstå. Denne proces kan tage lang meget lang tid idet alle programlinier skal undersøges. Efter analysen skrives tests, der kan påvise fejlen og disse tests udføres. Der kan her være tale om automatiserede tests i form af små testprogrammer, der tester enkelte metoder eller manuelle tests.

Til sidst nævnes en enkelt metoder, der er testet vha. Unit test.

5.1 Test af funktionelle krav

I denne test implementeres en produktmodel af dele af en delmængde af en bil i programmet. Kan denne produktmodel implementeres, anses de funktionelle krav som værende opfyldt.

Produktmodellen oprettes idet flere kort oprettes i både CRC Card view, PVM Diagram view og Class Diagram view.



Figur 5.1.1 – Diagrammerne som de bør se ud. Placeringen og størrelsen på klasserne i klassediagrammet kan afvige men opbygningen bør være identisk.

- Når programmet er startet, ændres navnet på produktet i træet til "Bil."
- I træet tilføjes nu første del – karosseriet. I CRC kortet ændres navnet til "Karosseri".
- Der skiftes til PVM Diagram og en yderligere del tilføjes til "Bil". Der dobbeltklikkes på den ny del og navnet ændres i CRC kortet til "Undervogn".
- Der skiftes til Class Diagram. De to dele, der er oprettet flyttes manuelt med musen så begge kan ses.

- En yderligere del tilføjes til ”Bil” og flyttes. Der dobbeltklikkes på den ny del og navnet ændres i CRC kortet til ”Interiør”.
- Tilbage i CRC Cards vælges ”Undervogn”. I feltet Author indtastes et navn. I feltet Date vælges en anden dato end den, der er valgt. Felterne Card. Resp, Prod. Resp og Responsibilities udfyldes. Et billede indsættes i Sketch/Picture. Til sidst udfyldes én Attribute, en Constraint og én Method.
- Der skiftes til PVM Diagram og dobbeltklikkes på ”Undervogn”. Det undersøges, om de indtastede værdier ses i CRC kortet. Kortet lukkes.
- Der skiftes til Class Diagram og dobbeltklikkes på ”Undervogn”. Det undersøges, om de indtastede værdier ses i CRC kortet. Kortet lukkes.
- Tilbage i CRC Cards højreklikkes på ”Undervogn”. i contextmenuen vælges Copy. Der højreklikkes på ”Bil” og vælges Paste.
- Det undersøges om indholdet af kopien er identisk med indholdet af originalen.
- Navnet på kopien ændres til ”Motor”
- I CRC kortet for ”Motor” ændres felterne Author, Date, Card. Resp, Prod. Resp og Responsibilities.
- I CRC kortet for ”Undervogn” slettes billedet og den ene Attribute, Constraint og Method slettes hver især.
- Det undersøges at ”Undervogn” og ”Motor” nu er forskellige i alle ændrede felter.
- Til ”Undervogn” tilføjes en ny del og denne omdøbes til ”Foraksel”
- Vha. højre museknap dragges ”Foraksel” op på ”Undervogn”.
- I contextmenuen vælges ”Copy Here”
- Navnet på kopien ændres til “Bagaksel”
- Til ”Foraksel” tilføjes en ny del og denne omdøbes til ”Højre”
- Til ”Højre” tilføjes en ny del og denne omdøbes til ”Bremse”.
- ”Højre” dragges med venstre museknap op på ”Foraksel”. Samtidig holdes Alt på tastaturet nede når der droppes. Kopien omdøbes til ”Venstre”. Denne skal ligesom ”Højre” have en ”Bremse” under sig.
- Der højreklikkes på ”Højre” og tilføjes en ny del. Delen omdøbes til ”Affjedring”. ”Affjedring” skal kunne ses under både ”Højre” og ”Venstre”.
- Der højreklikkes på ”Affjedring” og tilføjes en ny variant vha. ”Add New KindOf”. Den ny variant omdøbes til ”Sport”.
- Vha. venstre museknap dragges ”Sport” op på ”Affjedring”. Samtidig holdes Ctrl på tastaturet nede når museknappen slippes. Kopien omdøbes til ”Standard”. Under både ”Højre” og ”Venstre” på ”Foraksel” skal ”Affjedring” have to varianter ved navn ”Sport” og ”Standard”.
- Der skiftes til PVM Diagram. Her tilføjes en ny del til ”Højre”. Navnet ændres i CRC kortet ved dobbeltklik på den ny del til ”Dæk”. ”Dæk” skal findes under både ”Højre” og ”Venstre” under ”Foraksel”.
- Der skiftes til Class Diagram. Alle klasser placeres ved at trække i dem med musen således at klassediagrammet ser fornuftigt ud.
- Her tilføjes en ny del til ”Venstre”. Navnet ændres i CRC kortet ved dobbeltklik på den ny del til ”Fælg”. ”Fælg” skal aggregere fra både ”Højre” og ”Venstre”.
- I CRC kortene for ”Bremse”, ”Dæk” og ”Fælg” indtastes forskellige attributter, constraints og methods.

- I alle CRC kortene gennemgås felterne ”Superparts”, ”Subparts”, ”SuperClasses” og ”SubClasses” i ”Relationships”. Det undersøges om de rigtige navne ses i felterne.
- I Class Diagram ændres størrelsen på nogle af klasserne ved at tage fat i hjørnerne med musen og trække.
- Produktmodellen gemmes og programmet lukkes og genstartes. Produktmodellen loades. Det undersøges om modellen stadig er som før.

De funktionelle krav er overholdt hvis alle disse punkter kan udføres. I Figur 5.1.1 ses hvordan diagrammerne bør se ud efter denne test.

5.2 Analyse af mulige fejl

Under programmeringen er der gennemført adskillige tests af forskellige funktioner. Disse tests har flere gange påvist fejl og disse er blevet rettet.

Da testeren i dette tilfælde selv har skrevet programkoden og har viden om hvilke fejl der er blevet påvist og rettet, vil den komplette programkode ikke blive analyseret for fejl.

Et punkt, der ganske vist er taget højde for i en del af programmet er aldrig blevet analyseret fuldt ud:

Den hierarkiske opbygning af produktmodellen kombineret med muligheden for at flytte og linke klasser introducerer risikoen for fejl idet en klasse kan flyttes ind i én af sine egne eller sine childrens child-collections. Når træstrukturen i CRC Cards eller PVM Diagrammet skal tegnes, vil klassen tegnes som sin egen child i en uendelighed. Dette vil givetvist føre til en form for overflow. En uendelig løkke vil også opstå i klassediagrammet, idet Draw metoden vil kalde sig selv igen og igen. Scenariet giver heller ingen mening i produktmodellen da en del ikke kan være en del af sig selv.

5.3 Test af analyserede fejl

Der er flere muligheder for at flytte en klasse ind i en collection, hvor den ikke bør ligge. Testen af problemet skal omfatte flytning vha. contextmenu'ens Cut/Copy Shortcut – Paste, ved brug af Drag and Drop med venstre museknap kombineret uden tastetryk samt kombineret med tryk på Alt og med højre museknap hvorefter Move here hhv. Link here vælges. Testen skal gennemføres idet der flyttes/linkes til en klassen selv, til én af klassens children, til andre referencer af klassen og til andre referencer af klassens children.

Resultat af test

Test	Resultat
Cut –> Paste - klassen til klassen selv	StackOverflowException
Copy Shortcut –> Paste – klassen til klassen selv	StackOverflowException
Cut –> Paste – klassen til en Child	StackOverflowException
Copy Shortcut –> Paste – klassen til en Child	StackOverflowException
Drag'n Drop venstre uden tastetryk – klassen til klassen selv	OK
Drag'n Drop venstre med tryk på Alt – klassen til klassen selv	OK
Drag'n Drop venstre uden tastetryk – klassen til en Child	OK
Drag'n Drop venstre med tryk på Alt – klassen til en Child	OK
Drag'n Drop højre Move here – klassen til klassen selv	OK *
Drag'n Drop højre Link here – klassen til klassen selv	OK *
Drag'n Drop højre Move here – klassen til en Child	OK *
Drag'n Drop højre Link here – klassen til en Child	OK *
Cut –> Paste - klassen til anden reference	StackOverflowException
Copy Shortcut –> Paste – klassen til anden reference	StackOverflowException
Cut –> Paste – klassen til Child af anden reference	StackOverflowException
Copy Shortcut –> Paste – klassen til Child af anden reference	StackOverflowException
Drag'n Drop venstre uden tastetryk – klassen til anden reference	StackOverflowException
Drag'n Drop venstre med tryk på Alt – klassen til anden reference	StackOverflowException
Drag'n Drop venstre uden tastetryk – klassen til Child af anden reference	StackOverflowException
Drag'n Drop venstre med tryk på Alt – klassen til Child af anden reference	StackOverflowException
Drag'n Drop højre Move here – klassen til anden reference	StackOverflowException
Drag'n Drop højre Link here – klassen til anden reference	StackOverflowException
Drag'n Drop højre Move here – klassen til Child af anden reference	StackOverflowException
Drag'n Drop højre Link here – klassen til Child af anden reference	StackOverflowException

*) Copy here er heller ikke mulig

5.4 Unit test

En enkelt metode er testet vha. unit tests. Til test af units har jeg skrevet et lille unit test framework. Frameworket består af et test program, der instantierer tests og udskriver resultaterne. Tests, der skal udføres af test programmet implementerer et lille Test-interface iTest, der udelukkende kræver implementeringen af metoden `Test()`, der skal returnere true hvis testen lykkedes og false hvis testen mislykkedes.

```
class Test
{
    [STAThread]
    static void Main(string[] args)
    {
        ArrayList TestArray = new ArrayList();
        bool Success = true;
        TestArray.Add(new TestKnowledgeGroup());
        foreach (iTest t in TestArray)
        {
            bool tSuccess = t.Test();
            Console.WriteLine("----------------------------------------------------------------");
            Console.WriteLine("Test Result : {0}", tSuccess ? "Success" : "Fail");
            Console.WriteLine("----------------------------------------------------------------");
            Success &= tSuccess;
        }
        Console.WriteLine();
        Console.WriteLine("----------------------------------------------------------------");
        Console.WriteLine("Total Test Result : {0}", Success ? "Success" : "Fail");
    }
}

interface iTest
{
    bool Test();
}
```

Figur 5.4.1 – Unit Test Frameworket

TestKnowledgeGroup

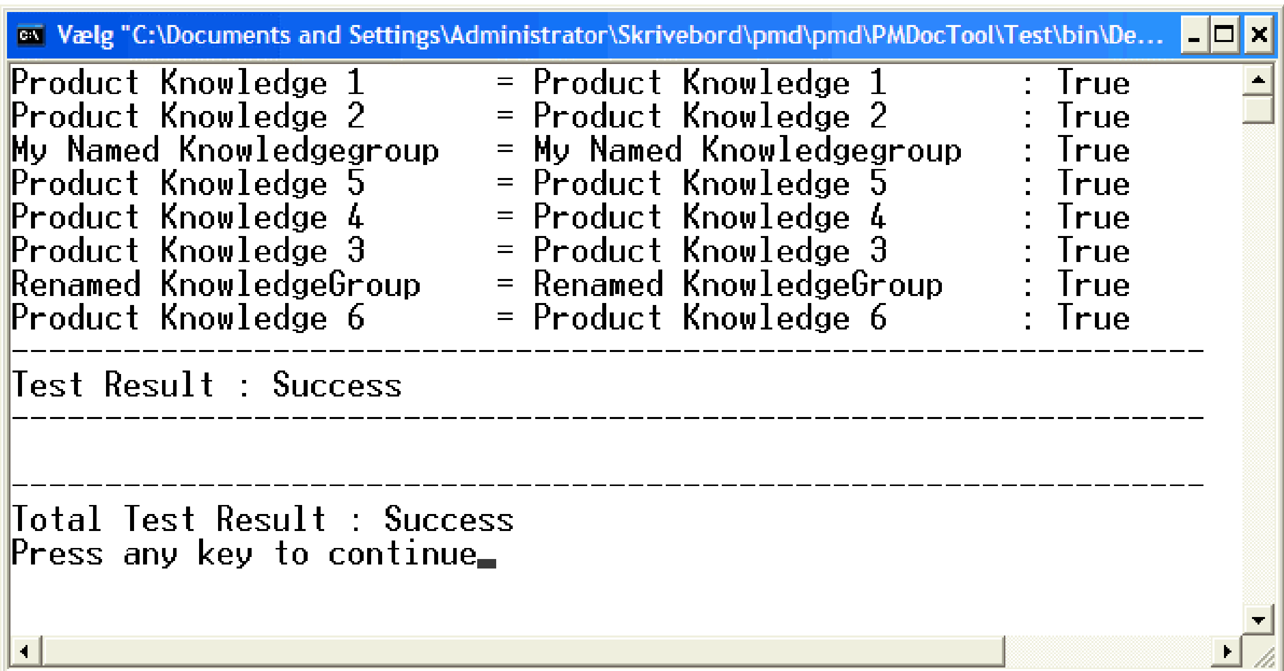
Et KnowledgeGroup objekt kan instantieres med og uden navn (`new KnowledgeGroup()` og `new KnowledgeGroup(string name)`) og addes til en PMClass' KnowledgeGroupCollection. Når en KnowledgeGroup uden navn addes til en KnowledgeGroupCollection, bliver denne navngivet så den får et unikt navn. Navnet er opbygget efter formen "Product Knowledge " + firstAvailableNumber, hvor firstAvailableNumber er det første hele tal, der ikke er brugt før i den samme KnowledgeGroupCollection. Hertil er der lavet metoden `FirstAvailableName()`, der benyttes når KnowledgeGroup uden navn addes til en KnowledgeGroupCollection. Metoden skal til enhver tid finde det første nummer, der ikke er benyttet i forvejen. Slettes en KnowledgeGroup med et navn med samme format eller renames en KnowledgeGroup med formatet til et andet navn, vil et nummer blive frit og kan benyttes senere.

Testen er skrevet så den i `SetupTest()` tilføjer KnowledgeGroups med og uden navn, sletter en KnowledgeGroup midt i KnowledgeGroupCollection'en og tilføjer yderligere KnowledgeGroups. Herved testes at nummereringen i navngivningen ikke overspringer "huller" i nummeringen, der opstår ved sletning af KnowledgeGroups. En KnowledgeGroup, der manuelt gives et navn, der overholder formateringen tilføjes til listen. Herved testes, at navngivningen også erkender denne type navne. Til sidst renames en KnowledgeGroup, så den ikke længere følger formateringen, hvorefter en ny KnowledgeGroup uden navn addes.

```
class TestKnowledgeGroup : iTest
{
    KnowledgeGroupCollection kgc;

    public TestKnowledgeGroup()
    {
        SetupTest();
    }

    private void SetupTest()
    {
        kgc = new KnowledgeGroupCollection();
        kgc.Add(new KnowledgeGroup());
        kgc.Add(new KnowledgeGroup());
        kgc.Add(new KnowledgeGroup());
        kgc.Add(new KnowledgeGroup("My Named Knowledgegroup"));
        kgc.Add(new KnowledgeGroup("Product Knowledge 5"));
        kgc.Add(new KnowledgeGroup());
        kgc.RemoveAt(2);
        kgc.Add(new KnowledgeGroup());
        kgc.Add(new KnowledgeGroup());
        kgc[6].Name = "Renamed KnowledgeGroup";
        kgc.Add(new KnowledgeGroup());

    }

    public bool Test()
    {
        bool Success = true;
        Success &= WriteAndCompare(kgc[0].Name, "Product Knowledge 1");
        Success &= WriteAndCompare(kgc[1].Name, "Product Knowledge 2");
        Success &= WriteAndCompare(kgc[2].Name, "My Named Knowledgegroup");
        Success &= WriteAndCompare(kgc[3].Name, "Product Knowledge 5");
        Success &= WriteAndCompare(kgc[4].Name, "Product Knowledge 4");
        Success &= WriteAndCompare(kgc[5].Name, "Product Knowledge 3");
        Success &= WriteAndCompare(kgc[6].Name, "Renamed KnowledgeGroup");
        Success &= WriteAndCompare(kgc[7].Name, "Product Knowledge 6");
        return Success;
    }

    public bool WriteAndCompare(string str1, string str2)
    {
        bool Success;
        Console.WriteLine("{0,-25} = {1,-25} : {2}", str1, str2, (Success = str1.Equals(str2)));
        return Success;
    }
}
```

Figur 5.4.2 – Test af den automatiserede navngivning af KnowledgeGroups i en KnowledgeGroupCollection vha. Unit Test Frameworket

Den egentlige `Test()` metode sammenligner de givne navne med de forventede navne. Testen udskriver desuden testresultater på skærmen. Udskriften er imidlertid ikke nødvendig for at testen kan fungere.



Figur 5.4.3 – Resultat af den udførte test vha. Unit Test Frameworket

Testen viser, at navngivningen fungerer som forventet og testen er derfor bestået.

5.5 Løsning på fejl fundet i tests

I de udførte test er der kun blevet fundet én fejl, der opstår når PMClass objekter tilføjes til deres egne eller deres childklassers childcollections, hvorved StackOverflowExceptions opstår når produktmodellen tegnes i et diagram.

Der bør tages hensyn til denne fejltype i datamodellen. Her må det ikke være muligt at lægge en klasse i sin egen eller sine egne children childcollections. Desuden skal der laves en metode – f.eks. IsInChildCollection, der returnerer true hvis flytningen ville medføre fejl. Denne metode vil kunne benyttes af GUI når flytning og linking vha. drag and drop ikke skal være tilladt og paste skal være deaktiveret samt af datamodellen i PMClassCollections Add metode.

KAPITEL 6

Konklusion

I løbet af dette bachelor projekt er der blevet udviklet en fungerende prototype på et værktøj til dokumentation af produktmodeller. Prototypen er blevet udviklet på baggrund af to artikler [Haug & Hvam (2006a) og (2006b)]. Der er blevet lagt stor vægt på at bevise konceptet i disse artikler. Af denne grund er softwarens funktion set som den vigtigste del af projektet. Mindre vægt er der lagt på softwarens arkitektur.

Softwaren beviser konceptet i de to artikler. På baggrund af dette bachelor projekt vil jeg i samarbejde med mine to vejledere skrive en artikel, der beskriver løsningen og dermed konceptets bevis. Denne artikel skal offentliggøres på en konference.

Artiklerne [Haug & Hvam (2006a) og (2006b)] var ved begyndelsen på projektet endnu ikke færdiggjort. Udviklingen af softwaren parallelt med videreudviklingen af kravene førte til at nye idéer opstod i samspil mellem parterne. Dermed omfatter softwaren en del funktioner, der når udover afgrænsningen. Et enkelt i afgrænsningen planlagt punkt er ikke kommet med i prototypen. Således er der endnu ikke implementeret en Undo-funktion.

I rapporten her beskrives baggrunden for projektet samt en del af kravene fra de før nævnte artikler. Softwarens arkitektur og implementering beskrives i detaljer og det demonstreres at prototypen faktisk fungerer og kan bruges til dokumentering af en produktmodel.

Prototypen vil evt. blive benyttet til demonstration ved præsentation af de to artikler på konferencen IMCM'06 i Hamborg, Tyskland, Juni 22-23, 2006. Den er endvidere blevet demonstreret for GEA Niro A/S.

6.1 Fremtidige forbedringer

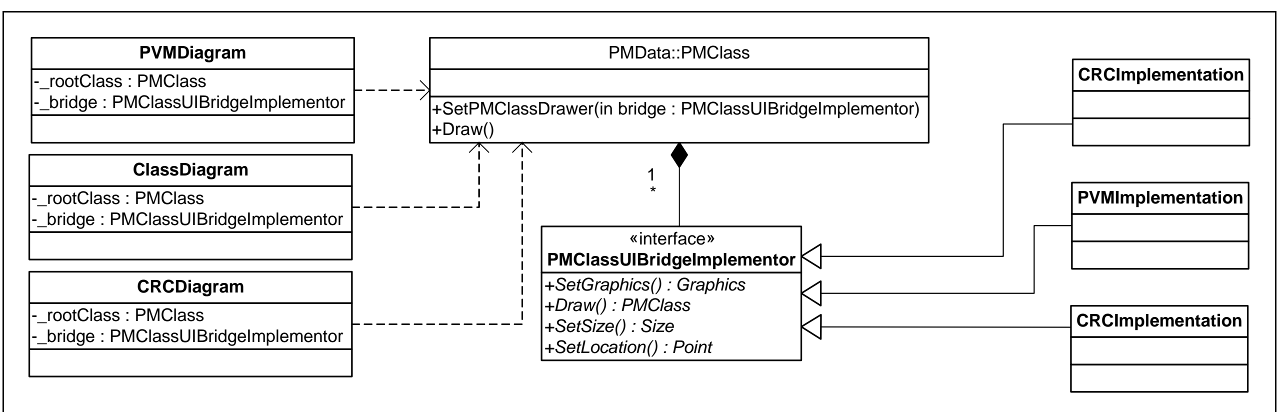
6.1.1 Arkitektur

Nogle beslutninger omkring arkitekturen er blevet til under implementeringen. Set i bakspejlet er nogle af disse beslutninger ikke optimale. I forbindelse med en videreudvikling af softwaren bør arkitekturen revideres. Enkelte idéer til forbedringer beskrives i de følgende afsnit.

Forbedret arkitektur til UI med Bridge Pattern

Det blev nævnt i afsnit 3.2.2, at UI'ets arkitektur ikke er fuldstændig ideel idet de enkelte views har en høj kobling til datamodellen. Et andet forslag gik ud på at give lavere kobling ved at lægge en Draw metode ned på datamodellen. Dette forslag er dog også dårligt, da det er uønsket at have UI-metoder i datamodellen.

I en fremtidig version af softwaren ville jeg træffe andre beslutninger mht. valget af arkitekturen. Dette kan ikke nås nu, da det ville vælte tidsplanen og opgaven ville ikke blive færdig til tiden. Jeg vil dog beskrive en løsning, der ville give lav kobling og alligevel flytte UI-metoder ud af data-modellen. Løsningen muliggøres ved brug af Gang of Four's (GoF) design pattern "Bridge" i en let simplificeret udgave.



Figur 3.2.2.3 – UML Diagram for en refactored udgave af præsentationskomponenten PMDocTool. Denne udgave benytter GoF Bridge Pattern for at opnå lav kobling mellem UI og datamodel.

Figur 3.2.2.3 viser, hvordan Bridge kan indsættes i arkitekturen. Den egentlige implementering af Draw-metoderne er flyttet fra UI-klasserne PVMDiagram, ClassDiagram og CRCDiagram til de tilsvarende Implementeringer af PMClassUIBridgeImplementor-interfacet. De tre UI-klasser kender ikke til detaljer om PMClass-objekter, men kan tegne PMClass objekter ved at kalde Draw-metoden på PMClass, der igen kalder Draw metoden in den forinden opsatte Bridge.

CRCDiagram

De to views ClassDiagram og PMDiagram har ligesom CRCDiagram en contextmenu, der muliggør redigering af produktmodellen idet klasser kan tilføjes, fjernes, flyttes og kopieres. Menuen på PVMDiagram og ClassDiagram er imidlertid ikke fuldt implementeret. CRCDiagram benytter en anden contextmenu, der ligger inde i CRCDiagram. Denne menus funktioner er færdigt implementerede. Men contextmenuen skal i alle tre views være ens og have de samme funktioner. Det vil være fornuftigt at flytte context menuen ud i en separat klasse, der er fælles for alle tre views.

ucCRCCard

CRCField er universel for alle typer felter og er derfor blevet for stor og uoverskuelig. CRCField bør laves om til en abstrakt klasse, der indeholder attributter og metoder, der er fælles for alle typer felter. Fra CRCField skal specialiserede versioner af CRC felter nedarve. Således bør der laves klasserne CRCTextField, CRCDateTimeField, CRCBoolField, CRCKnowledgeField og CRCSketchFied. Desuden skal hele klassen skrives om så den ligger som ét af tre valgbare views i et Bridge Pattern som vist på Figur 3.2.2.3. Yderligere forbedringer af arkitekturen kan opnås ved fuldstændig brug af MVC Pattern, hvoraf der i den nuværende arkitektur kun benyttes MV. Ved brug af Controller rapporteres til en observer ved ændring af data i viewet. PMClass, der er hooket op til observeren meddeles af observeren om de gennemførte ændringer. Brugen af observer ved redigering vil føre til yderligere afkobling af modellen fra viewet og dermed lavere kobling.

6.1.2 Implementering

Flere dele af softwaren blev ikke færdige i denne prototype. Disse skal laves færdige i en endelig version. Det handler her om følgende punkter:

1. Contextmenu i ClassDiagram og PVMDiagram skal laves færdige, så de fungerer fuldt ud.
2. Nye klasser skal placeres automatisk i klassediagrammet.
3. Printfunktionen skal laves færdig, så der kan udskrives diagrammer, der fylder flere sider, zoomes og ses et print-preview.
4. Det skal være muligt at zoome i PVM- og klassediagrammerne
5. Der skal tilføjes en undo-mulighed
6. Save-funktionen skal udvides. Lukkes programmet eller en produktmodel uden at modellen er gemt, skal der spørges om modellen skal gemmes.
7. De sidste felter i CRC-kortet skal laves så de fungerer.
8. Tekst i constraints skal delvist erstattes af hyperlinks.

Yderligere skal fejlen fundet under test rettes.

6.2 Demonstrationsmøde med GEA Niro A/S

Deltagere:

Jesper Riis, Configuration Team Supervisor
GEA Process Engineering Division, IT

Jesper Nielsen,
GEA Process Engineering Division, IT

Anders Haug, Ph.D. stipendiat,
IPL, DTU

Anders Degn

Den 25.04.2006 afholdtes et møde med formålet at præsentere prototypen og få feedback på denne.

Jesper Riis lagde ud med at fortælle om Niros brug produktmodeller og konfigureringssystemer samt om hans tilknytning til DTU og IPL. Han har tidligere afsluttet en Ph.D. på IPL og forsket i produktmodellering. En del af de i [Haug & Hvam (2006a) og (2006b)] nævnte krav til et dokumentationsværktøj er opstået på baggrund af hans forskning og inputs fra Niro.

Derefter viste Jesper Riis det tool, Niro benytter til dokumentation af produktmodeller. Dette tool er lavet som en template i Lotus Notes. Lotus Notes datastruktur kombineret med at Notes er et standard datapool værktøj i stedet for at være et værktøj udviklet med henblik på dokumentation af produktmodeller sætter naturligvis en række begrænsninger. Derfor afventer Niro iflg. Jesper Riis nu at DTU's forskning indenfor produktmodeller fører til et konkret værktøj, der kan benyttes af Niro inden for overkommelig tid. Skulle dette ikke ske, overvejer Niro nu at udvikle et eget dokumentationsværktøj, der opfylder deres krav.

Efter demonstrationen af Lotus Notes templaten demonstrerede Jesper Riis det konfigureringssystem hvori Niros produktmodeller er implementeret. Demonstrationen viste muligheden for at konfigurere komplette fabrikshaller med maskiner på kort tid. Systemet kunne automatisk lave tilbud, styklister og CAD-tegninger samt vise opbygningen af maskinerne i en fabrikshal i 3D. Alt dette er muligt på baggrund af den implementerede produktmodel, der på trods af det mangelfulde dokumentationsværktøj er yderst kompleks og godt beskrevet.

Efter demonstrationen af Niros nuværende værktøjer, demonstrerede Anders Degn prototypen på dokumentationssystemet, som er udviklet i forbindelse med dette eksamensprojekt. Værktøjet blev beskrevet på baggrund af en produktmodel for en delmængde af en bil. Funktionerne i CRC kort viewet er tæt på de samme som findes i Niros værktøj. Jesper Riis og Jesper Nielsen kommenterede den dokumenterede models objektorienterede opbygning med muligheden for at referere til samme instans af dele af produktet som værende en forbedring i forhold til det nuværende værktøj. Endvidere giver templaten ikke mulighed for at illustrere nedarvning. Dette er muligt i dokumentationsværktøjet. Selvfølgelig er der i prototypen på CRC kort viewet også mangler, der bør overvejes og implementeres i en evt. senere version af softwaren. Her nævntes versionsstyring og change

requests. Disse to muligheder er ganske vidst nævnt i [Haug & Hvam (2006a) og (2006b)] men er blevet afgrænset, da de ikke er nødvendige for at bevise disse artiklers koncept. Der er siden artiklerne blev skrevet desuden kommet ny ønsker fra Niro – bla. ønskes en mulighed for at lave frit definerbare tabeller til definition af Constraints samt en form for pseudoprogrammeringssprog til definition af disse. De automatisk genererede PVM og klasse diagrammer samt muligheden for at arbejde med produktmodellen fra disse views vækkede begejstring. Hidtil er disse diagrammer blevet tegnet i hånden.

Både Jesper Riis og Jesper Nielsen gav meget positivt feedback på prototypen og ønskede en kopi af programmet til test. De mente at programmet meget godt viser mulighederne indenfor dokumentation af produktmodeller og udtrykte at de var meget imponerede over softwarens funktionelle omfang på trods af den relativt korte udviklingsperiode. De påpegede dog at muligheden for at kunne arbejde med modellen på en central database mangler og er nødvendig inden Niro vil kunne benytte softwaren.

APPENDIKS A

Referencer

Haug & Hvam (2006a).The modelling techniques of a documentation system that supports the development and maintenance of product configuration systems, IMCM'06, Hamborg, Tyskland, Juni 22-23, 2006

Haug & Hvam (2006b). CRC-cards for the development and maintenance of product configuration systems, IMCM'06, Hamborg, Tyskland, Juni 22-23, 2006

Hvam, L. & Malis, M. (2001): A Knowledge Based Documentation Tool for Configuration Projects. *Proceedings of World Congress on Mass Customization and Personalization*, Hong Kong, Okt. 1-2, 2001.

Hvam, L., Mortensen, N.H. & Riis, J. (2005b): *Produktkonfigurering*. Publikation til undervisning ved DTU, Lyngby: IPL, DTU, 2005.

Hvam L., Pape, S., Jensen, K.L. & Riis, J. (2005a): Development and maintenance of product configuration systems - Requirements for a documentation tool. *International Journal of Industrial Engineering*, 12 (1), 79-88.

Riis, J. (2003): Fremgangsmåde for opbygning, implementering og vedligeholdelse af produktmodeller - med fokus på konfigureringssystemer. Ph.d.-afhandniling, Lyngby: IPL, DTU, 2003.

Young J. (2003): Generating missing Paint event for TreeView and ListView controls. http://www.thecodeproject.com/cs/miscctrl/genmissingpaintevent.asp

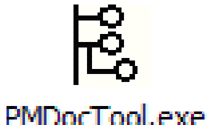
mav.northwind (2004): In-place editing of ListView subitems http://www.codeproject.com/cs/miscctrl/ListViewCellEditors.asp

Gerrard, P. (1997): Testing GUI Applications (http://www.evolutif.co.uk/GUI/TestGui.html)

APPENDIKS B

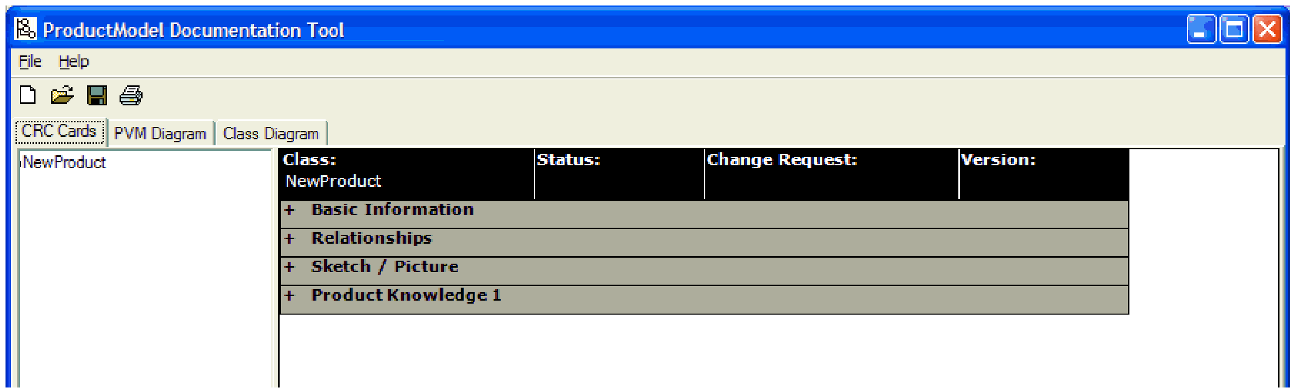
Brugervejledning

Programmet startes ved at klikke på ikonet

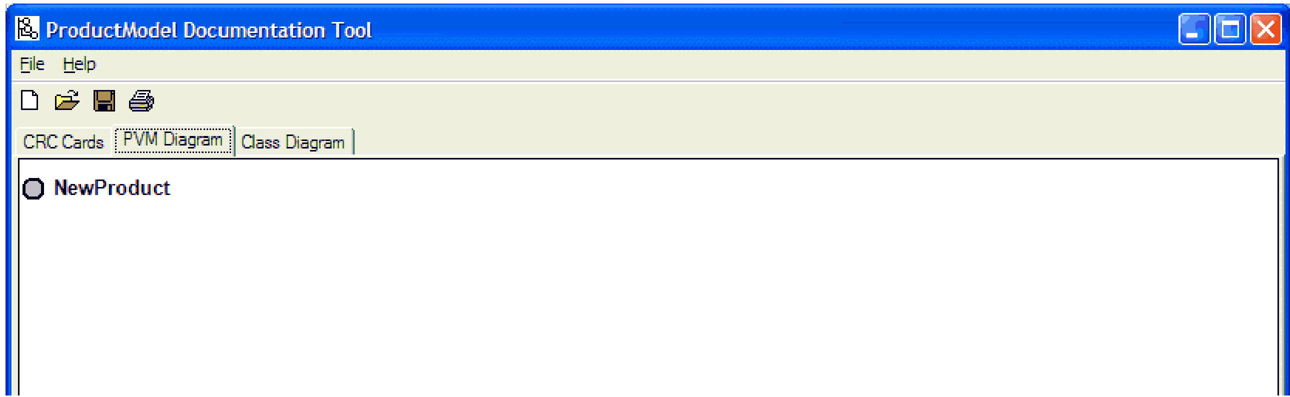


Programmets tre views

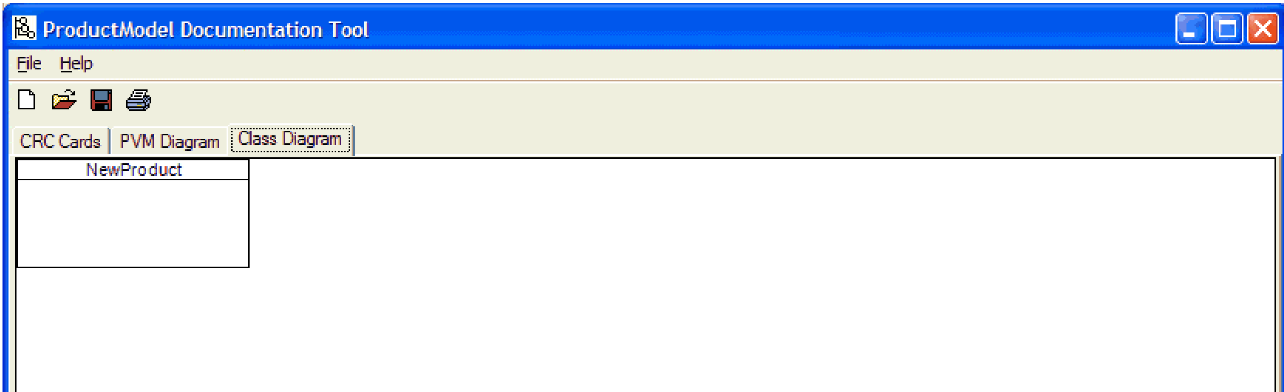
Når programmet startes, ses det første view CRC Cards.



Der kan navigeres mellem de tre views ved at klikke på fanebladene. Klikkes på PVM Diagram, skiftes til viewet PVM Diagram.

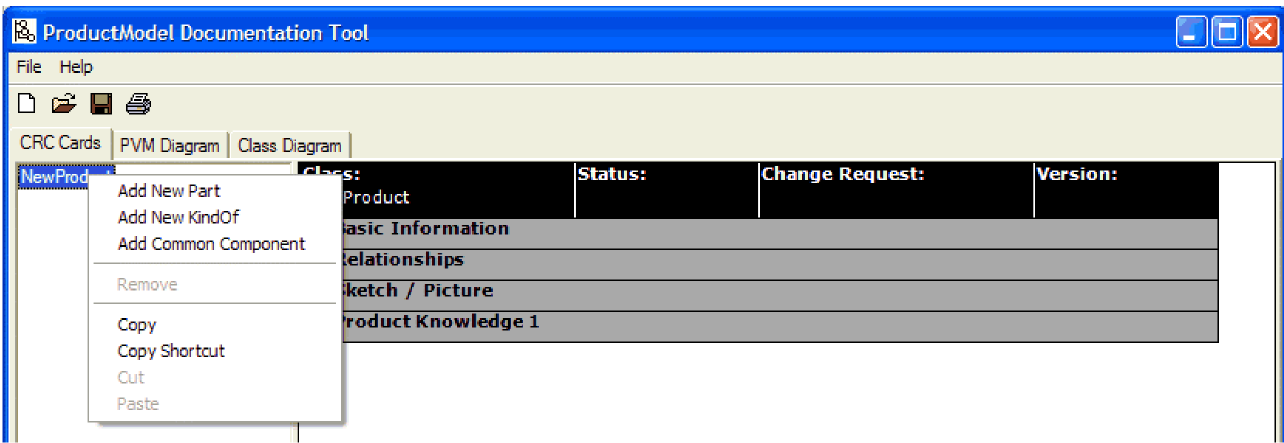


Klikkes på Class Diagram, skiftes til viewet Class Diagram.

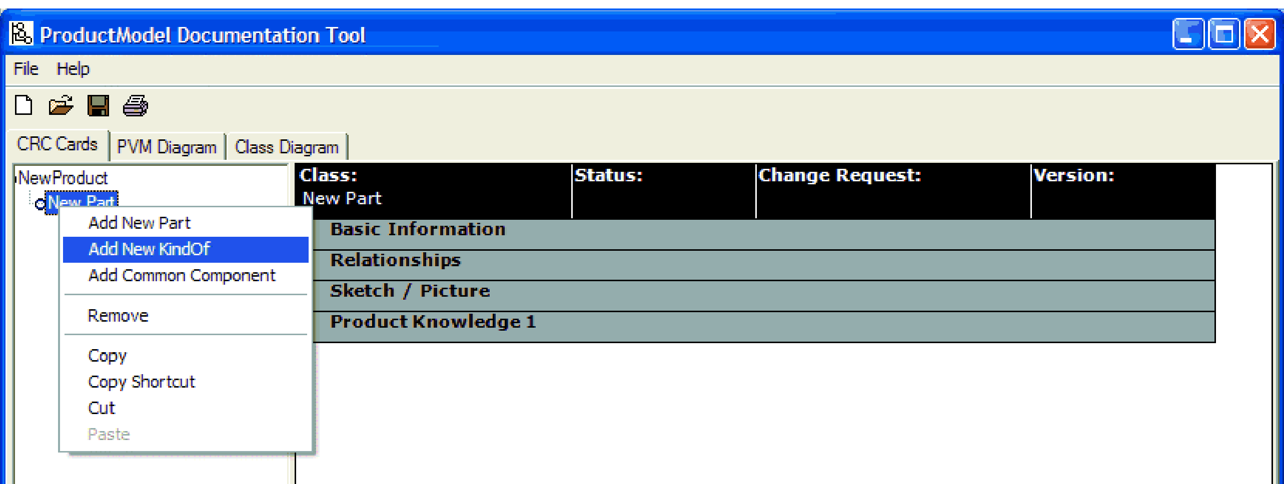


Arbejde med træstrukturen i viewet CRC Cards

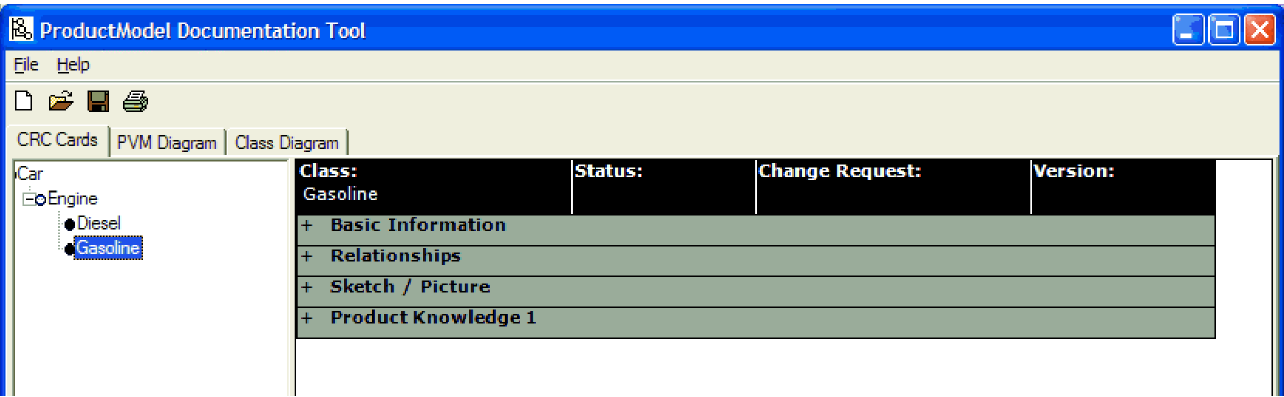
Viewet CRC Cards er opdelt i to dele – en træstruktur i venstre side og et CRC kort i højre side. Når programmet startes, er et produkt (”NewProduct”) allerede oprettet. Vha. træstrukturen kan der nu oprettes nye dele og varianter. Højreklikkes på produktet, vises en contextmenu. For at tilføje en ny del til produktet klikkes på ”Add New Part”. Den ny del ses i træstrukturen med navnet ”New Part”.



En ny variant oprettes ligeledes vha. contextmenuen ved at klikke på ”Add New KindOf”. Den ny variant tildeles automatisk navnet ”New KindOf”.



Dele er i træstrukturen symboliseret vha. symbolet ○, varianter har symbolet ●. Begge kan omdøbes i træstrukturen ved at klikke en enkelt gang på delen eller varianten, der skal omdøbes. Direkte i træet kan det ny navn nu skrives ind.



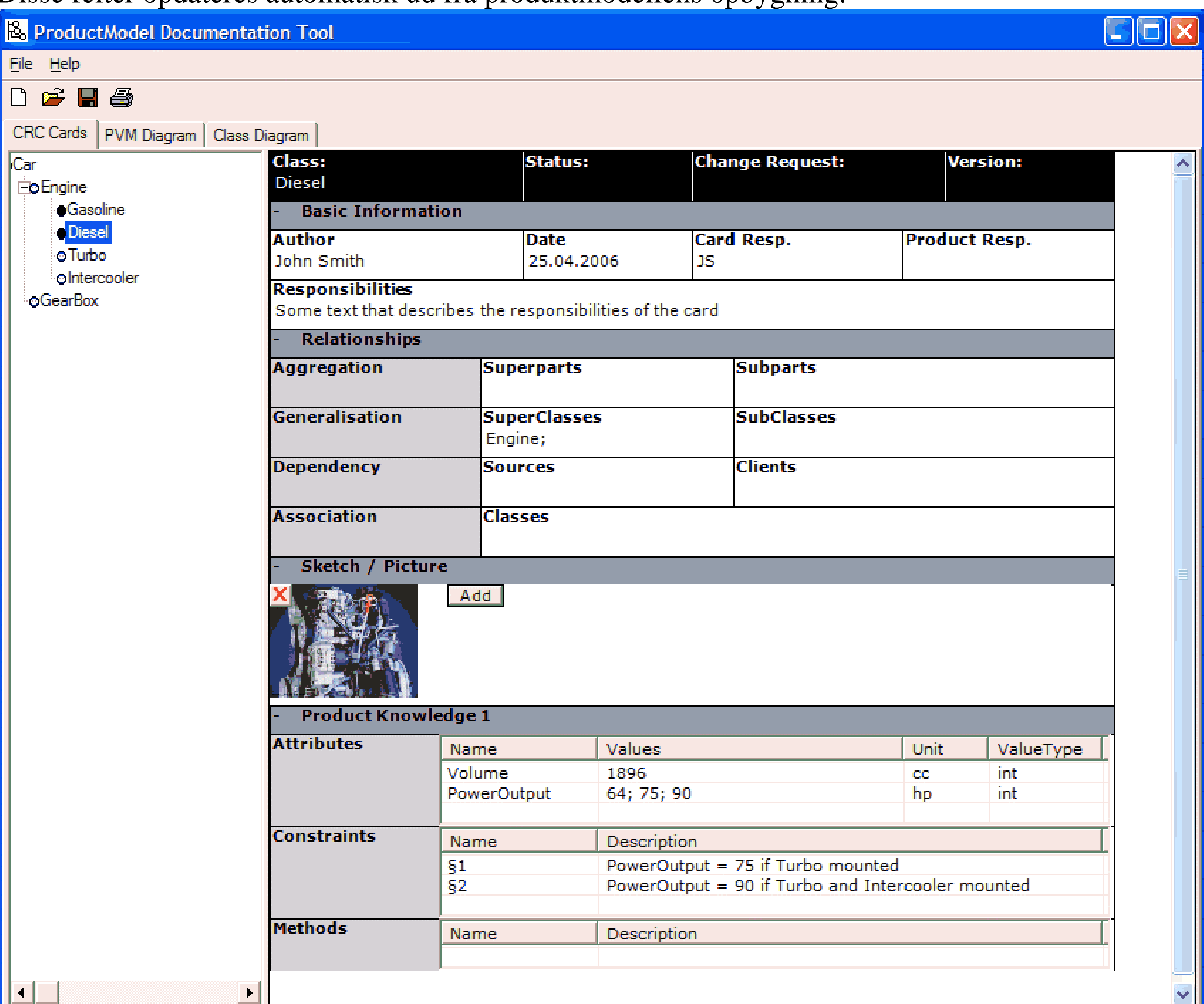
Det er endvidere muligt at slette, kopiere, linke og flytte eksisterende CRC kort ved hjælp af contextmenuen. Menupunktet Copy laver en eksakt kopi af et CRC kort. Dette kan indsættes andetsteds i træstrukturen vha. menupunktet paste. Copy Shortcut laver en henvisning til et CRC kort, der ligeledes kan indsættes andetsteds i træstrukturen. Forskellen på Copy og Copy Shortcut er, at et Shortcut repræsenterer det samme CRC kort, hvorimod en kopi opretter et nyt identisk CRC kort. Ændringer foretaget i et kopieret kort ses ikke i kopien. Ændringer foretaget i et kort, der er henvist til ses begge steder. Denne funktion giver muligheden for at kunne genbruge dele og varianter flere steder i modellen.

Copy, Copy Shortcut og Move er desuden muligt vha. drag and drop. Dragges med venstre museknap uden samtidig at trykke på tastaturet, flyttes et kort i træet når der droppes. Dragges med venstre museknap samtidig med at Ctrl hhv. Alt trykkes ned på tastaturet, ses ved siden af cursoren

et lille ”+” hhv. et lille ”↪” symbol. Disse symboler svarer til Copy hhv. Copy Shortcut. Dragges med højre museknap, vises en contextmenu når der droppes. Contextmenuen giver mulighederne Move here, Copy here og Link here, hvor Link here svarer til Copy Shortcut.

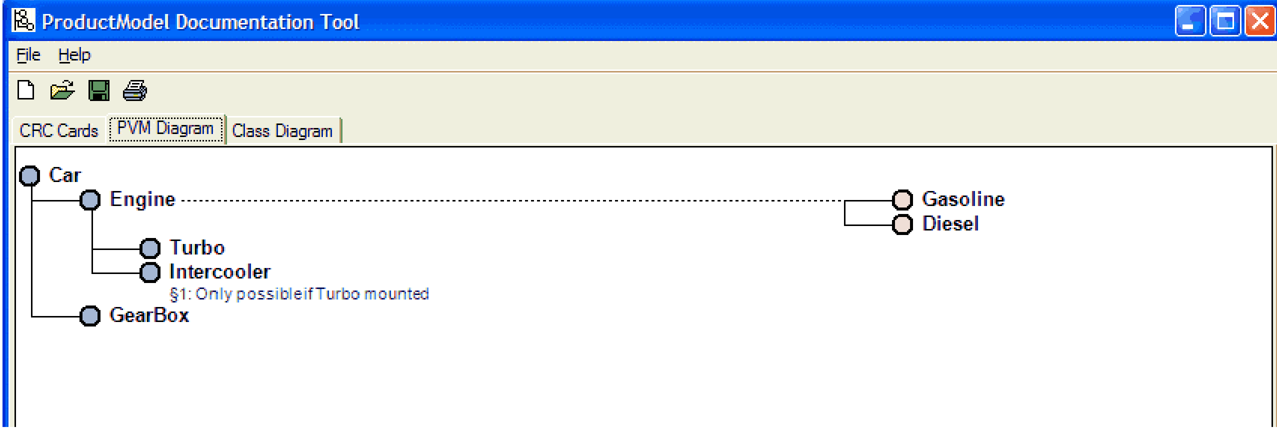
CRC kortet

Idet der klikkes på en del eller en variant vises i højre side et CRC kort. Når programmet startes ses blot en overskrifts linie samt overskrifter på CRC kortets grupper. Grupperne kan åbnes og lukkes ved at klikke på deres overskrifter. Derved vises alle felter i de åbnede grupper. Felterne i overskriften samt felterne i grupperne Basic Information, Sketch / Pictures og Product Knowledge kan ændres ved at klikke med musen i dem. Felterne i gruppen Relationships kan ikke redigeres. Disse felter opdateres automatisk ud fra produktmodellens opbygning.

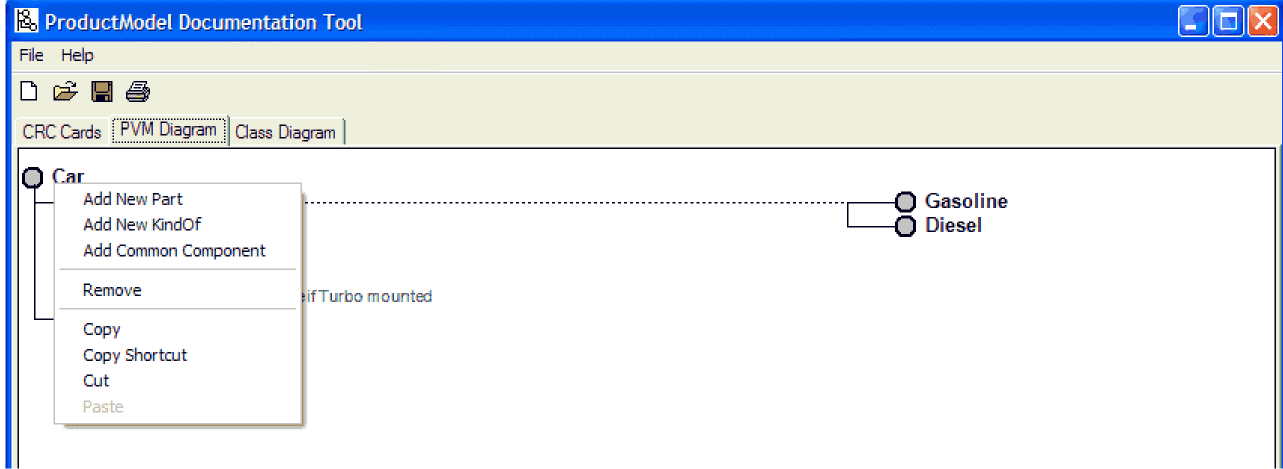


PVM Diagram

Ændringerne i produktmodellen foretaget i CRC Cards ses også i PVM Diagram. Her ses dele til venstre i en træstruktur og varianter ses til højre. Dobbeltklikkes på en del eller en variant, åbnes et nyt vindue og det tilhørende CRC kort vises i et nyt vindue. I vinduet kan kortet redigeres.



Det er muligt at tilføje dele og varianter i PVM Diagram på samme måde som det er muligt i træstrukturen i CRC Cards. Contextmenuen vises ved højreklik på en del.



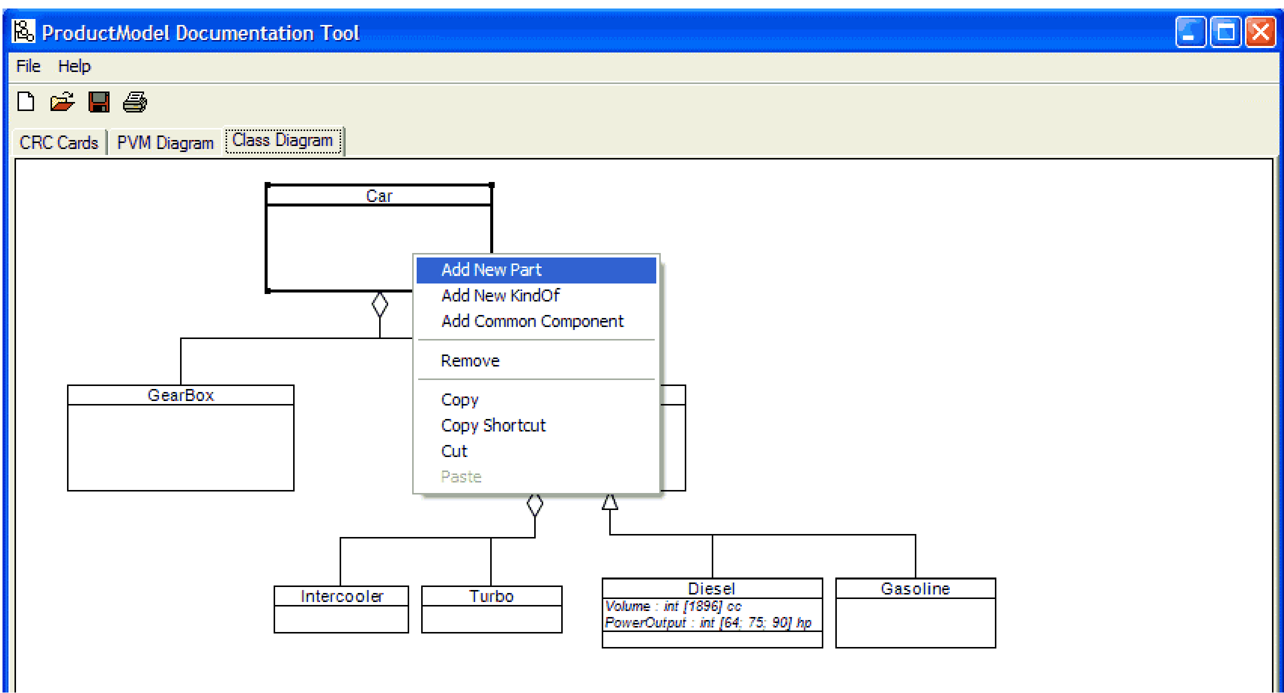
Class Diagram

Ligesom i PVM Diagrammet afspejles alle ændringer på produktmodellen i klassediagrammet. Dog står alle klasser samme sted øverst til venstre i diagrammet. Disse skal manuelt placeres i diagrammet ved at trække dem på plads med musen. Klassernes størrelse kan ændres ved at trække i et af de fire punkter i klassernes hjørner.

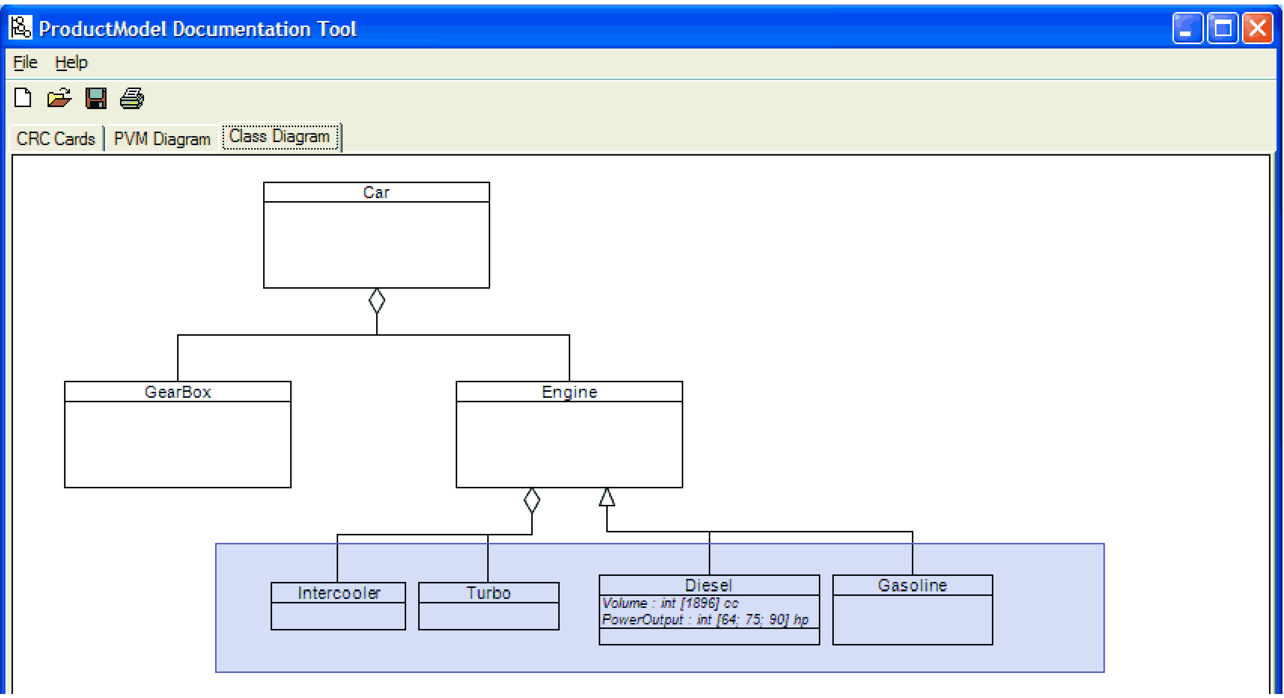
I klassediagrammet svarer aggregering til PVM diagrammets dele (PartOf) og generalisering svarer til varianter (KindOf). Ligesom i PVM Diagrammet og træstrukturen i CRC Cards, er det her muligt

at tilføje dele og varianter til modellen. Contextmenuen vises ved højreklik på en del eller en variant.

Dobbeltklikkes på en del eller en variant, åbnes et nyt vindue og det tilhørende CRC kort vises i et nyt vindue. I vinduet kan kortet redigeres.

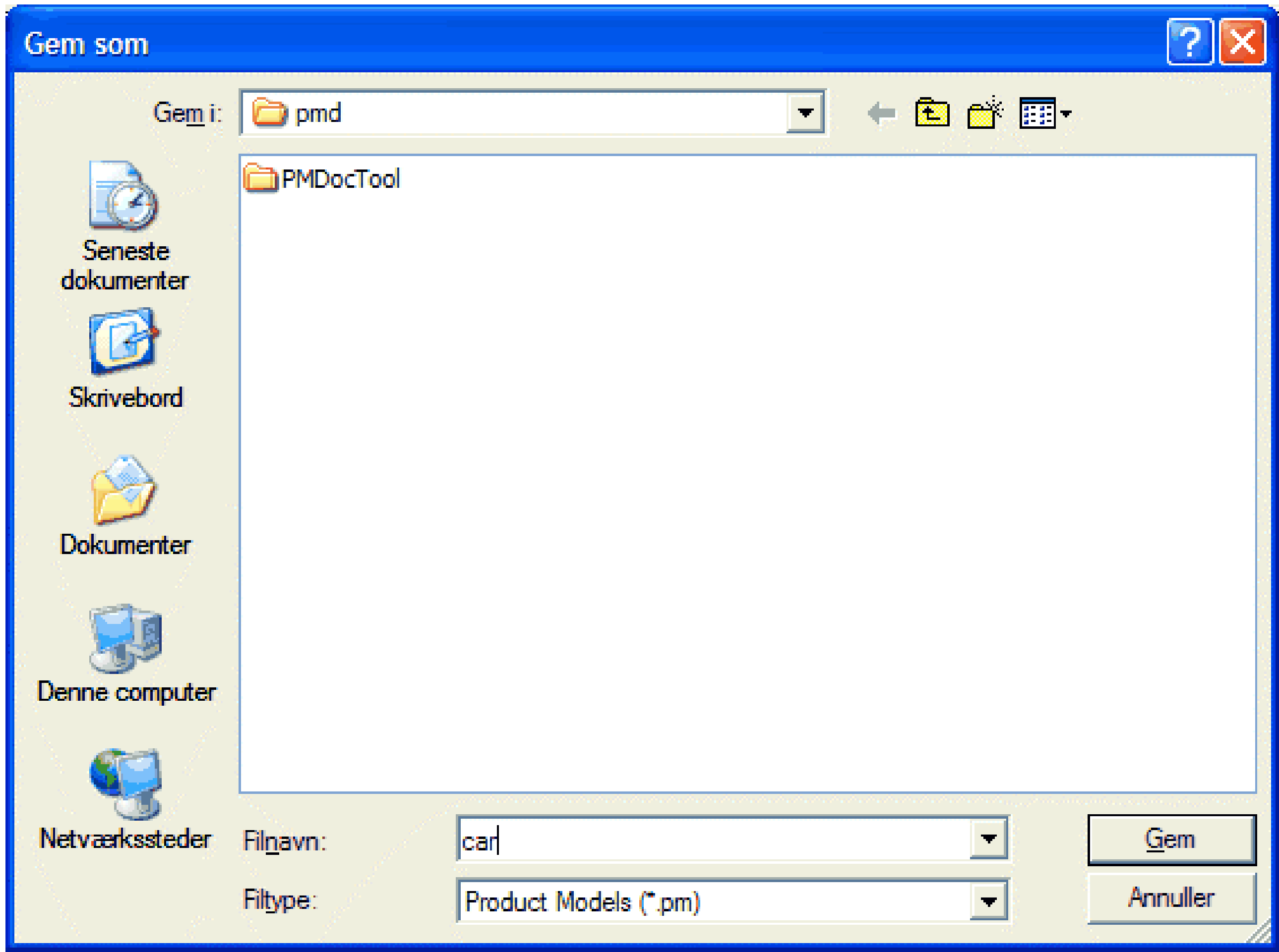


Ønsker man at flytte en hel gruppe af klasser på én gang, er det nødvendigt at selektere flere klasser. Dette gøres ved at klikke udenfor klasserne med musen og trække den hen over de klasser, der skal selekteres. Herved indrammes klasserne i en blå ramme. Når musen slippes er klasserne selekterede. Den selekterede gruppe kan flyttes ved at trække i én af de selekterede klasser med musen.



Filhåndtering

En produktmodel kan gemmes på disken ved at klikke på diskettesymbolet i programmets toolbar eller gennem menupunktet File – Save eller Save As. Diskettesymbolet gemmer modellen med det samme navn som modellen allerede har fået ved en tidligere lagring. Har modellen endnu ikke fået et filnavn, åbnes ”Gem som”-dialogen. Her kan et filnavn og en placering angives. Den samme dialog kommer altid frem når der klikkes på Save As i menuen File.



En lagret produktmodel kan hentes fra disken ved at klikke på mappesymbolet i programmets toolbar eller ved at vælge menupunkter Open i menuen File.

File – New og papirsymbolet i programmets toolbar starter en ny produktmodel.

Det aktuelle view kan udskrives på en printer ved at klikke på printersymbolet eller på menupunktet print i menuen File.

The modelling techniques of a documentation system that supports the development and maintenance of product configuration systems

Anders Haug and Lars Hvam

Abstract

This paper deals with the subject of how to create a documentation system to support the development and maintenance of product configuration systems (PCS).

A procedure for building product configuration systems from the Centre for Product Modelling (CPM) at the Technical University of Denmark has, for more than ten years, been applied in numerous projects. The CPM-procedure includes three main modelling techniques for defining the knowledge to be implemented in a PCS. In the development phase the models are normally created by using different kinds of software. This means that there no is automatic integration between the models, why some information has to be modelled repeatedly, which is time consuming and may lead to errors. As the products of a company change, the knowledge base of the PCS has to be updated for the PCS to support the company processes. When dealing with PCS's with large knowledge bases, it is often necessary to use external documentation for getting an overview of the implemented knowledge that allows updates to take place. These aspects present a demand for a coherent documentation system that supports the modelling techniques of the CPM-procedure, which does not exist at present.

During the last five years research has been carried out on how to create a documentation system that supports the CPM-procedure. However, this research does not provide detailed definitions of the different modelling techniques that should be included. In order to fulfil the ambition of creating such a system these definitions have to be formulated, which is done by this paper.

Keywords

Product modelling, product configuration, knowledge representation, documentation of product models.

1 Introduction

To meet customer demands for customised products, while keeping the costs low, product configuration systems (PCS) can be applied. A PCS can be defined as a product-oriented expert system, which allows users to specify products by selecting components and properties under restriction of valid combinations. The application of configuration technology can produce benefits such as: shorter lead times, reduction of resources needed to produce specifications and fewer errors in specifications (Hvam, 2004).

The development of a PCS implies a relocation of knowledge from domain experts to a software system. This means that domain knowledge has to be represented, which is one of the greatest challenges of a configuration project (Sabin & Weigel, 1998; Hansen et al., 2003). The development of PCS's can also imply the defining of new domain knowledge, as when a PCS project is started, often the included product families do not have a well-considered permanent architecture or a basis for creating a robust generic product model (Pulkkinen, 2000). Great involvement of domain experts is therefore often necessary when developing PCS's.

To support the development of PCS's, several approaches have been proposed. In this paper the focus is on a procedure from the Centre for Product Modelling (CPM) at the Technical University of Denmark, which includes three major techniques for modelling configuration knowledge. In the development phase, models are normally drawn by using programs such as MS Visio, Excel and Word or CASE tools (Computer Aided Software Engineering) such as IBM's Rational Rose. The use of different kinds of software means that there no is automatic integration between the models. Therefore some information has to be modelled repeatedly, which is time consuming and may lead to errors.

Over time the products of a company change, which means that the knowledge base of the PCS has to be updated for the PCS to support the company processes. As a PCS can easily have a knowledge base that consists of thousands of interrelated components, constraints and methods, and the modelling environment of a PCS often does not provide an adequately comprehensible overview of the implemented knowledge, getting an overview that allows updates to take place often requires external documentation.

The described aspects present a demand for a coherent documentation system that supports the modelling techniques of the CPM-procedure, which at the present time does not exist. Research on how to create this documentation system has been carried out in recent years (Hvam & Malis, 2001; Hvam et al., 2005a). This research offers a list of requirements, including the modelling techniques to be supported, but does not explain in detail how these modelling techniques should be presented in the user interface, and what is even more critical, how the different modelling techniques should be mapped to each other. Existing research does, therefore, not provide an adequate basis for creating a documentation system that supports the required modelling techniques. To fulfil this need this paper presents such definitions. However, this paper does not deal with functional requirements in detail, which to some extent has been provided by earlier research (Hvam et al., 2005a).

In section 2 the CPM-procedure is outlined with a focus on its techniques for the modelling of product knowledge. Next, in section 3, previous research concerning the documentation of PCS's is resumed. In section 4 a definition of a documentation system to support the development and maintenance of PCS's is presented. The paper ends with a conclusion in section 5.

2 The CPM-procedure and applied modelling techniques

2.1 The CPM-procedure

The CPM-procedure was introduced by Hvam (1994) and has since undergone further development at CPM. The CPM-procedure has been applied in several projects, which in many cases has led to major improvements in lead times, the number of errors, the use of resources etc. (Hvam et al., 2002, Hvam, 2004). The CPM-procedure embraces many aspects of a product configuration project and provides guidelines for its seven phases: 1 (Business) Process analysis, 2 Product analysis, 3 Object-oriented analysis, 4 Object-oriented design, 5 Programming, 6 Implementation and 7 Maintenance. In the product analysis phase the prescribed modelling technique is the so-called product variant master (PVM). In the succeeding object-oriented phases the PCS is defined by using UML class diagrams as the primary notation. The object-oriented analysis phase includes a transfer of PVM's to class diagrams, which is not intended to be done in a one-to-one manner but with optimisation of the models in mind. The basic argument for using both modelling techniques instead of just one of them is that the PVM notation offers a relatively easily comprehensible syntax, which, as experience has shown, is easily accepted by domain experts. On the other hand, class diagrams are richer and more flexible, though still more formalised, wherefore this notation is better suited for the handling of complex designs. Also the widespread use of UML within software development is an important factor. To allow more detailed specifications of the classes represented in diagrams, the CPM-procedure proposes the use of special CRC-cards (Class, Responsibilities, and Collaborators).

2.2 Product variant masters

As mentioned, the use of PVM's is prescribed in the product analysis phase of the CPM-procedure. The newest version of the CPM-procedure (Hvam et al., 2005b) includes an updated definition of the PVM-notation, originating from (Harlou, 2005). This is shown in figure 1.

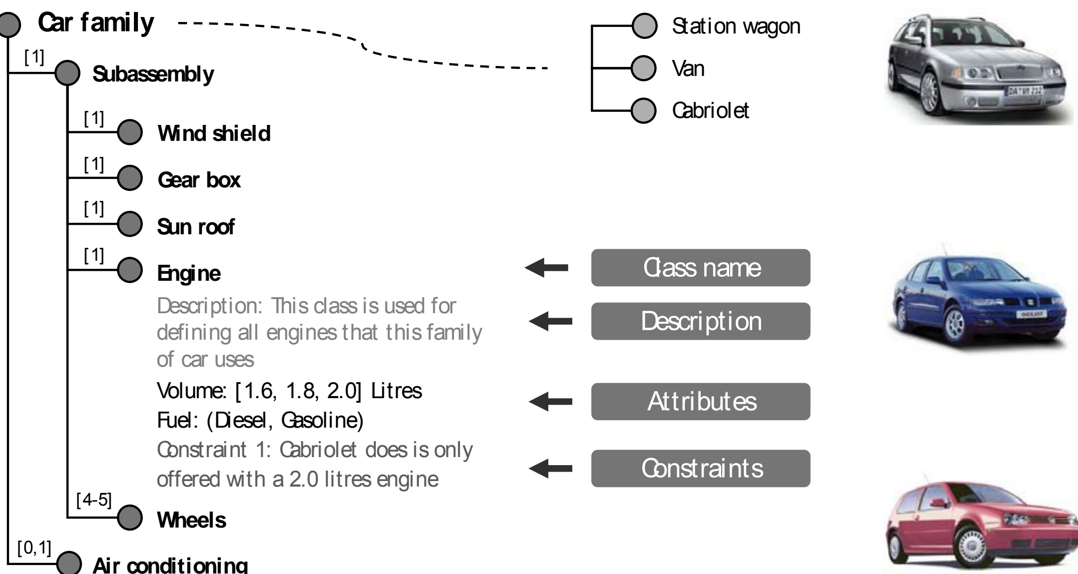


Figure 1: Definition of the PVM-notation (Hvam et al., 2005b)

A PVM consists of two sections, on the left side *generic part-of structure* and on the right side *generic kind-of structure*. The part-of section describes the elements of which a product can consist, while the kind-of section describes the possible variance of an element.

A PVM-model is intended to be printed on a large sheet of paper and be discussed in repeated sessions of model-managers (often industrial engineers) and domain experts, leading to continuous refinements of the model.

Experience from the use of the PVM-technique (however, based on earlier definitions of the notation) has shown that a PVM can be a useful tool for discussing where to start modelling the PCS, which product variants to include, the stability of products etc. (Hvam et al., 2002; Hvam, 2004).

2.3 Class diagrams

In the object-oriented analysis phase of the CPM-procedure, the problem domain and the field of application in which the IT-system will be used are analysed. In the succeeding design phase, the focus changes towards implementation issues, such as design of user interfaces, adaptation of analysis model, data management etc. A main purpose of the object-oriented phases is to formalise the model created in the preceding product analysis phase in a way that allows it to be implemented in a configuration system. To do so, UML diagrams, primarily class diagrams, have been chosen as the modelling technique.

The CPM-procedure (Hvam et al., 2005b) prescribes the use of a subset of the class diagram notation, which is shown in figure 2. For the full notation see (OMG, 2005).

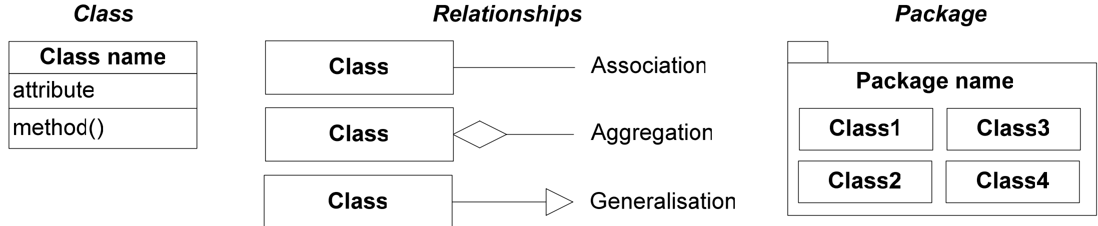


Figure 2: The selection of class diagram model elements of the CPM-procedure

According to Hvam et al. (2005b), the *kind-of* and *part-of* relations of a PVM can be translated into respectively *generalisation* and *aggregation* relationships in a class diagram. Aggregation is in this context perceived in a broad sense, not differentiating between *aggregation* and *composition* (symbolised by a black diamond). In this paper, the composition relationship is applied, as this is most correct according to the definitions of the two relationship types (OMG, 2005; Fowler, 2005). Figure 3 gives a simple example of how part-of and kind-of structures can be transferred to a class diagram.

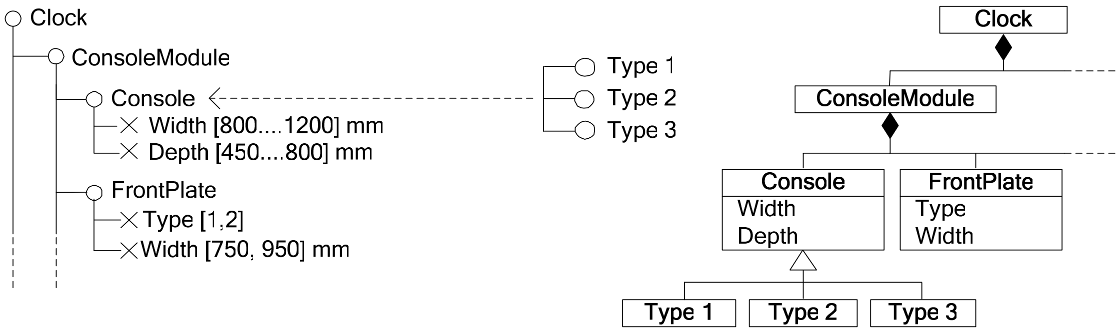


Figure 3: Transfer from a PVM to a class diagram (based on Hvam et al., 2005b)

Not included in (Hvam et al., 2005b) but in (Riis, 2003) is the UML concept *stereotypes*, which is also found in other PCS development approaches that include class diagrams (Felfernig et al.,

2000). Stereotypes allow the introduction of new model elements based on existing elements (Arlow & Neustadt, 2005), which enables the use of platform or domain specific terminology or notation. Some stereotypes are predefined in the UML others may be user-defined. Stereotypes can be collected in a profile, which is a concept that was added in the UML 1.4 specification for extending a reference metamodel to target a specific platform or domain (OMG, 2005). Riis (2003) defines a collection of stereotypes to be applied in class diagrams in a PCS development context. This, among others, includes: Product family, Part, Function and Property.

2.4 CRC-cards

CRC-cards were invented by Cunningham in the late eighties (Fowler, 2005) and presented in a paper by Beck and Cunningham (1989). These CRC-cards consist of the class name together with two columns for *responsibilities* and *collaborators*. In brief, responsibilities are summarisations of the things an object should do, while collaborators are the other classes with which a class must work together (Fowler, 2005). CRC-cards can be seen as an alternative to structural diagrams in the early phases of a software development project as it is a good technique for exploring object interactions by moving cards around. CRC-cards are not part of UML but can be a valuable technology for learning object-orientation. CRC-cards can also be applied beyond the early phases of a project for different purposes (Bennet et al., 2002).

Hvam and Riis (1999) present a revised kind of CRC-cards to be used in product configuration projects. This CRC-card definition was later updated in (Hvam et al., 2005b), which is seen in figure 4. On the CRC-cards, the fields *Superparts* and *Subparts* refer to aggregation relationships, while the fields *Subclasses* and *Superclasses* refer to generalisation relationships.

<table>
<tr><th>Class name:</th><th colspan="2">Date:</th><th>Author/version:</th></tr>
<tr><td colspan="4">Responsibilities:</td></tr>
<tr><td colspan="2">Aggregation</td><td colspan="2">Generalisation</td></tr>
<tr><td colspan="2">Superparts:</td><td colspan="2">Superclasses:</td></tr>
<tr><td colspan="2">Subparts:</td><td colspan="2">Subclasses:</td></tr>
<tr><td colspan="4">Sketch:</td></tr>
<tr><td colspan="2">Attributes:</td><td colspan="2">Collaborators:</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="2">System methods:</td><td colspan="2"></td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="2">Product methods:</td><td colspan="2"></td></tr>
<tr><td colspan="2">Internal methods:
External methods:</td><td colspan="2"></td></tr>
</table>

Figure 4: The latest CRC-card definition from CPM (Translated from Hvam, 2005b)

2.5 Dealing with adapted use of the CPM-procedure

Even though the CPM-procedure includes a transfer from PVM's to class diagrams, Riis (2003) and Hvam et al. (2005b) recognise that in some projects, the creation of PVM's with matching CRC-cards would be adequate, wherefore class diagrams should not be elaborated. Riis (2003) states that either PVM's or class diagrams should be chosen as final documentation, since maintaining the same information in two different diagrams would most often not be efficient.

Hvam et al. (2005b) distinguish between two parts of a PCS that are relevant to document, namely the knowledge base and other software aspects, such as user interface design and software architecture. As class diagrams are better suited for describing software aspects than PVM's, a combination of both PVM's and class diagrams for documenting different aspects could be a sensible choice in some cases. This type of approach is described by Haug and Hvam (2005) who propose a procedure for developing 3D PCS's in which PVM's are applied for the design of the knowledge base and class diagrams for the design of other PCS aspects.

3 Documentation of PCS's

3.1 Research concerning documentation of PCS's

As mentioned, the CPM-procedure prescribes the use of PVM's, class diagrams and CRC-cards during the development and maintenance of PCS's. At present, software which supports all three techniques in an integrated fashion does not exist. The lack of such a documentation system means that different kinds of software are applied. This causes a need for manual transfers of information between PVM's, class diagrams and CRC-cards, which are time consuming activities that hold risks of errors. This aspect of product configuration has been subject to some investigations.

During 2003 and 2004 the experience gained with product configuration within twelve Danish companies was investigated in technical, economical and organisational dimensions (Edwards et al., 2005). According to these investigations, the documentation task was often the first activity to be postponed or cut away from a PCS project. Doing so might save some work in the short run, but in the long run have big consequences, as several of the companies could not maintain or further develop their PCS. Another problem experienced due to poor documentation was problems in the daily communications between persons involved in product development and PCS development respectively.

Hvam and Malis (2001) define requirements for a documentation system to support product configuration projects and describes the development of a prototype created in Lotus Notes. Today, this Lotus Notes application is applied by the companies, GEA Niro A/S and American Power Conversion A/S (APC), though in further developed versions (Hvam, 2004). Figure 5 depicts a screenshot from the Lotus Notes application at GEA Niro A/S.

Even though the use of the Lotus Notes based documentation system has shown that there are significant benefits of applying such a tool for the maintenance of PCS's (Hvam, 2004), much is still to be wanted. The Lotus Notes application does not support class diagrams or the full PVM notation, but only offers a hierarchical list of components with matching CRC-cards. Consequently, both of the above-mentioned companies, who use the Lotus Notes application, apply other software for the creation of PVM's during the development phase.

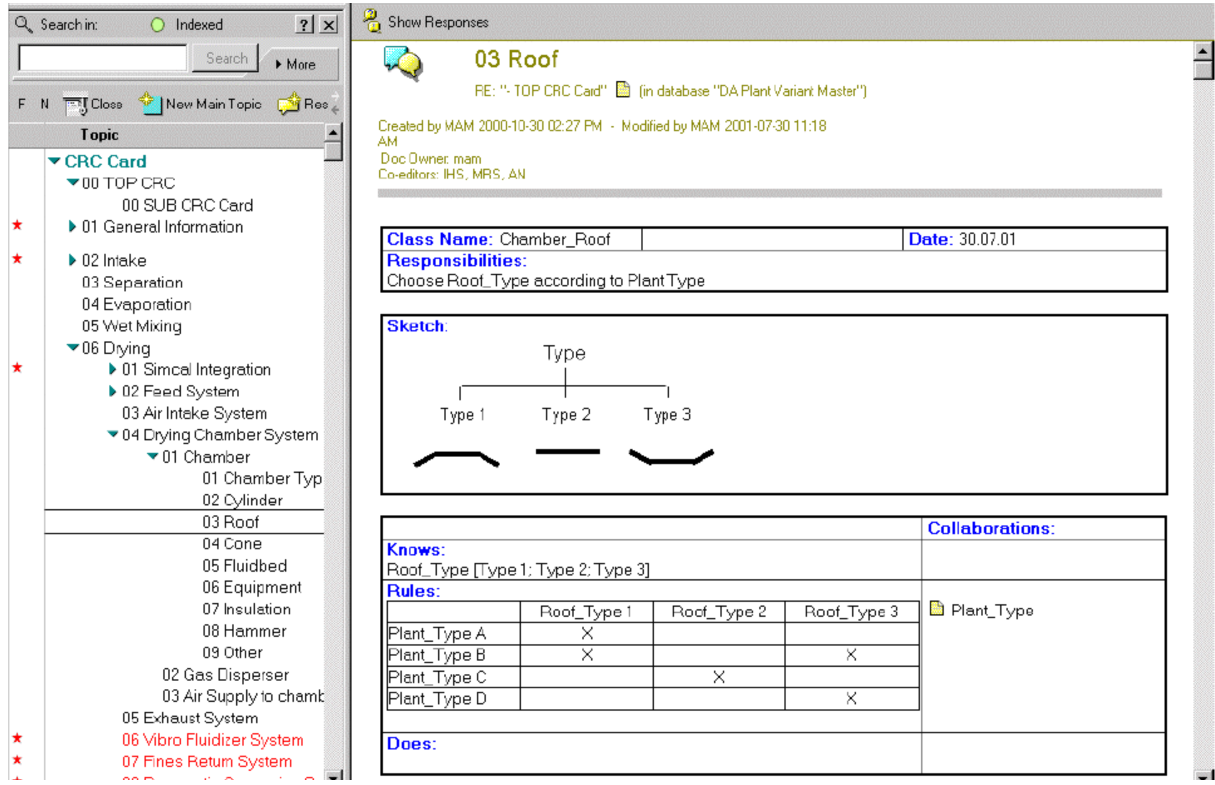


Figure 5: The documentation system of GEA Niro A/S (Hvam & Malis, 2001)

The general impression of Hvam et al. (2005a) is that the CPM-procedure can enhance the process of developing and maintaining PCS's, but that the manual documentation task requires too much effort as a model grows bigger and more complex. This means that companies settle for less comprehensive and more inadequate documentation, which could change by the presence of a documentation system that supports the CPM-procedure. Hvam et al. (2005a) therefore emphasize the need for a tool to ensure better documentation and relieve companies from the time-consuming tasks of ensuring consistent product models. To get closer to the creation of a documentation system that supports the development and maintenance of PCS's, Hvam et al. (2005a) present a requirement specification, which is inspired by the functionality of typical CASE-tools and PDM-systems (Product Data Management). The article includes a long list of requirements, but does not in a detailed manner deal with topics like user interface design, definitions of graphical notations and how to handle interrelated models. An important part of the basis for developing a documentation system to support the development and maintenance of PCS's is therefore still missing.

4 Definition of a concept for a documentation system

4.1 Method for the elaboration of a new concept

The concept for a documentation system to support the development and maintenance of PCS's, which is presented in this chapter, has been elaborated on the basis of discussions with potential users of the system and experts of software development. The companies which have provided inputs are the earlier mentioned GEA Niro and APC, who have presented their current documentation systems and participated in discussions of ideas and requirements for a new documentation system. While elaborating this paper, the Department of Informatics and

Mathematical Modelling at the Technical University of Denmark has provided feedback regarding which definitions that together with existing research would provide a satisfactory basis for development of the documentation system.

4.2 Definition of windows and navigation

Having established that the documentation system should support the use of PVM's, class diagrams and CRC-cards, the question is how this functionality should be presented to the users, i.e. which interfaces should be included. An obvious solution would be to create a user interface where the user creates PVM's or class diagrams in a way similar to when using e.g. Visio or Rational Rose, and where a CRC-card is opened by clicking on a class. This principle is very good for defining the structure of the knowledge model and should therefore be included in the documentation system. On the other hand, this kind of interface does not present the best overview of the CRC-cards included in the model. For administering CRC-cards the principle used in the documentation systems of GEA Niro and APC, as described in section 3, seems to be well suited. However, this view is also not sufficient in itself due to limitations concerning the structuring of models. The documentation system should therefore include what could be called: CRC-card view, PVM view and class diagram view.

In figure 6 the defined views and the navigation between the views are depicted.

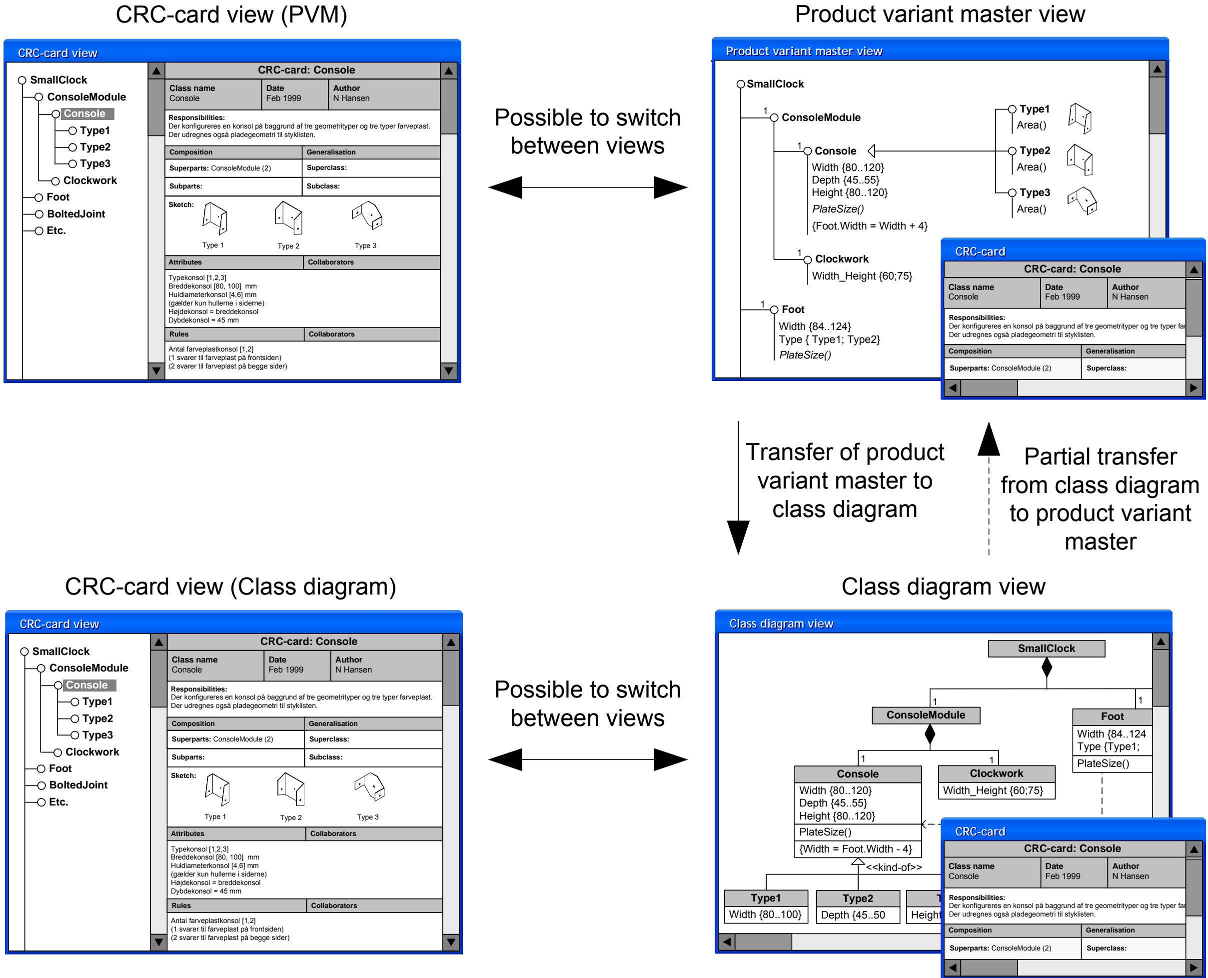


Figure 6: Windows and top level navigation in the documentation system

The documentation system should allow users to choose the model elements to be displayed in PVM's and class diagrams, for instance, choosing to hide the constraints of a class diagram.

As seen in figure 6, a transfer from class diagram to PVM is included, though this is not defined in the CPM-procedure. This functionality allows companies to go back to a more basic view, which could be beneficial if a product changes and domain experts need to be presented for a simpler representation of a model. This can, however, only be a partial transfer of model elements, since the class diagram notation is richer.

As mentioned in section 2.5, adapted use of the CPM-procedure can occur. The documentation system should therefore offer support weather PVM's or class diagrams are used as final documentation, or even both at the same time.

Besides the constraints that can be expressed graphically in diagrams, there is a need to be able to formulate further constraints. In would in most cases be advantageous to use a representation form that to some degree corresponds to the representation in the PCS. Several representation formalisms which are employed in configuration systems exist, for instance *production rules* and *constraints* (Stumptner, M., 1997; Sabin & Weigel, 1998). Since class diagrams are supported by the documentation system, it is chosen to use the terms *constraints* and *methods* in the current definitions. However, it should be possible for the users of the documentation system to rename captions in CRC-cards and type what they want in the constraints and methods fields.

The data model underneath the different views has to be common, as defined in earlier research (Hvam & Malis, 2001; Hvam et al, 2005a). A major basis for allowing the same data to be used in different models, created by using different notations, is that mappings between the notations are defined. These definitions are given in the following sections.

4.3 PVM's and matching CRC-cards

As mentioned, different definitions of the PVM technique exist, why a definition of the PVM notation to be included in the documentation system is needed. In figure 7 a PVM definition is given, which shows the minimum content to be included in the documentation system.

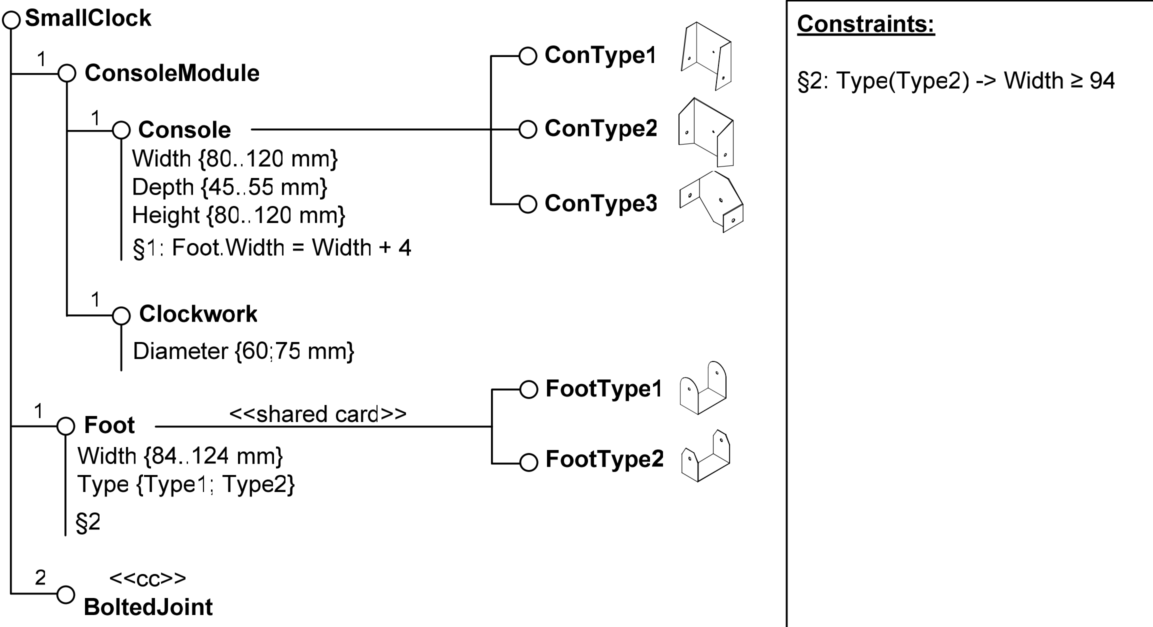
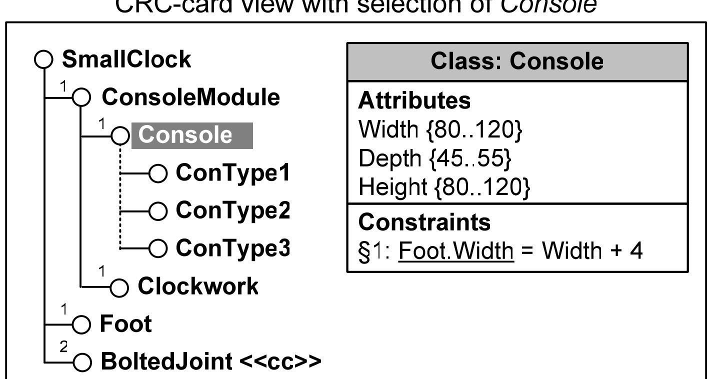


Figure 7: PVM definition

Compared to the PVM definitions in (Hvam et al., 2005b) (seen in figure 1) the following changes have been made:

- The description element within classes is found to have little use (can be found in CRC-card if needed), while consuming too much space, why this element is excluded.
- As too many constraint expressions in a PVM can lead to a confusing representation, a constraint sheet is introduced. This allows users to use constraint identifiers (e.g. §1) in the PVM and state the expressions outside the model. A constraint can therefore be expressed both in the PVM (as §1) or only be shown by a constraint identifier (as §2), while stating the expression in the constraint sheet or on the matching CRC-card.
- As specialised classes in a kind-of relationship are not always given individual CRC-cards and instead described in the generalisation class, a *shared card* stereotype (<<shared-card>>) can be applied to show if classes do not have their own CRC-card.
- A *common component* stereotype (<<cc>>) is introduced for classes and sub-models that are used several times in a model or across models, but maintained centrally.
- Methods are added to the notation to allow a differentiation between constraints and methods (though not shown in the example in figure 7). Obviously, the concept of methods can be left out of the models if this differentiation is not wanted.

Having defined the PVM-notation, the matching CRC-card view must be defined. Moving from a PVM to a hierarchical list of classes presents one basic problem, namely that a PVM includes two relationship types, while a hierarchical list itself only includes one. To include both relationship types in the hierarchical list, a full line is applied to represent part-of relationships, while a dotted line represents kind-of relationships. This is shown in figure 8 where the example corresponds to the PVM in figure 7. It should be noticed that this paper does not deal with detailed design of CRC-cards and that the ones displayed in the figures only have the purpose of illustrating certain aspects.



(Only relevant CRC-card content in this context is shown – other fields should be used according to the definitions by CPM)

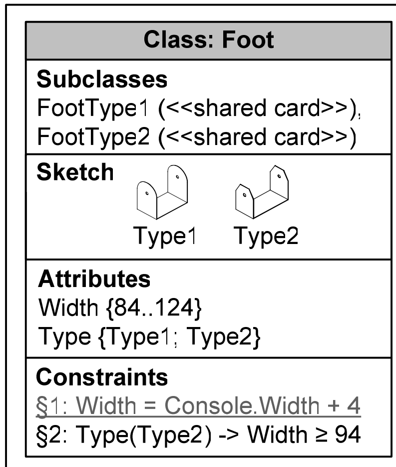


Figure 8: CRC-card view with selection of Console and CRC-card for Foot

In figure 8 it is seen that the *Console* types are shown as individual classes in the class hierarchy while the *Foot* types are not. This is because the kind-of relation from *Foot* to its subclasses is marked with the *shared card* stereotype (as seen in figure 7), contrary to the relation between *Console* and its subclasses.

In figure 8 the first constraint (§1) is placed on both cards, but with different class references and in grey text on the CRC-card for *Foot*. It is intended that this constraint should be created automatically in the CRC-card for *Foot*, which is made possible by allowing attributes from external classes to be selected from a list when creating constraints. In the CRC-card for *Console* the external class in the constraint (*Foot*) is underlined as this is intended as a hyperlink to this class. In the CRC-card for *Foot* the constraint, which is owned by *Console*, is also a hyperlink to the CRC-card for the class *Console*.

4.4 From PVM to class diagram

Class diagrams have a richer and more flexible notation than PVM's, e.g. by: including more relationship types, allowing multiple inheritance and multiple wholes. On the other hand, PVM's do not include notation that cannot be expressed in class diagrams, why defining the transfer from PVM to class diagram is quite straight forward. The only real changes are that subclasses in kind-of relations with the *shared card* stereotype are not transferred to the class diagram, and sketches do not appear in class diagrams. It would, however, be possible to include sketches within a normative use of UML by placing these inside the notes symbol.

As some might wish to remove or add classes during a transfer from a PVM to a class diagram, it should be possible to select if classes should be excluded or added during transfer.

4.5 Class diagrams and matching CRC-cards

Figure 9 depicts an example that shows the model elements to be included in class diagrams. The example corresponds to the PVM model in figure 7, but with addition of an attribute (*ManufacturingTime*) and a method (*CalcManTime*) in the class *Console*, and by adding the classes *AluminiumComp* and *OtherConstraints*.

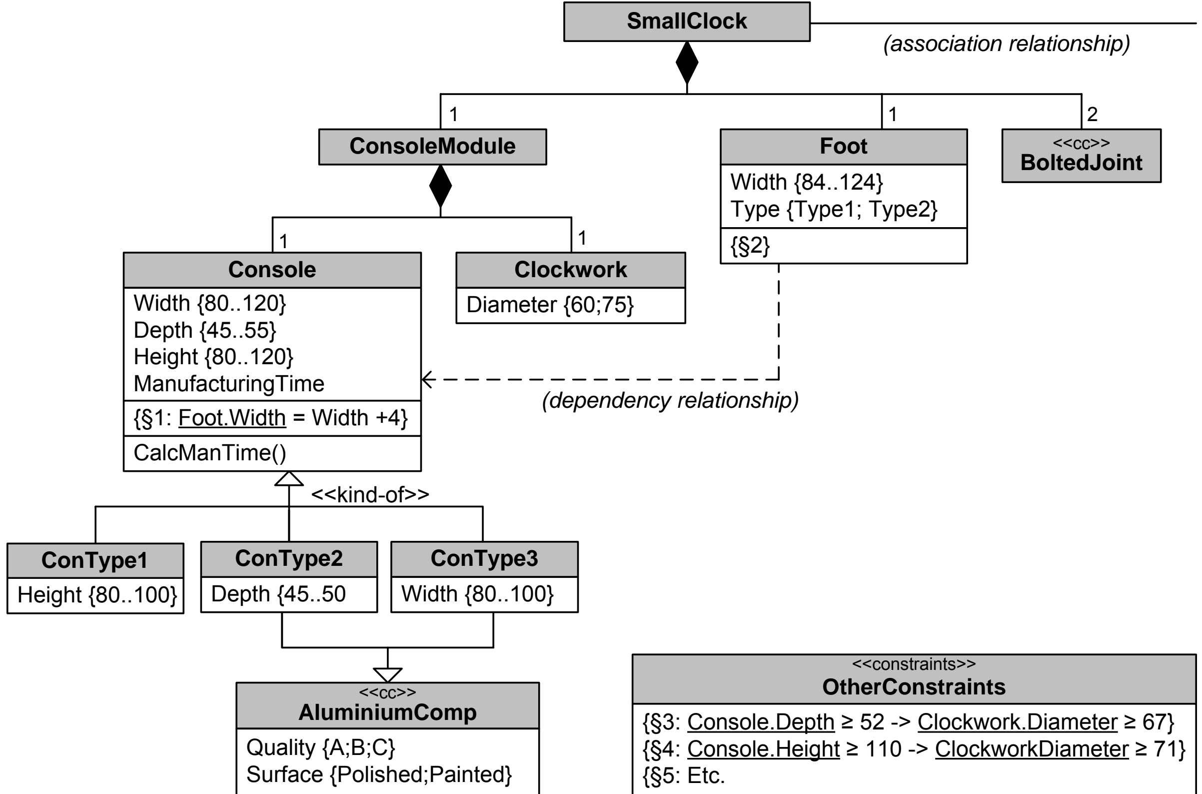


Figure 9: Class diagram definition

Compared to the class diagram definitions of the CPM-procedure the following modifications have been made:

- The dependency relationship is added to the selection of model elements used in class diagrams. This relationship type is very useful for illustrating if classes depend on others, i.e. if a class holds a constraint that affects another class, such as between *Console* and *Foot* in figure 9.
- A *kind-of* stereotype (<<kind-of>>) is added to deal with multiple inheritance in the class hierarchy section of the CRC-card view. The subclasses in a generalisation relationship with the kind-of stereotype are shown below the superclass in the main class hierarchy, while other generalisation relationships are shown below the main class hierarchy. If, on the other hand, a class has multiple wholes, the class is shown twice in the main class hierarchy.
- Besides the constraints that are formulated behind attributes, a field for holding constraints is added to the class element. As for PVM's it can be chosen only to display a constraint identifier (e.g. §2) and state the constraint on the CRC-card.
- A *constraints* stereotype (<<constraints>>) is introduced to support the placement of constraints in special classes. Where constraints can be placed in standard PCS's differ, and obviously also where the developers choose to place these. Sometimes constraints are placed in a class (or a similar construct) and sometimes elsewhere.
- As for PVM's, the *common component* stereotype (<<cc>>) can be applied for classes that are used at several different places in a model or across models, but maintained centrally.

In figure 10 the CRC-card view that matches the class diagram in figure 9 is shown. In the class hierarchy section it should be noticed that the dependency relation is not shown, but its presence appears in the matching CRC-card. Also, it is seen that the generalisation relation from *Console* to *ConType2* and *ConType3* is shown in the main model while the relation from *AluminiumComp* is shown below the model. This is because the relation from *Console* is marked with the kind-of stereotype as opposed to the relation from *AluminiumComp*.

To incorporate multiplicities into the CRC-cards, this information is stated behind relationship classes, as done for *Console* and its superclass *ConsoleModule*.

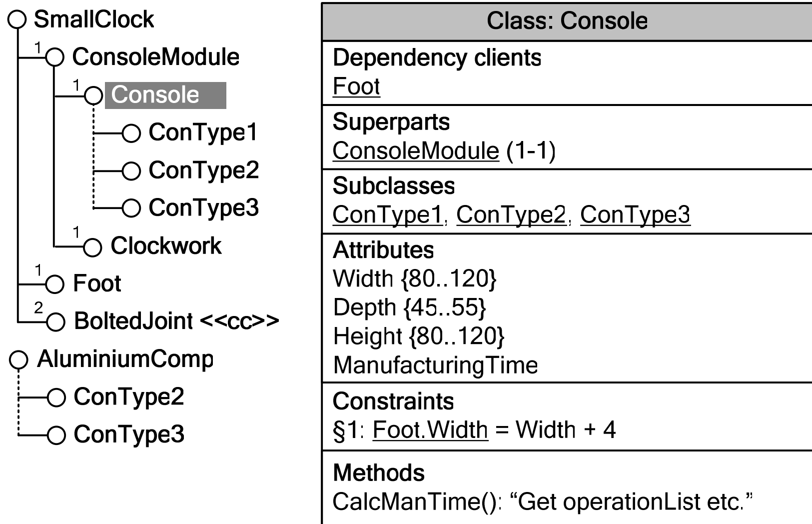


Figure 10: CRC-card view with selection of Console

In figure 11 the CRC-card view with *ConType3* selected is shown. As *ConType3* appears twice in the class hierarchy, both instances are highlighted. It is also seen that the attribute *Width* does not appear as being inherited from *Console*. This is because the inherited attribute has been overridden by defining a new value domain for the attribute.

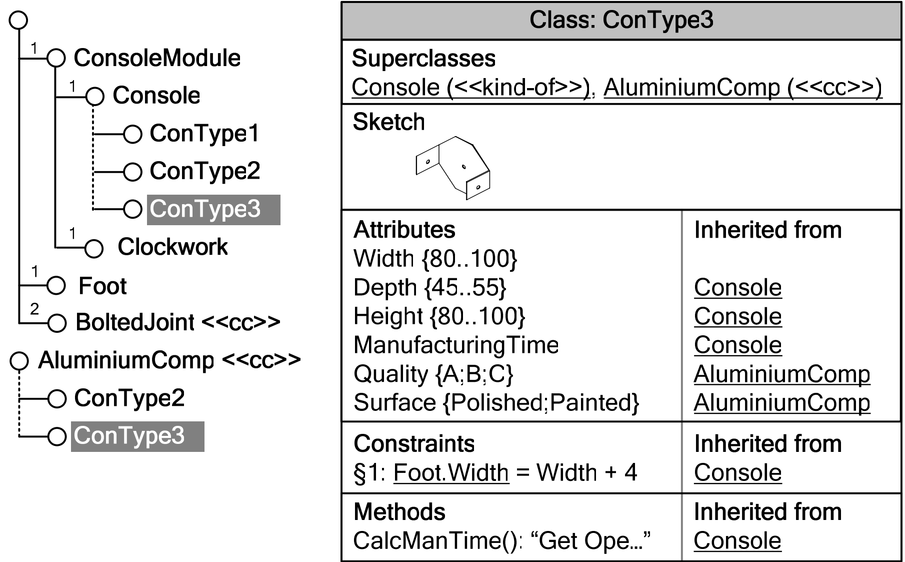


Figure 11: CRC-card view with selection ConType3

4.6 From class diagram to PVM

Contrary to the transfer from PVM to class diagram, the definition of the transfer the other way around is not that straight forward. A basic problem is that class diagrams include more relationship types and allow multiple inheritance and wholes.

Generalisation and composition relationships in class diagrams should be transferred to PVM's. In cases where a class have multiple superclasses in a class diagram, only one of the relationships can be shown in the main class hierarchy of a PVM. Using the same principle as for the class hierarchy section of the CRC-card view, the *kind-of* stereotype can be applied to show which of the relationships to include in the main class hierarchy of a PVM, while other generalisation relationships are placed below the main model.

The remaining relationship types in the defined selection of class diagram elements are association and dependency. Even though these are not a part of the PVM notation, they could in principle be shown on a PVM, though this could make the PVM representation a bit confusing. As a starting point it is chosen not to include the two relationships in the transfer from PVM to class diagram.

The transfer of model elements from a class diagram to a PVM should to a great extent be determined by the selections of the user, e.g. if a specific class should be excluded or added during this operation.

4.7 Interrelated models

A product model can consist of several interrelated sub-models, which creates a need to organise models at a higher level. These sub-models can be representations of the physical components of a product, but they can also represent different aspects. In (Hvam, 2004; Hvam et al., 2005b) a framework that defines different kinds of product models is found. The division includes:

property models, product models and models of meetings with life phase systems, which are all further subdivided into more specific kinds of models. In addition, Hvam et al. (2005b) describes three different views on a specific model: customer view, development view and production view.

To deal with grouping of models, the package notation from UML is applied. In figure 12 is outlined how package diagrams can be included in the documentation system.

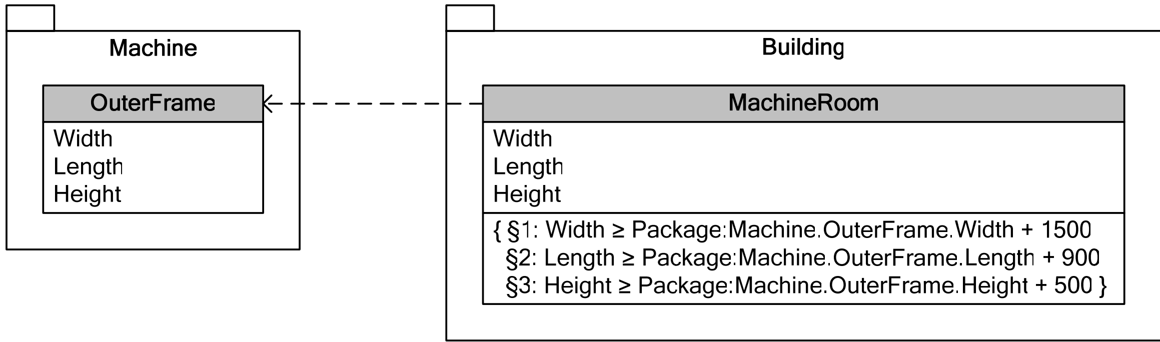


Figure 12: Package model

In figure 12 the two packages symbolise two product models, which are connected by a dependency relation between classes from each of these. In this simple example it should be imagined that there is a building model, which among other classes includes the class *MachineRoom*. The class *MachineRoom* includes constraints that ensure a machine can fit into the room. The two packages in the example would, as mentioned, consist of other classes, but if the depicted dependency relation is the only thing that connects the two packages it is adequate to show the two relevant classes.

In figure 13 a principle of how to navigate between models in the CRC-card view is outlined. In this top level view, the window *Model navigation* can be used to navigate between models, the window *Model structure* can be used to navigate between elements within a model and in the window to the right the matching CRC-cards are shown.

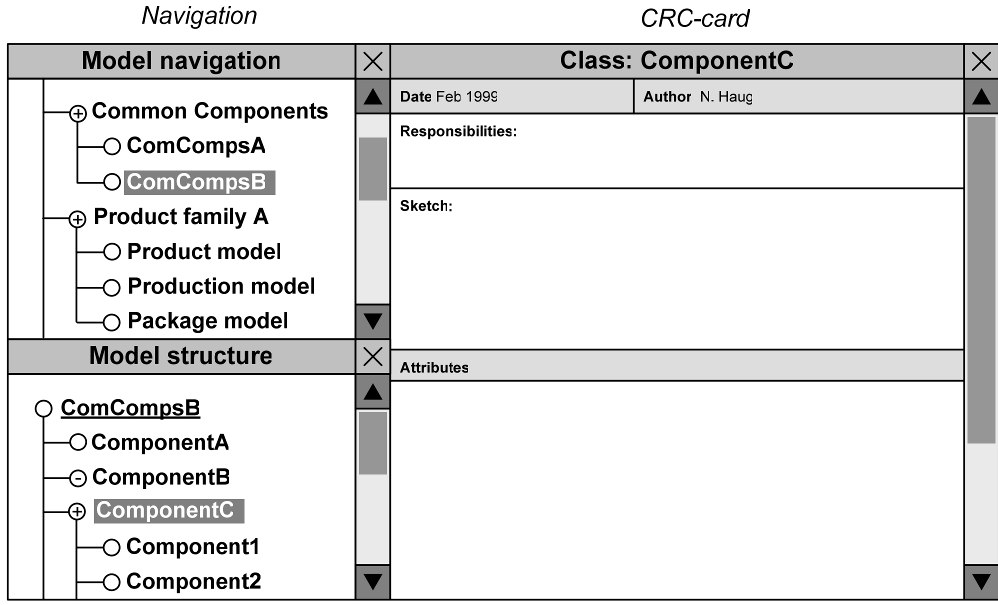


Figure 13: Top level navigation principle

The parts of the product models that are included in the package models should be imported from existing product models and be connected to the package models by the common data model. This means that if an element of a class changes in a product model, this should automatically be reflected in a package model that includes this class.

5 Conclusions

In this paper a much needed concept for how to integrate different modelling techniques in a documentation system that supports the CPM-procedure for building PCS's was presented.

The fact of not having a documentation system that supports the modelling techniques of the CPM-procedure means that different software must be applied throughout a project. It might be chosen to use MS Visio for creating PVM's, Rational Rose for class diagrams and MS Word for CRC-cards in the development phase, and finally an application like Lotus Notes in the maintenance phase. This approach implies several manual transfers of information between models without automated checks for consistency across models. A tool that supports the mentioned techniques in an integrated fashion could, therefore, ease documentation tasks considerably.

A study of earlier research shows that the documentation task is often one of the first to be cut out of a PCS project. Such a decision may later turn out to have very negative consequences, as some companies, who did not document their PCS, found themselves unable to maintain or further develop their PCS. The choice of not documenting a PCS can be seen as a consequence of that the documentation task is too time consuming. Therefore, it seems fair to assume that a documentation system, which in a user-friendly and adequately extensive way supports documentation tasks, would lead to greater documentation efforts in companies applying PCS's. This assumption should be seen in the light of that the use of a simple application developed within Lotus Notes, which only supports part of the CPM-procedure, despite its limitations, is claimed to have produced significant benefits.

Earlier research regarding how to create a documentation system to support the CPM-procedure has mainly dealt with functional requirements. However, definitions of the included techniques and in particular how these should be mapped to each other were not present in an adequate manner prior to this paper. By forwarding these specifications, an important part of the basis for the creation of a documentation system to support the CPM-procedure is laid. Still more needs to be defined regarding issues as: how to handle change requests, the definition of a common data model together with the creation and tests of prototypes. Based on the definitions presented in this paper this work is currently ongoing at CPM.

References

Arlow, J. & Neustadt, I. (2005): *UML 2 and the Unified Process* (2nd edition). Upper Saddle River, NJ: Addison Wesley, 2005.

Beck, K. & Cunningham, W.A. (1989): A laboratory for teaching object-oriented thinking. *SIGPLAN Notices*, 24 (10), 1-6.

Bennet, S., McRobb, S. & Farmer, R. (2002): *Object-Oriented Systems Analysis and Design using UML* (2nd edition). Glasgow: McGraw-Hill, 2002.

Edwards, K., Hvam, L., Pedersen, J.L., Møldrup, M. & Møller, N. (2005): *Udvikling og implementering af konfigureringssystemer: Økonomi, Teknologi og Organisation.* [Development and implementation of configuration systems: Economy, Technology and Organisation]. Final report from research project, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2005.

Felfernig, A., Friedrich, G. & Jannach, D. (2000): UML as domain specific language for the construction of knowledge based configurations systems. *International Journal on Software Engineering and Knowledge Engineering*, 10(4), 449-470.

Fowler, M. (2005): *UML Distilled* (3rd edition). Boston, MA: Addison-Wesley, 2005.

Hansen, B., Riis, J. & Hvam, L. (2003): Specification process reengineering: concepts and experiences from Danish industry. *Proceedings of the 10th ISPE international Conference on Concurrent Engineering: Research and Applications*, Madeira, Portugal, July 26-30, 2003.

Harlou, U. (2005): *Developing product families based on architectures: Contribution to a theory of product families*. Unpublished dissertation, Lyngby: Department of Mechanical Engineering, Technical University of Denmark, 2005.

Haug, A. & Hvam, L. (2005): Developing 3D Configuration Systems for manufacturers of complex building components. *Proceedings of IMCM05*, Klagenfurt, Austria, June 2-3, 2005.

Hvam, L. (1994): *Application of product modelling – seen from a work preparation viewpoint* (Trans). PhD thesis, Lyngby: Department of Industrial Management and Engineering, Technical University of Denmark, 1994.

Hvam, L. (2004): A Multi-perspective approach for the design of Product Configuration Systems - An evaluation of industry applications. *Proceedings of International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems*, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2004.

Hvam, L. & Malis, M. (2001): A Knowledge Based Documentation Tool for Configuration Projects. *Proceedings of World Congress on Mass Customization and Personalization*, Hong Kong, Oct. 1-2, 2001.

Hvam, L., Mortensen, N.H. & Riis, J. (2005b): *Produktkonfigurering*. [Product configuration]. Publication for teaching at the Technical University of Denmark, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2005.

Hvam L., Pape, S., Jensen, K.L. & Riis, J. (2005a): Development and maintenance of product configuration systems - Requirements for a documentation tool. *International Journal of Industrial Engineering*, 12 (1), 79-88.

Hvam L. & Riis J. (1999): CRC cards for product modeling. *Proceeding of the 4th Annual International Conference on Industrial Engineering Theory*, San Antonio, Texas, Nov. 17-20, 1999.

Hvam, L., Riis, J. & Malis, M. (2002): A multi-perspective approach for the design of configuration systems. *Proceedings of the 15th European Conference on Artificial Intelligence*, Lyon, France, July 21-26, 2002.

OMG (2005): *Unified Modeling Language: Superstructure* (Version 2.0: Formal/05-07-04). www.uml.org, 2005.

Pulkkinen, A. (2000): A framework for supporting development of configurable product families. *Proceedings Norddesign*, Lyngby: Technical University of Denmark, 2000.

Riis, J. (2003): *Fremgangsmåde for opbygning, implementering og vedligeholdelse af produktmodeller - med fokus på konfigureringssystemer*. [Procedure for building, implementing and maintaining product models - with focus on configuration systems]. PhD thesis, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2003.

Sabin, D. & Weigel, R. (1998): Product Configuration Frameworks - A survey. *IEEE Intelligent Systems & Their Applications*, 13(4), 42-49.

Stumptner, M. (1997): An overview of knowledge-based configuration, *AI Communications*, 10(2), 111- 125.

CRC-cards for the development and maintenance of product configuration systems

Anders Haug and Lars Hvam

Abstract

This paper presents a new definition of special CRC-cards (Class, Responsibility and Collaboration) to be used in the development and maintenance of product configuration systems (PCS).

CRC-cards are an informal and user-friendly technique for object-oriented modelling. In the early phases of a software development project, CRC-cards can be useful for coming up with design alternatives, as moving the cards around allows exploration of interactions between classes. In 1994, extended CRC-cards with the purpose of holding detailed descriptions of classes in other diagrammatic representations were incorporated into a procedure for developing PCS's. This procedure has since been applied in several configuration projects and further developed at the Centre for Product Modelling (CPM) at the Technical University of Denmark.

The proposed use of CRC-cards by CPM has been a valuable method to support the development and maintenance of PCS's for a number of companies. Investigations of two companies who apply CRC-cards to document the knowledge in their PCS's, however, showed that the contents of their CRC-cards differed from the definitions by CPM in many respects. In this paper these modifications are incorporated into a new definition of CRC-cards, which besides improving the basis for companies taking up the technique, is an important input to the project at CPM concerning the creation of a software-based documentation system that supports the development and maintenance of PCS's.

Keywords:

Product configuration, product modelling, CRC-cards, documentation of product models

1 Introduction

The use of product configuration systems (PCS) is a technology that supports the manufacturing paradigm of mass customisation (Pine et al., 1993). A PCS can be defined as a product-oriented expert system, which allows users to specify products by selecting components and properties under restriction of valid combinations. Applying PCS's can produce benefits such as: shorter lead times, reduction of resources needed to produce specifications and fewer errors in specifications (Hvam, 2004).

The development of a PCS entails a relocation of knowledge from domain experts to a software system. One of the greatest challenges in a configuration project concerns the representation of domain knowledge (Sabin & Weigel, 1998; Hansen et al., 2003). To visualise and capture domain knowledge, various diagrams can be used for creating graphical models. Diagrams for organising elements and their relations (such as class diagrams) can provide a good overview of the contents and structure of a model, but can easily become confusing if too much information is included. To avoid this information overflow, these models can be extended by special CRC-cards (Class, Responsibility and Collaboration), where more detailed information about model elements can be placed (Hvam, 2004).

Experience shows that if a PCS of a certain size is undocumented, it can be difficult or even impossible to maintain (Edwards et al., 2005). CRC-cards can be useful for documenting the knowledge in a PCS, as CRC-cards compared to the modelling environment of many PCS's can hold easily comprehensible descriptions of what is in a PCS. Having such external descriptions can ease the tasks of updating knowledge bases and tracking errors.

Several companies have applied CRC-cards for documenting the knowledge in their PCS's based on definitions of a procedure for developing PCS's from the Centre for Product Modelling (CPM) at the Technical University of Denmark. CPM's CRC-card definition provides an easily understandable basic layout that can be extended by companies according to their individual needs. This does, however, not mean that there is not a need for more extended descriptions of how to apply CRC-cards. The fact of having only a basic definition of CRC-cards can represent a problem if the standard description of how to elaborate CRC-cards is applied uncritically, as this holds risks of having redundant information in the documentation and neglecting to document important aspects.

At CPM it has for some years been an ambition to create a software-based documentation system to support the development and maintenance of PCS's. Documentation systems that support only parts of the CPM-procedure have been created (Hvam & Malis, 2001), but the ambition is to create a system which offers a much more complete support. Therefore, CPM is currently involved in a project of creating such a system, based on definitions by e.g. (Hvam et al., 2005a). For a documentation system to fulfil the needs of different companies, the existing CRC-card definitions have to be extended, e.g. by fields for change management.

This paper presents a new definition of CRC-cards, which besides providing an improved basis for companies who apply CRC-cards in their PCS projects, is an important input to the project at CPM concerning the creation of a software system to support the development and maintenance of PCS's.

This paper is organised as follows. In section 2 the use of CRC-cards in two procedures for developing PCS's is outlined. In section 3 studies two of companies applying CRC-cards to

document the knowledge in their PCS's are presented. Next, in section 4 a new definition of CRC-cards to support the development and maintenance of PCS's is presented. The paper ends with a conclusion in section 5.

2 CRC-cards for the development and maintenance of PCS's

The CRC-card technique was invented by Ward Cunningham in the late eighties (Fowler, 2005) and originally presented in (Beck & Cunningham, 1989), where it was described as a way of teaching the object-oriented way of thought. A CRC-card consists of the *class name* together with two columns for *responsibilities* and *collaborators*, as seen in figure 1. In brief, responsibilities are summarisations of the things that an object from the class should do, while collaborators are the other classes with which the class needs to work together (Fowler, 2005).

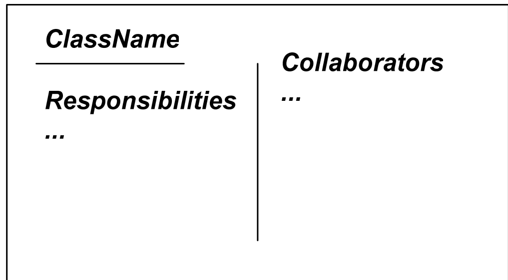


Figure 1: The original CRC- cards (Redrawn from Beck & Cunningham, 1989)

CRC-cards can be a valuable technology for coming up with good object-oriented designs, as moving the class cards around allows exploration of interactions between classes. Although CRC-cards are not part of the Unified Modelling Language (UML), they are a popular technique among skilled object designers (Fowler, 2005). CRC-cards are often used in role-playing sessions, which can be organised according to different principles (Fowler, 2005; Bellin & Simone, 1997; Bennet et al., 2002).

2.1 CRC-cards according to the CPM-procedure

Hvam (1994) presented a procedure for building product configuration systems, which has since been applied in several projects and further developed at CPM. The CPM-procedure consists of the seven phases: 1 Process analysis, 2 Product analysis, 3 Object-oriented analysis, 4 Object-oriented design, 5 Programming, 6 Implementation and 7 Maintenance.

The CPM-procedure proposes the use of three main techniques for modelling the knowledge to be included in a PCS: product variant masters (PVM), UML class diagrams and CRC-cards. In the product analysis phase, PVM's are prescribed for describing the product assortment. Later in the object-oriented phases, the contents of PVM models can be formalised and extended by using class diagrams. The basic argument for including both diagrams is that PVM's seem to be more user-friendly, which can be beneficial in modelling tasks that include domain experts with limited modelling skills. On the other hand, class diagrams are richer and more formalised, which makes these better suited for describing what should be implemented in a PCS.

The purpose of CRC-cards in the CPM-procedure is to hold detailed information about classes represented in PVM's and class diagrams, as illustrated in figure 2. For descriptions of PVM's and class diagrams, according to the definitions of the CPM-procedure, see e.g. (Hvam et al., 2002; Hvam, 2004).

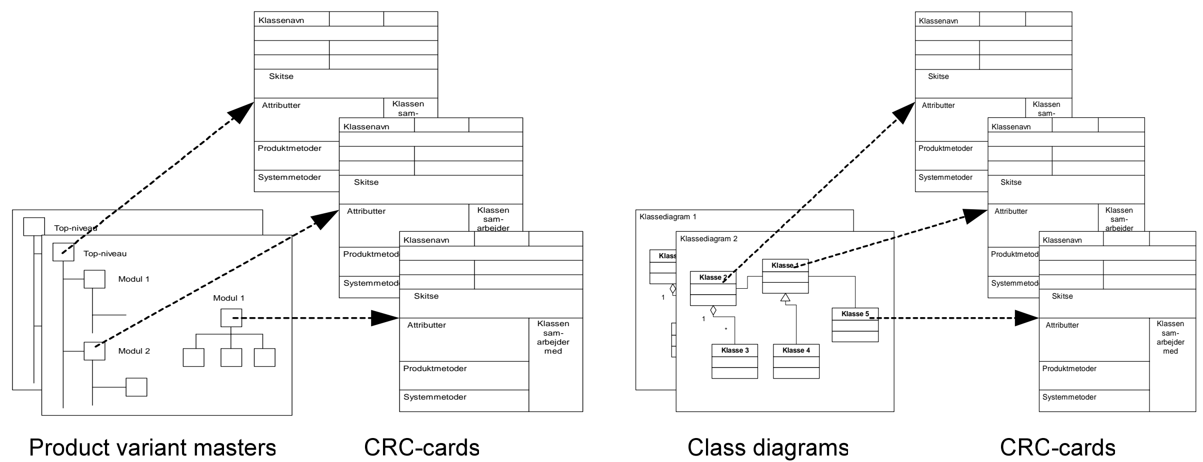


Figure 2: Diagrams connected to CRC-cards (Hvam et al., 2005b)

The CRC-cards defined in (Hvam, 1994) are compared to (Beck & Cunningham, 1989) extended by fields to describe *superclasses/subclasses* (generalisation-specialisation relationships) and *superparts/subparts* (whole-part relationships). Furthermore, *responsibilities* and *collaborations* are divided into two sections called *Knows* and *Does*, which in UML terminology are basically other names for *attributes* and *methods*.

Later, Hvam and Riis (1999) presented a revised definition of the CRC-cards to be used in PCS projects. Compared to (Hvam, 1994), the proposed definition includes the addition of the fields: *Author*, *Date*, *Sketch* and *Object mission.* This CRC-card definition is seen in figure 3.

Object no.	**Object name**	**Date**	**Author**
Object mission			
Superparts:		**Superclass:**	
Subparts:		**Subclass:**	
Sketch:			
Responsibilities			**Collaborations**
Object knows			
Object does			

Figure 3: CPM CRC-cards (Redrawn from Hvam, 1999)

The CRC-cards were further described by Hvam et al. (2003), where the field *Object mission* is renamed *Responsibilities* and the caption *Responsibilities* to *Know/Does.* In the latest definition from Hvam et al. (2005b) the field *Version* is included, and the fields *Knows* and *Does* are replaced by: *Attributes*, *System methods* and *Product methods*, where the product methods are divided into internal and external. This CRC-card definition is seen in figure 4.

<table>
<tr><th>Class name:</th><th>Date:</th><th>Author/version:</th></tr>
<tr><td colspan="3">Responsibilities:</td></tr>
<tr><td>Aggregation</td><td colspan="2">Generalisation</td></tr>
<tr><td>Superparts:</td><td colspan="2">Superclasses:</td></tr>
<tr><td>Subparts:</td><td colspan="2">Subclasses:</td></tr>
<tr><td colspan="3">Sketch:</td></tr>
<tr><td>Attributes:</td><td colspan="2">Collaborators:</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>System methods:</td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>Product methods:</td><td colspan="2"></td></tr>
<tr><td>Internal methods:

External methods:</td><td colspan="2"></td></tr>
</table>

Figure 4: The latest CRC-card definition by CPM (Translated from Hvam et al., 2005b)

On the CRC-card in figure 4, the methods fields are intended to hold information about both methods and what in other contexts is referred to as *constraints* or *(production) rules* (Stumptner, M., 1997; Sabin & Weigel, 1998). *Product methods* concern knowledge about products and their life phase properties, while *system methods* concern software aspects of a PCS. *Internal methods* concern the internal structure, functions and properties of a class, while *external methods* concern interfaces to others classes (Hvam et al., 2005b).

2.2 A procedure for conceptual modelling of product families

Mortensen et al. (2000) propose a five phase procedure for conceptual modelling of product families in configuration projects. The procedure is developed together with Baan Development, a developer of enterprise resource planning and configuration software. The procedure consists of the phases: 1 Identification of configuration task, 2 Identification of product family master plan, 3 Conceptual modelling of product family master plan, 4 Detailed modelling of product family master plan and 5 Modelling of product family in the configuration system.

The mean for describing the product assortment is the PVM technique (in this context called *product variant master plan*). After creating PVM models, the use of a modelling tool developed by Nielsen and Harlau (1999) is prescribed. Using this tool each class of a PVM is described in *Class Description cards* (CD-cards). Later *Class Responsibility cards* (CR-cards) are applied. CR-cards have the same structure as CD-cards, but instead of modelling independently of a PCS, the language from the configuration system is now used. An extract from the CD/CR cards is shown in figure 5.

Class Description Card		CD-Card
Class name:	Responsibility:	Status:
Class number:	Date:	☐ Working ☐ Final

List of aggregation classes:
List of inheritance classes:

Function and picture:

Figure 5: Extract of CD/CR-cards (Mortensen, et al., 2000)

The main contents of the CD/CR-cards are (Mortensen et al., 2000):

- List of aggregation classes: Describes *part-of* relations to other classes.
- List of inheritance classes: Describes the *kind-of* relations to other classes.
- Function and picture: A short description of functionality and a sketch/picture/diagram of the class.
- Defining parameters: Attributes that can be determined directly during configuration.
- Components: Elements, which the class consists of.
- Constraints within the class: Restrictions on how the components and defining parameters may be related.
- Constraints to other classes: Restrictions on how the classes can be related to other classes in the configuration system.
- Mode of action: Description of the input and output parameters for the class. Input and output can be related to the user of the configuration system and other IT systems.
- Sources: Description of the reason for the contents of the CD Card.

3 Case studies

The ways in which the companies GEA Niro A/S and American Power Conversion A/S apply CRC-cards have been investigated by the author. The two companies were chosen because of their relatively structured procedures for using CRC-cards as documentation of the knowledge in their PCS's. The investigations were carried out during 2005 and 2006, through several interviews and by analysing documentation produced by the companies.

3.1 Approach of GEA Niro A/S

GEA Niro A/S (in the following referred to as Niro) is a company with a leading market position within the area of industrial drying technology. Niro is a part of the *GEA Process Engineering Division*, which is represented in more than 50 countries and retains a total of about 3,200 employees. In Denmark, Niro employs more than 400 people.

The products of Niro can be characterised as highly individualised, meaning that much time is used for the creation of product specifications. The PCS project of Niro focuses on the quotation process and aims at: reducing lead times, reducing resources spent on making quotations, optimization of product design and formalisation of product knowledge. At the moment the PCS only supports certain products, but more will be included.

Niro applies CRC-cards for documenting the knowledge in their PCS, which includes thousands of attributes and rules spread across several product families. The CPM-procedure formed the basis for the development of the PCS, but compared to the definitions by CPM, their CRC-cards have been modified to reflect the modelling environment of their standard configuration software (Oracle Configurator) and include fields for management of changes.

Niro had implemented Lotus Notes Release 5 throughout the company as a standard application, wherefore it was chosen to base their documentation on this application, using Notes templates as CRC-cards. Besides CRC-card templates, the Lotus Notes application provides a hierarchical list of the classes, which can be used to navigate between cards. The CRC-card layout with the five folders collapsed is seen in figure 6 (a dummy class).

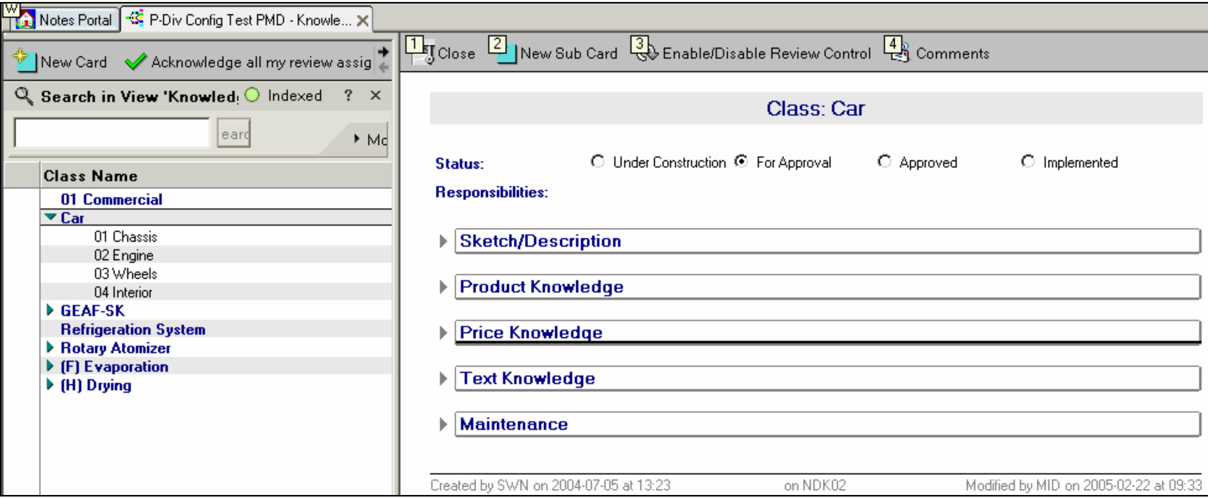


Figure 6: CRC-card layout of Niro

In the CRC-cards of Niro their knowledge is divided into the categories of: *product*, *price* and *text knowledge*. This division means that different kinds of domain experts can access the cards from different places and hereby avoid mixing their information. The three kinds of knowledge are all subdivided into *attributes* and *rules.*

The CRC-cards in the documentation system are connected by hyperlinks. For instance, if an attribute from an external class is part of a rule in a CRC-card, the CRC-card of the external class can be reached by clicking on the specific attribute. The code of the PCS is not described in the CRC-cards, as these are aimed at domain experts and therefore formulated on a higher level of abstraction.

Already implemented cards can be modified and have new elements added. This means that some elements of a CRC-card might be implemented, while others are only requested. To differentiate between elements of different states in the CRC-cards different colours are applied.

The configuration project at Niro is further described in (Hvam, 2004).

3.2 Approach of American Power Conversion A/S

American Power Conversion A/S (APC) produces data centre infrastructure such as uninterruptible power supplies, battery racks, power distribution units, cooling equipment etc. APC has sales offices throughout the world and also manufacturing facilities in several countries, including one in Denmark, where a configuration team of around thirty persons is placed. This team is responsible for the development and maintenance of the PCS's of APC. These PCS's are

used worldwide and support the elaboration of quotations and manufacturing specifications. APC's use of configuration technology has led to great benefits, e.g. reductions of lead times from weeks to hours.

The PCS's of APC include thousands of attributes and rules, and are documented in CRC-cards. Like Niro, APC uses Lotus Notes for administering their CRC-cards. Even though the CPM-procedure formed the basis for the development of their PCS's, the CRC-card layout of APC differs much from the CPM definitions. These modifications reflect the products and the modelling environment of the standard configuration software (CinCom Knowledge Builder), and include fields and functionality for management of changes.

The CRC-cards of APC fall into three categories: *product* (product descriptions), *lifecycle* (descriptions of the lifecycle of the product) and *specification* (descriptions of the documents throughout the life phases of products). Besides using CRC-cards for describing product related knowledge, APC also applies CRC-cards for describing requested changes of existing cards. Who can change a CRC-card depends on the state of the card and the rights of the user.

The CRC-cards of APC are primarily used by domain experts, wherefore the code from the PCS's is not described in the CRC-cards. Instead, this knowledge is formulated on a higher level of abstraction.

The CRC-cards contain the following top level information: *Title*, *Version*, *Category*, *Sub-Category*, *Requested Go Live Date* and *Phase*. The CRC-cards further contain the fields:

- Class Description
- Domain Expert (Product responsible)
- Link to other CRC-cards
- Sketch/Picture
- Knows (i.e. attributes) and Collaborations
- Rules and Collaborations (Rules are divided into configuration and selection rules)
- SKU's/Parts (Variants and subparts with and without product codes)
- Edit History (Who created different versions of the card)

From the CRC-cards it is possible to: request acknowledgement from a domain expert, search for CRC-cards linked to the current solution and get help to fill out the CRC-card.

The configuration project at APC is further described in (Hvam, 2004).

4 Definition of CRC-cards for the development and maintenance of PCS's

4.1 Discussion of existing designs versus empirical investigations

In spite the fact that the CPM-procedure, as mentioned, formed the basis of the PCS projects at both Niro and APC, their CRC-card layouts differ much from the CPM definitions.

In the newest CRC-card definition from CPM, methods (in this context including rules/constraints) are divided into system methods and product methods. This division does not correspond with either of the investigated companies, as they do not include the system aspect. The distinction of CPM, however, could be useful in other cases.

Besides not dividing and naming their *knowledge* according to the CPM definitions, Niro also includes a status-field that tells if a card is: *under construction*, *waiting for approval*, *approved*

or *implemented*. When using CRC-cards for documentation in the maintenance phase, it is in most cases necessary to include this type of information to be able to administer changes.

The CRC-card template of APC also differs much from the CPM definitions. Besides naming and dividing attributes and rules differently, the cards of APC: are categorised, include a requested go-live-date and have a field for stating the owner of the real product together with a functionality that allows requesting acknowledgement from this person. Like the CRC-cards of Niro, the ones of APC also include a status-field, in this case called *Phase*. Furthermore, the CRC-cards of APC include fields that reflect that the CRC-cards, besides being used for describing classes, are also used for describing change requests.

The CRC-card definition of the procedure by Mortensen et al. (2000) to some degree resembles the CPM definition, but is more loosely connected to class diagrams and closer to the Baan configurator software. This is for instance seen from that the term *parameters* is used instead of *attributes* and a field for *methods* is not included. But the main difference is the use of two types of cards, CD-cards for describing domain knowledge independently of a PCS and CR-cards for describing what should be implemented in a PCS. In the CRC-card definition provided in this paper, the primary intention of the CRC-cards is that these are used for describing the knowledge that should be (or is) implemented in a PCS, but in a manner which is comprehensible to relevant domain experts, i.e. not PCS code. This purpose is in accordance with the CPM definitions and the way the two investigated companies apply CRC-cards.

Based on the investigations of only two companies it seems to be a hopeless mission to define a standard CRC-card layout that will fit the needs of all companies without these making individual modifications. The aim of the following definitions of a new CRC-card layout is therefore to provide a basis, which to a greater degree is prepared for different kinds of use than the existing definitions.

4.2 The basic layout of the new CRC-cards

The proposed new CRC-card layout consists of some top level information together with six types of folders for class information. A one-page layout with expandable/collapsible folders is chosen, as this, contrary to a multi-page layout, allows information from any two folders to be shown on the screen simultaneously.

The proposed layout with the folders collapsed is shown in figure 7.

Class:	Status:	Change requests:	Version:
▶ Basic information			
▶ Relationships			
▶ Sketch/Picture			
▶ Knowledge group 1			
⋮ ▶ Knowledge group N			
▶ Change requests			
▶ Change history			

Figure 7: Basic layout

In the top level section, the field *Change requests* is intended as a field which tells whether there are changes that need to be implemented in the current card, i.e. *yes* or *no*. A card can therefore have the status *implemented*, even though later requests for changes have emerged.

It should be noticed that the folders called *Knowledge group 1-N* are intended to be applied for grouping of different kinds of knowledge and to be named according to preferences in individual projects.

4.3 Basic information

The *Basic information* folder holds information about: the creator of a CRC-card, who is responsible for the card, and who is responsible for the product or component that is represented by the class. In this folder also the main responsibilities of the current class can be described. In figure 8 the fields within the *Basic information* folder are shown.

Class:		Status:	Change requests:	Version:
▷ Basic information				
Created by:	Date:	Card responsible:	Product responsible:	
Responsibilities:				
▶ Relationships				
▶ Sketch/Picture				
▶ Knowledge group 1				
⋮ ▶ Knowledge group N				
▶ Change requests				
▶ Change history				

Figure 8: Basic information

Besides the fields shown in figure 8, it could in some cases be relevant to include fields that describe system interfaces.

4.4 Relationship types

The CPM-defined CRC-cards only include information about two relationship types, *generalisation* and *aggregation*, where, in this context, the latter is used without differentiation between the two types of aggregation, *aggregation* and *composition* (or *composite aggregation*). The difference between the two types of aggregation is that the composition relationship requires that a part instance is included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the part (OMG, 2005; Fowler, 2005). The properties of a composition relationship therefore correspond better to the perception of a product composed of parts, wherefore this relationship type is used in this paper.

A class diagram can also include other relationship types. In the CPM-procedure (Hvam et al., 2005b) a third kind of relationship is applied, namely *associations*. Consequently, this relationship type is included in the new CRC-card layout. Also the dependency relationship is included in the new definitions, as this can be useful for displaying that a constraint in one class affects another class. For a complete description of the relationships in class diagrams see (OMG, 2005).

Based on these observations, a basic layout for the *Relationships* folder is defined, as shown in figure 9. Here multiplicities can be stated in brackets following relationship classes.

▷ **Relationships**		
Composition	**Superparts:**	**Subparts:**
Generalisation	**Superclasses:**	**Subclasses:**
Dependency	**Sources:**	**Clients:**
Association	**Classes:**	

Figure 9: Relationships

In cases where PVM's are used for final documentation instead of class diagrams, obviously the dependency and association relationships should be left out, and instead the constraint relationship of PVM's can be added if this is used.

Felfernig et al. (2000; 2001) provide an approach for developing PCS, which does not include CRC-cards but includes class diagrams. In this approach the four mentioned relationships are applied, which are additionally extended by the UML concept *stereotypes*. If CRC-cards are applied together with class diagrams, including stereotyped relationships, then either the stereotype should be stated behind the class names in the relationship fields or the layout shown in figure 9 should be extended with additional fields.

If PVM's or class diagrams are applied together with the CRC-cards, and these diagrams describe all relationships, it would in some cases not be necessary to describe these relationships in the CRC-cards as well. However, stating relationships in the CRC-cards opens the possibility of navigating between CRC-cards without using the two structural diagrams, which could be particularly advantageous in cases with software-supported handling of CRC-cards where these are connected by hyperlinks.

4.5 Sketch/Picture

An example of the use of the field *Sketch/Picture* is shown in figure 10. Here topological variance of the subclasses is shown to the left and the placement of different measure attributes to the right.

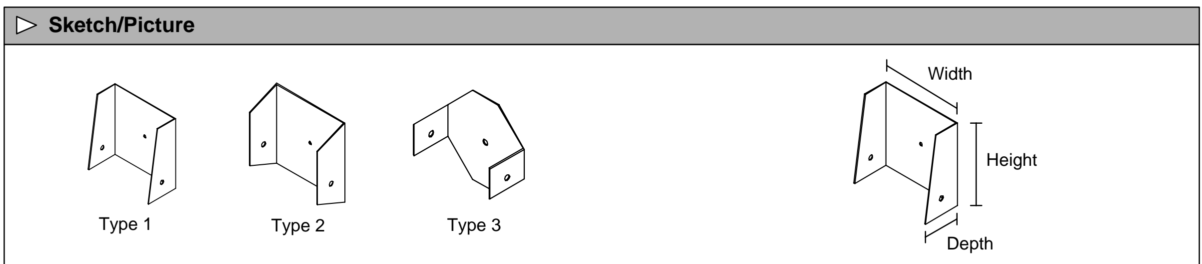


Figure 10: Sketch/Picture

4.6 Knowledge groups

In the CPM definitions and the conducted company studies there are different ways of dividing and naming: rules, constraints, methods and attributes (from now on, referred to as *class knowledge*):

- CPM definitions: Methods are divided into *system methods* and *product methods*, the latter into *internal* and *external*.

- Niro: Knowledge is grouped into *product*, *price* and *text knowledge*, which are all subdivided into *attributes* and *rules*.
- APC: Rules are divided into *configuration* and *selection rules*, and parts divided into parts with and without SKU number when describing product relationships.

The three ways of handling class knowledge differ in two respects, namely the categorisations and the subdivisions of class knowledge.

If all the defined categories of class knowledge from the two companies and the CPM definitions should be included in the CRC-card layout, this would in most cases mean that only few of these categories would be used. It therefore seems that a generic concept (as illustrated in figure 7), where this information is not pre-grouped, but can be defined by a specific company, would be a better solution. Of possible categories of class knowledge can be mentioned: product, process, visualisation, price, text, software etc.

The other question is how class knowledge should be subdivided. On the cards of Niro and APC, the different types of knowledge are divided into attributes and rules, and in the CPM definitions into attributes and methods. In the definitions provided by this paper, concepts from UML are applied in order to conform to a widespread standard. In UML, methods and constraints are not the same thing, and the concept of *rules* is not applied. Here a constraint is an assertion indicating that a condition or restriction must be satisfied by a correct design, which can be expressed by using any formalism, as long as placed within braces ({}). A method is defined as the implementation of an operation, which is a behavioural feature that specifies the name, type, parameters, and constraints for invoking an associated behaviour (OMG, 2005). For instance, an operation that retrieves data from another system would be a method, while an expression that restricts a combination of two specific components would be a constraint.

In the proposed CRC-card layout, a knowledge group is therefore divided into: attributes constraints and methods. In cases where other concepts or terms are used in the modelling environment of a PCS, such as *parameters*, *rules* or *calculations*, the fields can be changed according to preferences.

Based on the chosen definitions the basic content of a knowledge group is shown in figure 11.

▷ **Knowledge group 1**	
Attributes	
Constraints	
Methods	

Figure 11: Knowledge group

The layout in figure 11 basically lets the users formulate any kind of information within the fields. However, it might in some cases be better to specify what information is needed to ensure that adequate and relevant information is documented. Therefore, the three fields in figure 11 can be further divided, as shown in the example in figure 12.

In figure 12 it should be noticed that attributes from other classes (collaborators) are qualified by the class and underlined, as they are intended to be hyperlinks in a software-supported environment. This makes the field *Collaborators* from the existing CRC-card layout superficial, unless this is used for other purposes than describing collaborating classes.

▷ Product knowledge						
Attributes	***Name***	***Values***	***Unit***	***Type***	***Inherited***	***Comments***
	Height	30; 35; 40	mm	[Integer]		
	Width	30; 40; 50	mm	[Integer]		
Constraints	***Name***	***Description***			***Inherited***	***Comments***
	§1	Heigth ≥ Chair.Height + 300				
	§2	Type2 -> Legs.Type1 or Legs.Type3				
Methods	***Name***	***Description***			***Inherited***	***Comments***
	CalcArea ()	Height * Width				

Figure 12: Product knowledge

As seen in figure 12, the field *Inherited* is included to tell if an attribute, constraint or method is originating from another class, which can be useful information when making changes.

Besides the fields shown in figure 12, other fields could in some contexts be relevant, such as:

- Implemented (Describing the code implemented in the PCS)
- External systems collaborations (External interfaced systems, e.g. an ERP-system)
- Name in external system (Name of the field in an interfaced system)

In the mentioned approach by Felfernig et al (2001) four class stereotypes are defined: *component*, *resource*, *function* and *port*. Using these stereotypes would, if the logic of creating one CRC-card per class was obeyed, imply that some CRC-cards with sparse information are created. To avoid having more cards than necessary, it might be considered to place this kind of information within the product component classes, of which these elements are a part.

4.7 Handling of change requests

The CPM definition of CRC-cards does not include fields for management of changes, besides a version number. However, the two investigated companies do. As mentioned, Niro uses a different colour of text to indicate a requested change, while APC creates a new CRC-card, which acts as a change request. While the Niro approach seems to include some risk of errors, the APC approach on the other hand seems a bit elaborate. A new concept for handling changes in CRC-cards is therefore defined in the following.

Until now, the CRC-card definition proposed in this paper does not require real software development and could more or less be handled by e.g. MS Word templates. The proposed concept for dealing with change requests, however, requires some software development in order to function efficiently. The basic idea is that a user can click on a line or box on a CRC-card that is to be changed and then select an action: *modify*, *delete* or *add*. After this a change request, where additional change information can be stated, should automatically be created. The three kinds of change requests are illustrated in an example in figure 13, where it should obviously be possible to sort and filter these.

▷ Change requests									
Change ID	**Version**	**Requested by**	**Date**	**Action**	**Status**	**Assigned to**	**Completed by**	**Date**	**Folder**
1245		W. Haug	11/11 05	Modify	Pending	K. Larsen			Product knowledge
Exist. line	**Constraints**	§2	Type2 -> Legs.Type1 or Legs.Type3						
New line	**Constraints**	§2	Type2 -> Legs.Type2 or Legs.Type3						
Change ID	**Version**	**Requested by**	**Date**	**Action**	**Status**	**Assigned to**	**Completed by**	**Date**	**Folder**
1246	V.1.1	W. Haug	14/11 05	Delete	Implem.	K. Larsen	V. Jensen	15/11 05	Product knowledge
Exist. line	**Methods**	CalcArea ()	Height * Width						
Change ID	**Version**	**Requested by**	**Date**	**Action**	**Status**	**Assigned to**	**Completed by**	**Date**	**Folder**
1247	V.1.2	W. Haug	17/11 05	Add	Implem.	K. Larsen	K. Larsen	18/11 05	Product knowledge
New line	**Methods**	CalcVol ()	Height * Width * Depth						

Figure 13: Change requests

4.8 Change history

Based on the concept for handling change requests, the folder *Change history* is defined as a list of change records together with the history of versions. This is shown in figure 14.

▷ **Change history**

▷ **Versioning**

Version	Created by	Date	Comments
V.1.0	W. Haug	1/11 05	
V.1.1	V. Jensen	15/11 05	
V.1.2	K. Larsen	18/11 05	

▷ **Changes**

Change ID	Version	Requested by	Date	Action	Status	Assigned to	Completed by	Date	Folder
1245		W. Haug	11/11 05	Modify	Pending	K. Larsen			Product knowledge
1246	V.1.1	W. Haug	14/11 05	Delete	Implem.	K. Larsen	V. Jensen	15/11 05	Product knowledge
1247	V.1.2	W. Haug	17/11 05	Add	Implem.	K. Larsen	K. Larsen	18/11 05	Product knowledge

Figure 14: Change history

5 Conclusions

In this paper a new definition of specialised CRC-cards to support the development and maintenance of PCS's was presented.

Empirical studies of two companies showed that CRC-cards can be a valuable technique to support the development and maintenance of PCS's. The studies also showed that the current definitions of CRC-cards in many respects differed from the way in which CRC-cards were applied in praxis. The experience gained was incorporated into a new definition of a CRC-card layout to be used for the development and maintenance of PCS's.

The basic layout of the proposed CRC-card consists of top level information together with six groups of class information. One of these groups, called *Knowledge group*, is to be renamed and have multiple instances according to the preferences of a specific company. For instance, the group would in the case of one of the investigated companies have the instances: *Product knowledge*, *Price knowledge* and *Text knowledge*. The new CRC-card definition hereby offers a flexible basis that to a great extent supports the divisions of information of the CRC-cards, which are applied by the two investigated companies.

Compared to the existing CPM definitions of CRC-cards, the new layout proposed includes several new fields, e.g. for documenting additional relationship types and organising information about attributes, constraints and methods. Another main extension of the CRC-card layout concerns the handling of change requests. Contrary to the other content of the new CRC-card layout, this part requires some software development in order to function efficiently.

By incorporating the experience gained from two companies into a new definition of CRC-cards to support the development and maintenance of PCS's, an improved basis for other companies taking up the technique has been provided. The use of the new definitions could produce possible benefits such as minimising redundant information and avoiding neglecting relevant aspects in the PCS documentation. The definition of CRC-cards provided in this paper also forms an improved basis for realising the ambition of CPM to create a documentation system to support the development and maintenance of PCS's.

References

Beck, K. & Cunningham, W.A. (1989): A laboratory for teaching object-oriented thinking. *SIGPLAN Notices*, 24 (10), 1-6.

Bellin, D. & Simone, S.S. (1997): *The CRC Card Book*. Reading, MA: Addison-Wesley, 1997.

Bennet, S., McRobb, S. & Farmer, R. (2002): *Object-Oriented Systems Analysis and Design using UML* (2nd edition). Glasgow: McGraw-Hill, 2002.

Edwards, K., Hvam, L., Pedersen, J.L., Møldrup, M. & Møller, N. (2005): *Udvikling og implementering af konfigureringssystemer: Økonomi, Teknologi og Organisation.* [Development and implementation of configuration systems: Economy, Technology and Organisation]. Final report from research project, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2005.

Felfernig, A., Friedrich, G. & Jannach, D. (2000): UML as domain specific language for the construction of knowledge based configurations systems. *International Journal on Software Engineering and Knowledge Engineering*, 10(4), 449-470.

Felfernig, A., Friedrich, G. & Jannach, D. (2001): Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2), 165-176.

Fowler, M. (2005): *UML Distilled* (3rd edition). Boston, MA: Addison-Wesley, 2005.

Hansen, B., Riis, J. & Hvam, L. (2003): Specification process reengineering: concepts and experiences from Danish industry. *Proceedings of the 10th ISPE international Conference on Concurrent Engineering: Research and Applications*, Madeira, Portugal, July 26-30, 2003.

Hvam, L. (1994): *Application of product modelling – seen from a work preparation viewpoint* (Trans). PhD thesis, Lyngby: Department of Industrial Management and Engineering, Technical University of Denmark, 1994.

Hvam, L. (2004): A Multi-perspective approach for the design of Product Configuration Systems – An evaluation of industry applications. *Proceedings of International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems*, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2004.

Hvam, L. & Malis, M. (2001): A Knowledge Based Documentation Tool for Configuration Projects. *Proceedings of World Congress on Mass Customization and Personalization*, Hong Kong, Oct. 1-2, 2001.

Hvam, L., Mortensen, N.H. & Riis, J. (2005b): *Produktkonfigurering*. [Product configuration]. Publication for teaching at the Technical University of Denmark, Lyngby: Department of Manufacturing Engineering and Management, Technical University of Denmark, 2005.

Hvam L., Pape, S., Jensen, K.L. & Riis, J. (2005a): Development and maintenance of product configuration systems - Requirements for a documentation tool. *International Journal of Industrial Engineering*, 12 (1), 79-88.

Hvam L. & Riis J. (1999): CRC cards for product modeling. *Proceeding of the 4th Annual International Conference on Industrial Engineering Theory*, San Antonio, Texas, Nov. 17-20, 1999.

Hvam, L., Riis, J. & Hansen, B.L. (2003): CRC-Cards for Product Modelling. *Computers in Industry*, 50(1), 57-70.

Hvam, L., Riis, J. & Malis, M. (2002): A multi-perspective approach for the design of configuration systems. *Proceedings of the 15th European Conference on Artificial Intelligence*, Lyon, France, July 21-26, 2002.

Mortensen, N.H., Yu, B., Skovgaard, H. & Harlou, U. (2000): Conceptual modeling of product families in configuration projects. *Papers from the Workshop at the 14th European Conference on Artificial Intelligence*, Berlin, Germany, Aug. 21-22, 2000.

Nielsen, M.P. & Harlou, U. (1999): *Modelling Product Families in Configuration Systems*. M.Sc.-thesis, Lyngby: Department of Control and Engineering Design, Technical University of Denmark.

OMG (2005): *Unified Modeling Language: Superstructure* (Version 2.0: Formal/05-07-04). www.uml.org, 2005.

Pine, B.J., Victor, B. & Boynton, A.C. (1993): Making Mass Customization Work. *Harvard Business Review*, 71(5), 108-119.

Sabin, D. & Weigel, R. (1998): Product Configuration Frameworks - A survey. *IEEE Intelligent Systems & Their Applications*, 13(4), 42-49.

Stumptner, M. (1997): An overview of knowledge-based configuration, *AI Communications*, 10(2), 111- 125.