Afstudeer scriptie

Katholieke Universiteit Nijmegen

&

Pluriform Software

Afstudeeropdracht
Bedrijfsgerichte Informatica

Gert-Jan Haverkamp 9435670

Inhoudsopgave

INHOUDSOPGAVE **2**

INLEIDING **3**

PLURIFORM SOFTWARE (PS) 3
KATHOLIEKE UNIVERSITEIT NIJMEGEN (KUN) 3

PROBLEEMSITUATIE ..... ERROR! BOOKMARK NOT DEFINED.

PROBLEEMSTELLING **4**

DOELSTELLING **5**

VRAAGSTELLING **5**

HOOFDSTUK 2 WAT IS PLURIFORM SOFTWARE? **6**

HET BEDRIJF: 6
DE SOFTWARE: 6
HET VERSPREIDINGSMECHANISME 8

HOOFDSTUK 3 DATABASE VERANDERING **9**

INLEIDING 9
EVALUATIE VAN EEN SCHEMAEVOLUTIE METHODE. 9
ATTRIBUUT VERANDERINGEN 10
KLASSE VERANDERINGEN 10
ATTRIBUUT VERANDERINGEN 10
KLASSE VERANDERINGEN 10
VOORBEELD 10
FORMELE SPECIFICATIE 12
CORRECTHEID 14
VOLLEDIGHEID 14

HOOFDSTUK 4 TRANSPARENT SCHEMA EVOLUTION **16**

INLEIDING **16**

DE VERSCHILLENDE OPERATIES 17
CONTENT-BASED EVOLUTION ..... **ERROR! BOOKMARK NOT DEFINED.**

HOOFDSTUK 5 SCHEMAEVOLUTIE BIJ PLURIFORM (VERNIEUWD) **27**

INLEIDING 27
VERANDERINGEN OP KLASSE NIVEAU 27
SCHEMA OPERATIES OP EIGENSCHAPPEN NIVEAU 29

HOOFDSTUK 6 TSE VS PLURIFORM **30**

REFERENTIES **31**

TERMEN EN AFKORTINGEN **32**

MULTIVIEW EEN SAMENVATTING **33**

INLEIDING 33
BESCHRIJVING VAN DE OPERATOREN 33

Hoofdstuk 1: Inleiding

Mijn afstudeerproject heeft plaatsgevonden bij Pluriform Software (PS), een softwarehuis dat een Objectgeoriënteerd ontwikkeltool (Pluriform) heeft gemaakt. Het afstuderen heeft plaatsgevonden ter afsluiting van mijn studie bedrijfsgerichte informatica aan de Katholieke Universiteit Nijmegen.

Deze twee partijen denken beide dat er een vruchtbare samenwerking kan ontstaan. Mijn afstudeerstage is in het leven geroepen om de verschillende mogelijkheden hiervan te onderzoeken. Een probleem hierbij is dat de stage voor de universiteit van een voldoende hoog theoretisch niveau moet zijn en voor PS moet het vooral bruikbaar zijn. Mijn afstudeerproject bestond dan ook uit drie afzonderlijke delen. Er was geen enkelvoudige opdracht te vinden waar zowel PS als de KUN achter konden staan.

Het eerste gedeelte is theoretisch. Hierin heb ik geprobeerd een vergelijking te maken tussen de theoretische aanpak van de problemen die zich voordoen bij database - schemaveranderingen genaamd Transparent Schema Evolution (TSE) en de manier waarop Pluriform hiermee omgaat.

In het tweede gedeelte heb ik een splitsingsmethode ontwikkeld en geïmplementeerd van de business objecten uit de volledige Pluriform ontwikkelomgeving. Dit “kale” systeem heeft de naam Pluriform PDE (Pure Development Environment) gekregen. Het doel van dit deel was het verkrijgen van een sneller en efficiënter systeem dat ook op minder state-of-the-art computers goed gedraaid kon worden.

Het derde deel is een handleiding voor het leren modelleren van Pluriform. Deze opdracht was nodig om de samenwerking met Pluriform Software in stand te houden. Hij wordt als bijlage bij deze scriptie geleverd.

Tijdens mijn afstuderen was Patrick van Bommel mijn directe begeleider op de KUN. Bij Pluriform Software waren Tim van Hugte en Jurgen van de Donk mijn contactpersonen. Helaas is Jurgen van de Donk halverwege vertrokken naar KPMG.

Pluriform Software

PS ontwikkelt informatiesystemen waarmee organisaties veranderingen in hun bedrijfsproces beter kunnen uitvoeren en managen. PS wil graag een kleine onderneming blijven en niet als consultancyorganisatie opereren. Door deze strategische beslissing is zij de samenwerking aangegaan met organisaties die wel de gewenste consultancy capaciteit hebben (bijvoorbeeld Cap Gemini). PS heeft zelf ook een aantal klanten om hun materiedeskundigheid van specifieke primaire processen op peil te kunnen houden. Hierdoor wordt Pluriform inmiddels bij talloze soorten organisaties gebruikt, zoals voor organisaties in de bouwnijverheid, Justitie, handel- en productiebedrijven, kredietverzekeringen en non-profit organisaties.

Katholieke Universiteit Nijmegen (KUN)

De KUN is een wetenschapsgerichte organisatie die uit is op vernieuwende, innovatieve ideeën met als oogpunt het opleiden van studenten. De KUN is opgesplitst in verschillende

faculteiten. De School voor Informatica is een onderdeel van de faculteit van Wiskunde en Informatica

Deel 1 Schemaevolutie bij Pluriform

Hoofdstuk 1: Inleiding

Vanuit de universiteit is het oogpunt van schemaevolutie naar voren gekomen. Dit omdat het een goed theoretisch kader biedt en er in elke omgeving sprake is van database schema's en veranderingen daarop.

Probleemsituatie

Veranderingen aan applicaties zijn veel voorkomende, dure handelingen. Bestaande applicaties moeten veranderingen ondergaan om bij te blijven met de veranderingen in hun omgeving. [10] De kosten van dit onderhoud vormen een substantieel gedeelte van de software cyclus. Schattingen hierover lopen uiteen van 50% tot 90% [11], [12] Naarmate de applicaties groter en complexer worden groeien de kosten voor het onderhoud mee.

Ook de eisen die een gebruiker aan een applicatie stelt zullen in de loop van de tijd veranderen. En als de applicaties met behulp van een evolutionaire ontwikkelmethode worden gemaakt zullen er, zeker in het begin, veel aanpassingen gedaan moeten worden.

Het probleem met deze veranderingen is dat een kleine verandering op één plaats grote gevolgen kan hebben op een andere plaats. [14] [15]. Dit kan in de code zelf zijn, maar vaak heeft de verandering ook het inzicht van de gebruiker veranderd en wil deze nu ook op andere plaatsen veranderingen (zeker bij evolutionaire ontwikkeling).

Om competitief te blijven zal een IT-bedrijf dus flexibel moeten zijn om zo, zo effectief en efficiënt mogelijk deze aanpassingen aan applicaties kunnen doen. Een applicatie is in twee delen te splitsen: de functionaliteit en de gegevens. De functionaliteit bepaalt wat er kan en de gegevens geven aan waarmee dat gedaan kan worden. De functionaliteit ligt vast in de code en de gegevens zijn opgeslagen in een database.

Je kunt op twee niveaus naar een database kijken. Het eerste en meest gebruikte niveau is dat van de opgeslagen data. Hierin liggen de ingevoerde gegevens (tupels) van de database. Het tweede niveau is het implementatieniveau. Hierin ligt de structuur van de database. Deze structuur legt de vorm vast waarin de gegevens in de database worden opgeslagen. Veranderingen in deze structuur hebben dus directe gevolgen voor de gegevens en dus ook voor de functionaliteit die gebruik maken van deze gegevens. Het veranderen van deze structuur heet schemaevolutie

Vanuit wetenschappelijk oogpunt is er o.a. één theorie, Transparent Schema Evolution (TSE) [1] die, via verschillende schaduw databases (views) het probleem van schemaevolutie aanpakt. PS heeft het probleem opgelost door altijd met de laatste versie te werken en d.m.v. cross-reference en dataconversies ervoor te zorgen dat upgrades naar die versie zonder veel problemen verlopen.

Probleemstelling

Vanuit de probleemsituatie is de volgende probleemstelling opgesteld:

- Wat zijn de voor- en nadelen van TSE en de methode die bij PS wordt gebruikt?

Doelstelling

Het doel van dit gedeelte is het verkrijgen van een vergelijking van een methode die vanuit een theoretisch kader is ontwikkeld en een methode die vanuit de praktijk is ontwikkeld.

Vraagstelling

Uit de probleem- en doelstelling volgen de onderstaande vragen:

1) Wat is Pluriform? Speciaal aandachtspunt: hoe groot is de kracht op het gebied van re-use en schemaevolutie en hoe wordt er gewerkt met deze zaken?
2) Wat is TSE? In eerste instantie wordt gekeken naar een artikel uit IEEE Transactions on Knowledge and Data Engineering, getiteld: "A Transparent Schema-Evolution System Based on Object-Oriented View Technology."
3) Hoe verhouden (1) en (2) zich tot elkaar?

Hoofdstuk 2 Wat is Pluriform Software?

Het bedrijf

Pluriform Software (PS) is een bedrijf, gevestigd in Mariaheide. Het ontwikkelt informatie systemen volgens de RIAD (Rapid Interactive Application Development) methode en is in 1980, onder de naam Windgassen Software, opgericht. Tot 1989 heeft de onderneming zich toegespitst op een administratief (boekhoud-) pakket onder de naam Travers. Met Travers is PS tot 1989 marktleider geweest op het gebied van boekhoudpakketten voor PC's in Nederland. In 1986 is de strategische keuze gemaakt om te starten met de ontwikkeling van een Objectgeoriënteerde ontwikkelomgeving voor het bouwen van administratieve systemen gericht op re-use. Deze ontwikkelomgeving kreeg de naam Pluriform en in 1990 kwam de eerste applicatie hiervan uit. Nu is PS een volledig zelfstandige onderneming met een vijftiental medewerkers, en is de Pluriform ontwikkelomgeving uitgegroeid tot een volwassen tool met zeer veel mogelijkheden.

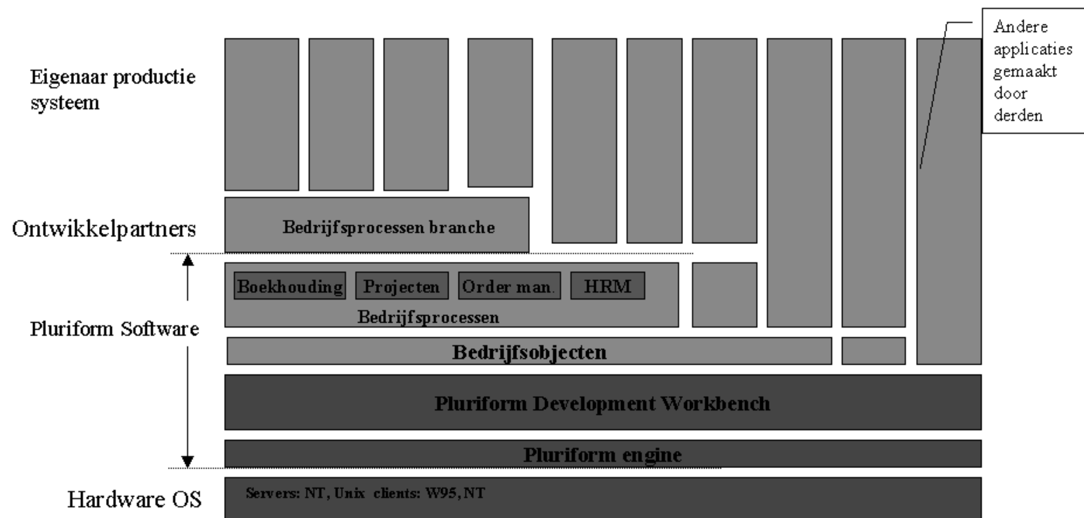
Pluriform betekent veelvormig. Dit past goed bij de tool omdat de Pluriform ontwikkelomgeving zo gemakkelijk op verschillende manieren kan worden gebruikt. De Meta Group [6] zegt: "Pluriform is zowel tool als applicatie, en richt zich hierdoor op zowel ontwikkelaars als eindgebruikers." Een groot gedeelte van de kracht van Pluriform zit hem in het feit dat de tool zichzelf ook evolutionair verder ontwikkelt. De bovenste lagen van de omgeving zijn in de tool zelf geschreven en hebben dus de voordelen die de tool zelf biedt. Maar ook het bedrijf PS is veelvormig. Het staat open voor nieuwe ideeën en probeert voorop te lopen qua ontwikkeling.

PS heeft twee soorten klanten, te weten de eindgebruikers en de partners. Bij eindgebruikers is het bedrijf verantwoordelijk voor de ontwikkeling, implementatie en evolutie (eerstelijns ondersteuning) van de gebruikersystemen. Partners vervullen bij PS de rol van Value Adding Reseller (VAR). Zij ontwikkelen, samen met Pluriform, de verschillende producten en distribueren deze op de markt. Verder implementeren zij ook zelf systemen bij eindgebruikers en verzorgen de eerstelijns ondersteuning.

De software

Objectgeoriënteerde programmeertalen zijn bij uitstek geschikt voor re-use. Een groot probleemvan re-use is dat er nog niet genoeg ervaring mee is en dat de meeste systemen òf niet voldoende functionaliteit hebben òf niet robuust genoeg zijn voor commerciële toepassingen. Pluriform denkt hiervoor de oplossing te hebben: het Pluriform Application Framework (PAF). Dit PAF is meer dan een set van bedrijfsprocessen, bedrijfsregels en bedrijfsframeworks zoals bijvoorbeeld SAP. Het is direct ontworpen voor evolutie en groei met een architectuur die hergebruik binnen systemen mogelijk maakt.

De PAF is opgebouwd uit vier verschillende lagen. (Zie figuur 2.1).



Figuur 2.1 De verschillende lagen van Pluriform

De onderste laag is de Pluriform engine. Deze C++ laag executeert de applicatie definities uit de repository en vormt een abstractielaag voor verschillende besturingssystemen, netwerken en databases. Bij executie wordt er geen code gegenereerd. De engine zelf is een executable die de beschikbare definities gebruikt om te besturen hoe de applicatie zich gedraagt en eruit komt te zien. In de repository wordt dus uitsluitend een definitie van de applicatie opgeslagen.

Boven de engine zit de Development Workbench. Deze laag bevat de algemene ondersteuning voor de applicatie segmenten. Hierin zitten een aantal applicatie ontwikkelgereedschappen zoals een scripting editor, een rapport generator, performance tuners en een distributiesysteem. De Workbench ondersteunt de gehele software ontwikkelcyclus (zowel bij klanten als binnen Pluriform zelf) met gereedschappen die zorgen voor het vastleggen van modellen, de (automatische) definitie van database schema's en het creëren van de GUI en het beheersbaar aanbrengen van wijzigingen in bestaande systemen. Zo is de dynamische cross-reference een belangrijk voorbeeld uit de Workbench.

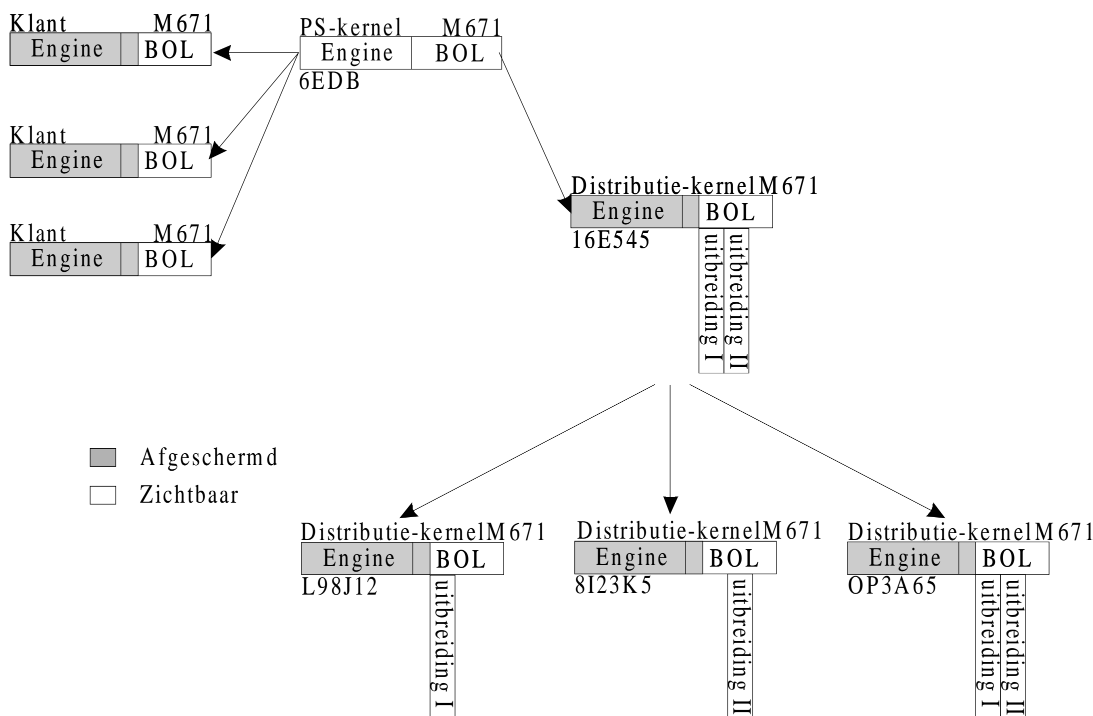
Hierop ligt de laag bedrijfsobjecten (ook wel Business Object Library (BOL) genoemd). Deze bevat een aantal verschillende applicatie segmenten. Voorbeelden hieruit zijn: Persoon, Onderneming, Artikel en Project. Momenteel bevat deze laag rond de 300 objecten.

De bovenste laag is die van de bedrijfsprocessen. Dit zijn typische high-level frameworks zoals Ordermanagement, Human Recource Management en Grootboek. Het gaat hier om bedrijfsprocessen die gebaseerd zijn op de transactie-, systeem- en abstractieconcepten volgens de Dynamic Essential Modelling of Organisations (DEMO) [8] methode. Deze methode levert een model op dat geheel onafhankelijk van de manier waarop het, zowel in organisatorische als in

informatorische zin is gerealiseerd. In dit model komen dus geen afdelingen en hiërarchische structuren voor.

Het verspreidingsmechanisme

Zoals gezegd maakt Pluriform geen nieuwe applicatie maar bouwt het intern een applicatie repository waarin de gehele structuur opgeslagen wordt. De klanten hebben dus in principe altijd het volledige Pluriform systeem. Wat er door de afzonderlijke klanten gebruikt kan worden wordt door middel van licenties geregeld. Hierin is er nog verschil tussen objecten die alleen gebruikt kunnen worden en objecten die ook kunnen worden aangepast. Er wordt dus maar eengedeelte van de BOL beschikbaar gesteld



Figuur 1.2 distributie systemen bij Pluriform

Het was tot voor kort zo dat alle klanten begonnen met een zogenaamd klant 0 systeem. Zo'n systeem bevat alle klassen en van bepaalde klassen alle of een gedeelte van zijn objecten. (Dit zijn dìe objecten die niet alleen gebruikt worden voor interne ontwikkeling) Wat hiervan door de klant gezien kan worden is afhankelijk van de verkregen licenties. Op dit "klant 0" systeem wordt dan het maatwerk gemaakt en worden de objecten gevuld. Het resultaat hiervan is het complete klantsysteem.

Nog niet zo lang geleden is Pluriform Software gebruik gaan maken van distributie systemen. Dit houdt in dat er een tussenlaag gevormd wordt met daarin een uitbreiding van de BOL (zie figuur 1.2). Deze uitbreidingen kunnen dan weer via licenties beschikbaar worden gesteld aan de uiteindelijke gebruikerssystemen. Op dit moment is het zo dat de gebruikersystemen geen afzonderlijk maatwerk meer kunnen / mogen bevatten. Maar of dit zo zal blijven staat nog niet vast.

Hoofdstuk 3 Database Verandering

Inleiding

Veranderingen aan applicaties zijn veel voorkomende, dure handelingen. Schema evolutie is onderwerp van vele discussies geweest. Veel applicaties zijn door slecht schema beheer ten onder gegaan. Hoe schemata moeten evolueren ligt niet éénduidig vast.
Er zijn verschillende manieren om met schemaevolutie om te gaan.

- Negeren, zeggen dat een programma niet aan te passen is. Dit heeft als grote voordeel dat het geen fouten oplevert. Maar is voor de rest natuurlijk onbruikbaar.
- Je kunt alles met de hand doen. Dit houdt in dat de ontwikkelaar gaat kijken waar de verandering impact heeft. En dan al deze plaatsen aan gaat passen.
- Verder zijn er verschillende manieren om het process automatisch te laten ondersteunen. Een aantal mogelijkheden is:
  - Transparent Schema-Evolution. (TSE) Dit houdt in dat er een initieel globaal schema wordt gemaakt aan het begin van de ontwikkeling dat wordt opgedeeld in verschillende gebruikersviews en wanneer er een schema verandering optreedt wordt er een kopie van die view aangepast. Zo wordt het initiële schema intact gehouden en lijkt het voor de gebruiker of zijn schema is aangepast. De methode wordt in hoofdstuk 4 in meer detail uitgelegd.
  - CEDB, dit is een methode die werkt met ontwerp checkpoints die per discipline zijn ingedeeld. Dit is meer een leidraad waarin staat waar je op moet letten. Voor meer uitleg zie [24]
  - Dynamic cross reference: dit is de ondersteuningsmethode zoals die wordt gebruikt door Pluriform. Het geeft per object een aantal karakteristieken en relaties tot andere objecten weer.
- Het weggooien van de oude applicatie en weer helemaal opnieuw beginnen.

Het beheren van veranderingen in applicaties kent twee niveaus. Het niveau van project management en het niveau van de implementatie. Project management is een activiteit van de software levenscyclus en is meer gericht op processen van de implementatie. De ondersteuning van veranderingen (change management [17] of change control [18] genaamd) op management niveau is erop gericht dat alle aanvragen tot veranderingen formeel worden aangediend en worden geëvalueerd op noodzaak en impact en het opstellen van schemata van benodigde activiteiten. De focus van dit onderzoek zit hem echter in het implementatieniveau. Dit houdt in: de methoden die de softwareontwikkelaars en –onderhouders helpen de veranderingen daadwerkelijk door te voeren en de veranderingen in te schatten (impact analysis).

Evaluatie van een schemaevolutie methode.

Voordat verschillende evolutie methodes met elkaar kunnen worden vergeleken zal er eerst een referentiemodel moeten zijn [16]. In dit model staat een beschrijving van de eigenschappen die idealiter in zo'n methode zouden zitten. Aan de hand hiervan kan er dan worden aangegeven hoe een methode hieraan voldoet en op plaatsen waar het model niet voldoet kan dan worden aangegeven wat hiervan de gevolgen zijn.

Voor alle mogelijke veranderingen kunnen er verschillende ondersteuningsniveaus zijn (van volledig handmatig naar volautomatisch). In zijn meest simpele vorm kan een vergelijking van

methodes een opsomming geven van de wel en niet ondersteunde veranderingen. Maar beter is een opsomming van de voor- en nadelen per soort verandering. De verschillende veranderingsoperaties die worden geïdentificeerd zijn gegeven in tabel 3.1

Attribuut veranderingen	**Klasse veranderingen**
• Eigenschap toevoegen • Eigenschap verwijderen • Verander het type van een eigenschap • Hernoem een eigenschap • Verplaats een eigenschap • Kopieer een eigenschap	• Voeg klasse toe • Haal klasse weg • Voeg klassen samen • Splits klasse • Hernoem een klasse • Verplaats een klasse • Voeg superklasse relatie toe • Haal superklasse relatie weg

Tabel 3.1 De verschillende veranderings operaties

Sommige van deze operaties zijn redundant. Neem bijvoorbeeld het verplaatsen van een klasse. Dit kan worden gedaan met het toevoegen en weghalen van superklasse relaties. Omdat niet alle veranderingen even vaak voorkomen kan ook nog een weging meegenomen worden aan de hand van hoe vaak een bepaalde verandering voorkomt in de praktijk. Er is nog maar weinig onderzoek gedaan naar de frequenties waarmee bepaalde veranderingen plaatsvinden. [19] geeft hiervoor echter de, hypothetische, waarden zoals weergegeven in tabel 3.2

Attribuut veranderingen		**Klasse veranderingen**	
• Eigenschap toevoegen	33%	• Voeg klasse toe	17%
• Eigenschap verwijderen	12%	• Haal klasse weg	7%
• Verander het type van een eigenschap	6%	• Voeg klassen samen	4%
• Hernoem een eigenschap	4%	• Splits klasse	3%
• Verplaats een eigenschap	3%	• Hernoem een klasse	2%
• Kopieer een eigenschap	2%	• Verplaats een klasse	2%
		• Voeg superklasse relatie toe	3%
		• Haal superklasse relatie weg	2%

Tabel 3.2 Een indicatie van frequenties van de veranderingen

Hieruit valt meteen af te lezen dat de belangrijkste operaties bij schemaevolutie het toevoegen en verwijderen van eigenschappen en klassen zijn.
Aan de hand van de opsplitsing in veranderingsoperaties en hun frequenties gerelateerd aan de impact zal er een vergelijking worden gemaakt tussen TSE en Pluriform.

Voorbeeld

Om de verschillende methodes te verduidelijken zal er gebruik gemaakt worden van een doorlopend voorbeeld. Het gaat hier om een Informatiesysteem (IS) voor een universiteit. Deze universiteit is aan het reorganiseren en wil hiervoor o.a. het IS veranderen. Dit IS houdt nu nog de volgende gegevens bij:

- **Per student:**

Achternaam	(type TEXT, moet gevuld zijn)
Voornaam	(type TEXT, moet gevuld zijn)
Geboortedatum	(type AMOUNT)
Inschrijfdatum	(type DATE)
Studenten Assistentt	(type DATE)

- **Per docent**

Achternaam	(type TEXT, moet gevuld zijn)
Voornaam	(type TEXT, moet gevuld zijn)
Geboortedatum	(type DATE)
Aanstellingsdatum	(type DATE)
Fulltime	(type BOOLEAN)

- **Per faculteit**

Code	(type TEXT, moet gevuld zijn)
Naam	(type TEXT, moet gevuld zijn)

- **Per vak**

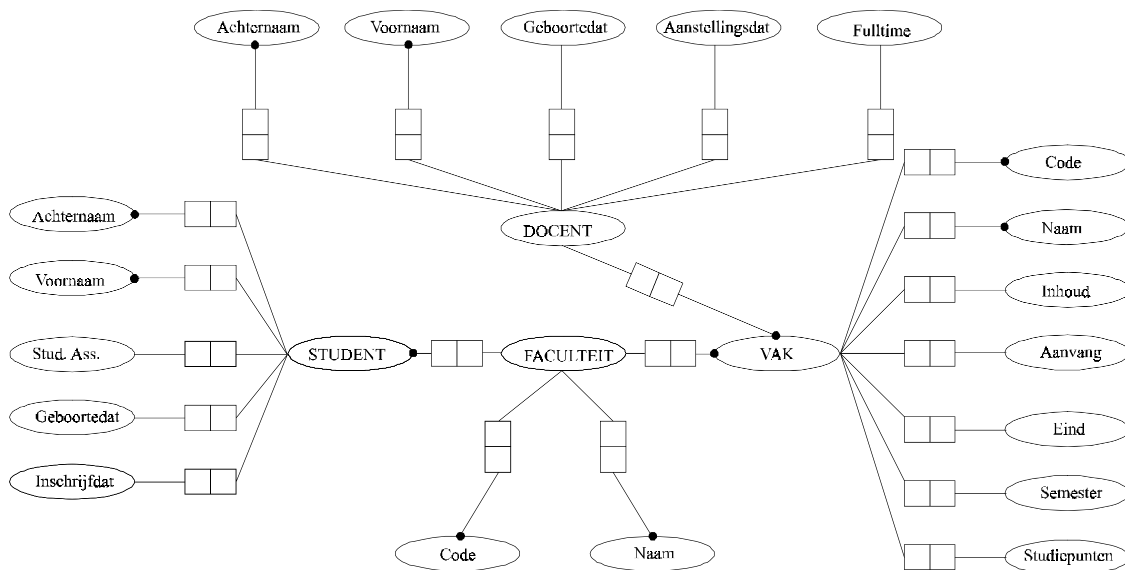
Code	(type TEXT, moet gevuld zijn)
Naam	(type TEXT, moet gevuld zijn)
Inhoud	(type TEXT)
Aanvangsdatum	(type DATE)
Einddatum	(type DATE)
Semester	(type AMOUNT)
Aantal studiepunten	(type AMOUNT)

Verder zijn er de volgende onderlinge relaties:
- Een vak wordt gegeven door één docent
- Een vak hoort bij één faculteit
- Een student hoort bij één faculteit

Figuur 3.1 geeft het NIAM schema. Dit is niet volledig, zelfs niet correct, maar wordt hier ter indicatie weergegeven. De bedoeling is dat het voorbeeld zo evolueert dat er aan het einde een goed, consistent IS overblijft.

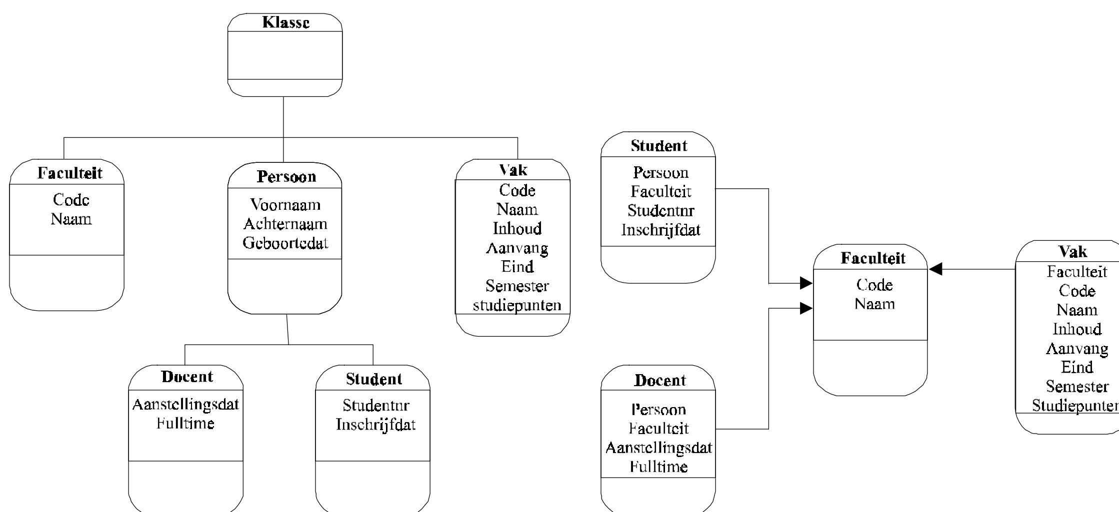
Omdat het hier gaat om Objectgeoriënteerde talen kan er aan de hand van het NIAM schema een object model worden afgeleid. De volgende klassen worden op basis van overeenkomstige relaties afgeleid:

- De klasse Faculteit, deze bevat de structuur van Faculteiten
- De klasse Persoon, deze bevat de voornaam, achternaam en geboortedatum van de verschillende personen. Deze klasse is ontstaan door de ontdekking dat deze eigenschappen voor drie klassen hetzelfde zijn.
- De klasse Docent, deze bevat de structuur van docenten.
- De klasse Student, deze bevat de structuur van studenten.
- De klasse Studenten Assistent, deze bevat de structuur van studenten assistenten. Deze is als een aparte klasse geïdentificeerd omdat uit analyse is gebleken dat een studenten assistent door de administratie gezien wordt als een apart soort student.



Figuur 3.1 NIAM schema van de studenten administratie

De hieruit volgende object hiërarchie en het object model



Figuur 3.2 De object hiërarchie (links) en het object model (rechts)

Formele specificatie.

Nu volgt een formele specificatie van schemaevolutie. Dit zal gebeuren aan de hand van een aantal axioma's. Voordat deze axioma's gegeven worden wordt er eerst een aantal basis termen en notaties vastgelegd.

- De set van alle klassen in een structuur zal worden aangegeven met T.
- Subtypering: een subtype zal worden aangegeven met $\Uparrow$, dit is dan een niet symmetrische, reflexieve en transitieve relatie. Hierin duidt $x \Uparrow y$ aan dat x een subklasse van y is en y een superklasse van x.

- Object hiërarchie: een object hiërarchie zal worden aangegeven met $L = \langle T, \Uparrow \rangle$. Dit is een gerichte a-cyclische graaf met de verschillende klassen als knopen en de subklassen relaties als gerichte paden.
- Eigenschappen: de eigenschappen van een klasse x worden aangegeven met E(x)

Uit deze basisdefinities worden de volgende termen afgeleid:

- $P(x) = \{t \in T \mid x \Uparrow t \wedge [\neg \exists_{n \ni T} \mid x \Uparrow n \wedge n \Uparrow t]\}$ De verzameling van directe superklassen van een klasse x.
- $L_{sup}(x) = \langle T_{sup}, \Uparrow \rangle$ met $T_{sup} = \{ t \in T \mid x \Uparrow t\}$ Dit is de object hiërarchie van de superklassen van klasse x. Dit bevat zichzelf en alle L_{sup} van zijn superklassen.
- $SUP(x) = \{t \in T \mid x \Uparrow t\}$ Dus de verzameling van superklassen van x.
- $SUB(x) = \{t \in T \mid t \Uparrow x\}$ De verzameling van subklassen van x.
- $H(x) = \{p \in E(x) \mid \exists_{t \in SUP(x)}[p \in E(t)]\}$ Dit is de verzameling van geërfde eigenschappen.
- $N(x) = \{p \in E(x) \mid p \text{ is locaal gedefinieerd}\}$ Dit zijn de eigenschappen waarvoor binnen de klasse een definitie is gegeven.
- $I(x) = N(x) \cup H(x)$ De interface van een klasse is de vereniging van zijn lokale en geërfde eigenschappen.

Een schema moet dan aan de volgende axioma's voldoen:

1) Volledigheid:
   $\forall_{x \in T}[\forall_{t \in P(x)}[t \in T]$ Dus alle superklassen van een klasse zitten in hetzelfde schema.
2) Geen cycles:
   $\forall_{t, x \in T}[(t \Uparrow x \rightarrow \neg x \Uparrow t) \wedge \neg x \Uparrow x]$ Een klasse mag geen superklasse zijn van één van zijn eigen superklassen.
3) Er is maar één wortel:
   $\exists!_{u \in T}[\forall_{t \in T} \mid u \in SUP(t) \wedge \#\{x \in T[u \Uparrow x]=0]$ Dus dat er precies één klasse in T is die geen superklassen meer heeft. Deze klasse wordt gedenoteerd met Δ. Dit axioma heeft een zwakke uitdrukkingskracht als je bedenkt dat de lege klasse een wortel kan zijn voor elk schema.
4) Er is maar één basis:
   $\exists!_{u \in T}[\forall_{t \in T} \mid u \in SUB(t) \wedge \#\{x \in T[x \Uparrow u]=0]$ Dus dat er precies één klasse in T is die geen subklassen meer heeft. Deze klasse wordt gedenoteerd met $\perp$. Dit axioma heeft een zwakke uitdrukkingskracht als je bedenkt dat de klasse met alle eigenschappen een basis kan zijn voor elk schema.
5) Lokale eigenschappen:
   $N(x) = I(x) - H(x)$ De lokale eigenschappen zijn die eigenschappen die niet zijn overgeërfd.
6) Overerving:
   $H(x) = \bigcup I(P(x))$ De geërfde eigenschappen van een klasse is de verzameling van interfaces van zijn directe superklassen.

Goede representatie

De bovenstaande formele specificatie is een goede representatie. Het bewijs hiervoor zal in drie stappen gaan. Dit zijn: terminatie , correctheid en volledigheid.

Terminatie

De axioma's termineren. Het bewijs hiervan berust op de aanname dat er een aftelbaar aantal klassen ($\#T < \infty$) is en per klasse een aftelbaar aantal eigenschappen ($\#(E(x)[x \in T]) < \infty$). Dit is een logische aanname omdat de verschillende klassen door de gebruiker moeten worden ingevoerd.
Uit deze aannames volgt dat P(x), N(x), H(x), I(x), SUP(x) en SUB(x) ook aftelbare verzamelingen zijn. Want $P(x) \subset T$, $N(x) \subset E$, $H(x) \subset E$, $I(x) \subset E$, $SUP(x) \subset T$, $SUB(x) \subset T$.
Er zijn ook maar een aftelbaar aantal sub-klasse relaties mogelijk vanwege axioma 2: geen cycles.
De andere axioma's lopen de hiërarchie af van een gegeven knoop naar de wortel.

Correctheid

Correctheid geeft aan dat alleen correcte schema's kunnen worden gemaakt met de axioma's. Dat de axioma's correct zijn kan als volgt worden ingezien:
Stel P(x) en N(x) zijn correct. Deze aanname is logisch omdat deze door de gebruiker worden gegeven. De eerste vier axioma's geven een conditie hierover en deze axioma's zijn onafhankelijk. Deze zijn dus per definitie correct.

Voordat axioma 5 niet correct is zal er moeten gelden dat $N(x) \neq I(x) - H(x)$. Aangezien $I(x) = N(x) \cup H(x)$ kan dat alleen maar waar zijn als er een eigenschap in N(x) zitten die ook in P(x) zit. Als kan worden bewezen dat: $\not\in_{n \in N(x)} [n \in H(x)]$ dan is de correctheid van het vijfde axioma bewezen.

Uit de definitie van N(x) blijkt dat een eigenschap in N(x) zit als er een lokale definitie voor bestaat. Een eigenschap zit in H(x) als er minstens één superklasse is waar de eigenschap ook in zit. Met axioma 2 (geen cycles) wordt bewezen dat zo'n eigenschap niet kan bestaan. Er wordt uitgegaan van een unieke identificatie per definitie.

Het bewijs van de correctheid van axioma 6 gaat met behulp van inductie.
Het axioma is correct voor de wortel van het schema: $H(\Delta)=U\ I(P(x)) = \{\}$. Stel nu dat H(x) correct is voor alle $x \in T^n$ (dus alle klassen tot diepte n) Neem nu een klasse v zo dat $v \in T^{n+1}$. H(v) is dan de verzameling van overgeërfde eigenschappen. Dit is de vereniging van de interfaces van I(x), maar deze waren al correct en dus is H(v) dan ook correct. Hieruit volgt, door middel van inductie, dat axioma 6 ook correct is.

Volledigheid

Nu zal de volledigheid van de axioma's worden bewezen. Volledigheid garandeert dat alle mogelijke schema's kunnen worden afgeleid. Het bewijs rust op de aanname dat P(x) en N(x) compleet zijn. Dit is een logische aanname omdat deze normaliter door de gebruiker worden ingegeven. Uit deze aanname volgt dat de axioma's 1 tot en met 4 volledig zijn omdat deze alleen een conditie uitspreken over T.

Axioma 5: Stel dat N(x) niet volledig is, dan zou er een lokale eigenschap p moeten zijn bij de klasse t waarvoor zou moeten gelden dat $p \not\in N(t)$ Maar aangezien p gedefinieerd is bij t zit hij in de interface van t, dus $p \in I(t)$. Verder is p niet geërfd want het is een lokale eigenschap dus $p \not\in H(t)$. Maar aangezien I(x) is gedefinieerd als $I(x) = N(x) \cup H(x)$ kan het alleen zo zijn dat $p \in I(t)$ en $p \not\in H(t)$ als $p \in N(t)$ dus de stelling is onjuist en het axioma dus volledig.

Axioma 6: (op basis van inductie) $H(\Delta)$ is compleet want $P(\Delta)=\{\}$ en dus is $H(\Delta) = \{\}$.$I(\Delta)$ is compleet want $I(\Delta)= N(\Delta)$, dit volgt uit de definitie van I(x) en $H(\Delta) = \{\}$.

Stel nu dat H(x) volledig is $\forall t \in T^n$. Neem nu een $v \in T^{n+1}$ en zie in dat P(v) volledig is. En dat I(x) compleet is voor alle $x \in P(v)$ want $x \in T^n$. Hieruit volgt dat H(v) volledig is omdat het de vereniging is van de volledige I(x).

Aangezien alle axioma's volledig zijn is ook de set van axioma's volledig.

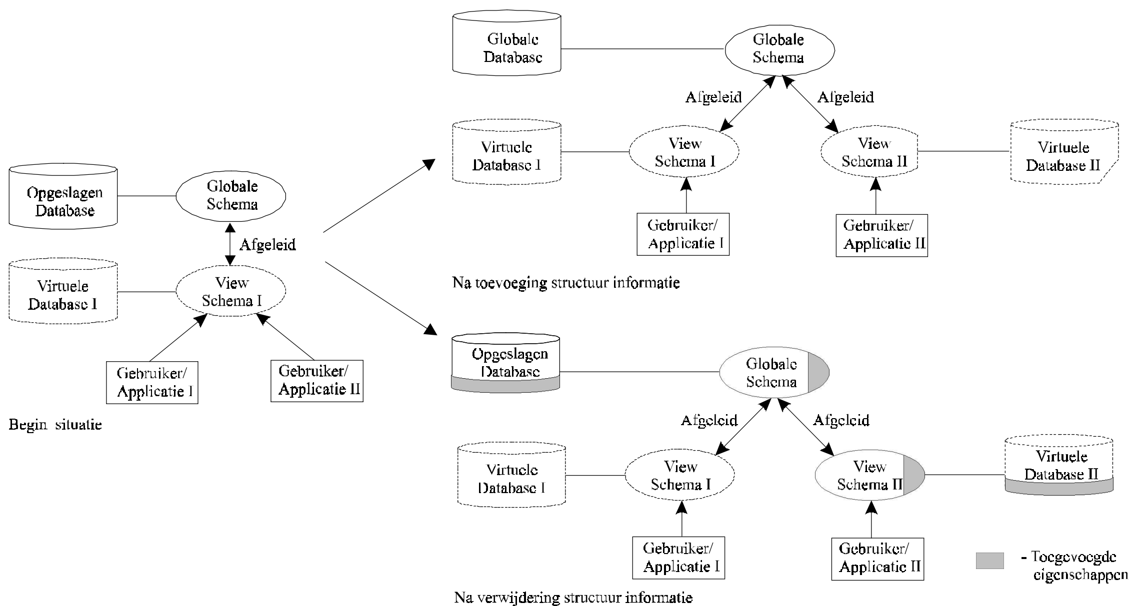
Hoofdstuk 4 Transparent Schema-Evolution

Inleiding

Eén manier om het probleem van schemaevolutie aan te pakken is via de Transparent Schema-Evolution methode (TSE). Deze methode is gebaseerd op Objectgeoriënteerde views. Een view is een gedeelte van een basis database die op een specifieke manier is opgebouwd zodat het voor de gebruiker lijkt op een complete, geheel onafhankelijk van zijn basis, nieuwe database. Onder een gebruiker kan ook een bepaalde groep van gebruikers vallen.

Het principe achter TSE is dat elke gebruiker binnen zijn eigen view werkt. Als een gebruiker zijn schema verandert dan wordt dit alleen binnen de view van die gebruiker gedaan (zie figuur 4.1). De views van de andere gebruikers wordt intact gelaten. Door dit mechanisme is het mogelijk dat verschillende gebruikers tegelijk met verschillende versies van een database werken terwijl wel van dezelfde gegevens gebruik gemaakt wordt.

Is er eenmaal een aanpassing aan een view gedaan dan kunnen ook andere gebruikers deze gemakkelijk overnemen. Dit omdat in TSE de scope van een view (schema versie) niet beperkt is tot de objecten die binnen deze view aanwezig zijn, maar ieder object in principe bereikt kan worden door iedere view. Alle objecten worden namelijk geassocieerd met, en gespecificeerd aan de hand van, één onderliggend globaal schema. Elke versie van een schema (view) is geïmplementeerd met een view specificatie die direct op deze globale database is gebaseerd.



Figuur 4.1 Views en updates hierop.

Na het ontwikkelen van het initiële schema definieert een gebruiker zijn eigen view van de database die dan dient als persoonlijke interface naar de database. Eventuele aanpassingen worden hierna op deze view gedaan. Het maakt hiervoor niet uit of deze view nou direct afgeleid is van het globale schema of dat het al om een virtueel schema gaat (subschema evolutie).

De verschillende operaties

Zoals genoemd in hoofdstuk 3, zijn er verschillende update operaties. Aan de hand hiervan volgt nu een uitleg van de werking per operatie zoals die wordt gebruikt bij TSE. Een implementatie van het algoritme zal telkens worden gegeven in (het op GemStone™ gebaseerde) MultiView [20] [21]. Dit omdat MultiView, in tegenstelling tot de meeste andere OO mechanismen een compleet virtueel schema maakt i.p.v. de verschillende virtuele klassen alleen maar af te leiden. Dit komt ten goede aan de onafhankelijkheid van de verschillende schemata en zorgt ervoor dat subschemaevolutie op dezelfde manier kan worden uitgevoerd als "gewone" schemaevolutie. De methode blijft hierdoor eenduidiger en makkelijker onderhoudbaar. Voor een korte uitleg van MultiView zie bijlage I.

Voeg eigenschap toe

Bij het toevoegen van een eigenschap moet onderscheid gemaakt worden tussen het toevoegen van een attribuut en het toevoegen van een methode. De twee verschillende algoritmen lijken wel sterk op elkaar.

Attribuut toevoegen

Een attribuut toevoegen gebeurt met de operatie: **Add_attribute** x **to** C. Met x een attribuutnaam en C een klassenaam. De operatie kijkt eerst of de nieuw toe te voegen attribuutnaam al locaal in de klasse wordt gedefinieerd. Als dit zo is dan wordt de operatie geweigerd. Is dit niet het geval dan wordt er gekeken of het attribuut al door de klasse wordt geërfd. Zo ja dan moet het domein van het geërfde attribuut compatible zijn met het domein van het nieuwe attribuut. Dat wil zeggen dat de extensies van de klasse niet ongeldig mogen worden. Gebeurt dit wel dan zijn deze instanties voor de gebruiker niet meer zichtbaar. Bestaat het attribuut nog niet dan creëert het algoritme virtuele klassen van C en zijn subklassen en voegt hieraan het nieuwe attribuut toe. De instanties van de extensie krijgen allemaal het nieuwe attribuut dat in het begin leeg is. Heeft een subklasse al een attribuut met de naam x dan stopt het algoritme daar. Het domein van het attribuut x in de subklasse wordt kleiner verondersteld dan het domein van de toegevoegde x. Is dit niet het geval dan treedt er een fout op door een meervoudig overerving conflict. TSE laat dan twee dezelfde namen toe en laat de oplossing over aan de gebruiker.

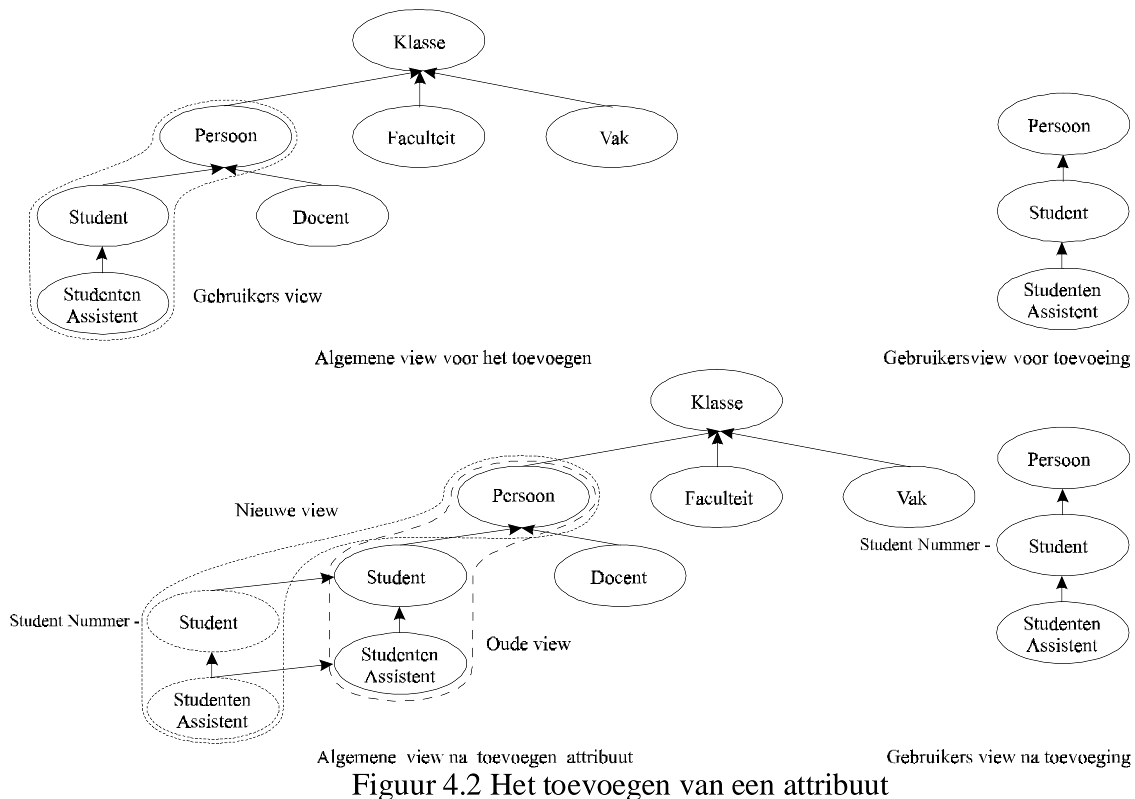
In MultiView:

```
define add_attribute x to C as
{IF x already exists in C
 THEN REJECT;
 ELSE  define VC C' as (refine x:attribute-def for C);
          set tmp:=C;
          FOR ALL subclasses (C_sub) OFCLASS tmp
          IF x NOT DEFINED FOR tmp
          THEN define VC C'_sub as (refine C':x for C_sub);
};
```

Stel je hebt de administratie van een universiteit zoals in hoofdstuk 3 gegeven is. Binnen deze administratie zit de studentenadministratie die alleen bij wil houden welke studenten er zijn en welke studenten er studenten assistent zijn. Voor deze studentenadministratie wordt dan een speciale view gemaakt die alleen bestaat uit de klassen Persoon, Student en Studenten Assistent

(Figuur 4.2 rechtsboven). Er werd nog geen studentennummer bijgehouden en de studentenadministratie wil deze eigenschap bij de klasse Studenten toevoegen. Dit kan door **add_attribute** student nummer **to** studenten te gebruiken.

Eerst wordt er een virtuele klasse gemaakt van studenten en daaraan wordt het attribuut student nummer toegevoegd. Daarna worden voor alle subklassen van student, binnen de oude view, virtuele klassen gemaakt met het nieuwe attribuut toegevoegd. De virtuele klassen die worden aangemaakt zijn subklassen van zijn basisklassen. Dit omdat deze virtuele klassen specifieker zijn (ze bezitten het nieuwe attribuut). De oude view wordt voor de gebruiker vervangen door de nieuwe. Figuur 4.2 linksonder geeft de eindtoestand weer op algemeen niveau. Figuur 4.2 rechtsonder dat op het niveau van de gebruiker.



Figuur 4.2 Het toevoegen van een attribuut

Methode toevoegen

Deze operatie lijkt op die van een attribuut toevoegen en zal verder ook niet worden uitgelegd.
In MultiView:

define add_method x **to** C **as**
{IF x already exists in C
 THEN REJECT;
 ELSE **define VC** C' **as** (**refine** x:method-def **for** C);
  set tmp:=C;
  FOR ALL subclasses (C_{sub}) OFCLASS tmp
  IF x NOT DEFINED FOR tmp
  THEN **define VC** C'_{sub} as (**refine** C':x **for** C_{sub});
};

Eigenschap verwijderen

Ook hier zal onderscheid gemaakt worden tussen het verwijderen van een attribuut en het verwijderen van een methode. Deze algoritmen zijn in wezen gelijk.

Attribuut verwijderen

Een attribuut verwijderen gebeurt met de operatie: **delete_attribute** x **from** C, met x een attribuutnaam en C een klassenaam. Om ervoor te zorgen dat regel van "volledige overerving" [1] gehandhaafd blijft kunnen we alleen die attributen weghalen waar, binnen de huidige view, C de hoogste klasse in de hiërarchie is die dit attribuut heeft, of waar C dit attribuut locaal definieert. Mocht er namelijk een klasse zijn die hoger in de hiërarchie zit en die dit attribuut heeft dan zal C, als subklasse hiervan, dit attribuut ook moeten hebben. Volledige overerving geldt alleen binnen een view want klassen buiten een view zijn non-existent binnen een view.

In MultiView:

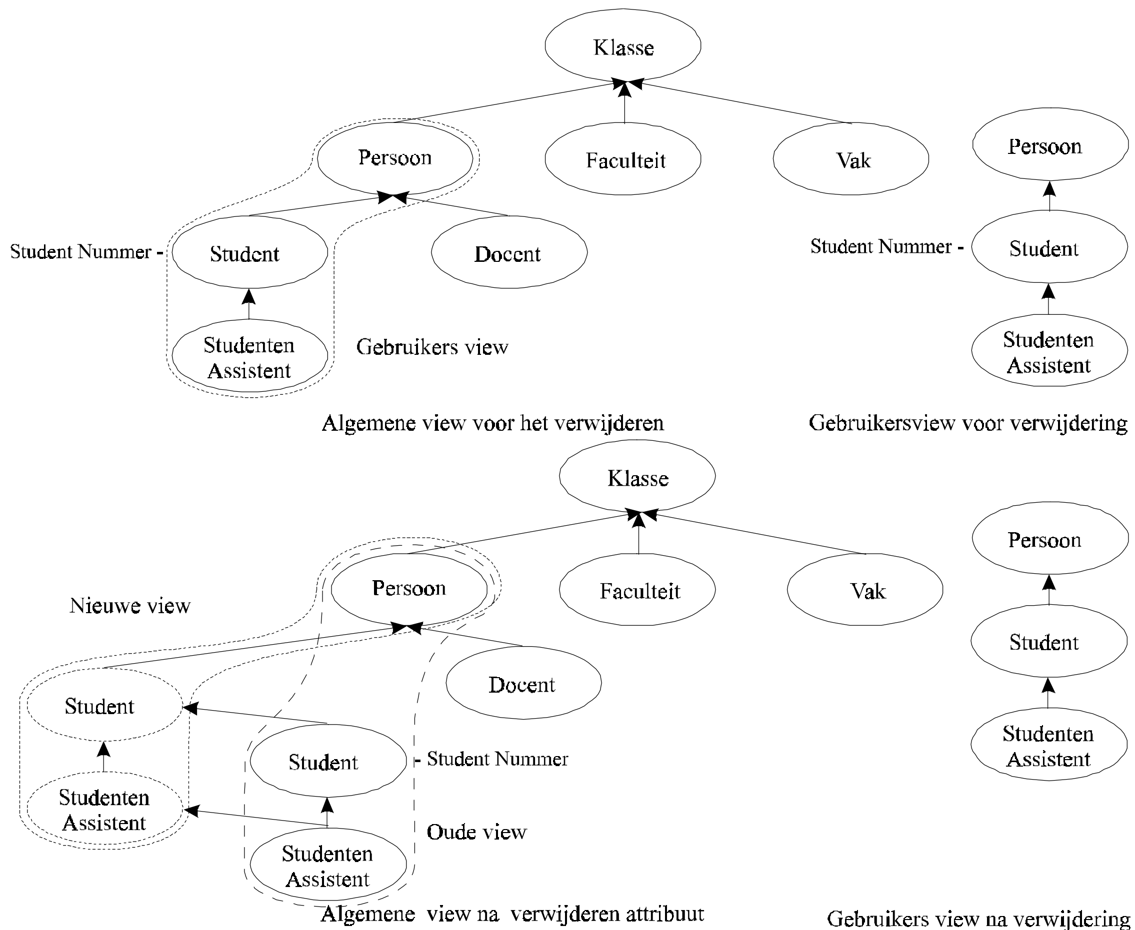
```
define delete_attribute x from C as
{FOR ALL SUBCLASSES C_sub OF CLASS C INCLUDING C
 define VC C'_sub as (hide x from sub(C));
 IF x is inherited in C
 THEN {superC := class that defines x;
        FOR ALL SUBCLASSES C_sub OF CLASS C INCLUDING C
        define VC C"_sub as (refine superC:x for C'_sub);
        }
};
```

Door het attribuut eerst binnen de klasse zelf weg te halen en dan pas te gaan kijken naar de superklassen om te zien of het attribuut daar al gedefinieerd is, wordt meteen het probleem van dubbel gedefinieerde attributen opgelost. Bij de **add_attribute** wordt er opgehouden met toevoegen als een subklasse al een attribuut heeft met die naam. Door deze manier van weghalen worden de toen gedefinieerde attributen in tact gelaten om zo de view consistent te houden.

Stel dat in het voorbeeld de studentenadministratie het attribuut 'studentnummer' niet meer mee wil nemen. Om dit te realiseren wordt **delete_attribute** studentnummer **from** student aangeroepen. Deze operator verwijdert het attribuut van de view. Het attribuut wordt echter niet weggehaald uit het onderliggende globale schema. Figuur 4.3 boven laat de beginsituatie zien. Van de klasse student wordt eerst een virtuele klasse gemaakt. Deze is een directe superklasse van zijn basisklasse. Dit omdat door het verwijderen van het attribuut zijn virtuele klasse minder specifiek is. De basisklasse krijgt alleen een superklasse relatie met zijn virtuele klasse omdat voor het globale schema de basisklasse en zijn virtuele klasse de nieuwe globale klasse omspannen. De virtuele klasse krijgt de superklassen van de basisklasse. Dit zorgt ervoor dat ook subschemaevolutie consistent zal gebeuren en dat er geen verwarring kan ontstaan over waar er super- of subklasse relaties van of naar gelegd moeten worden. Het attribuut is uit deze virtuele klasse verwijderd. Dan worden er van al zijn subklassen, binnen de oude view subklassen gemaakt waaruit het attribuut ook wordt verwijderd. Figuur 4.3 linksonder laat dit op algemeen

[1] Volledige overerving eist dat alle eigenschappen van een klasse worden overgedragen

niveau zien. Figuur 4.3 rechtsonder geeft aan hoe het er voor de gebruiker uit zag.



Figuur 4.3 Het verwijderen van een attribuut.

Methode verwijderen

Deze methode gaat parallel aan die van het verwijderen van een attribuut. Ook hier blijft de extensie van de klasse weer gelijk. We kunnen alleen die methoden verwijderen die locaal, dus binnen de view zijn gedefinieerd. Als de te verwijderen methode een andere methode met dezelfde naam overschreef, dan wordt de overschreven methode gepropageerd naar de virtuele klasse en zijn subklassen. Dit om zo de overerving consistent te houden.
In MultiView:

```
define remove_method x from C as
{FOR ALL SUBCLASSES Csub OF CLASS C INCLUDING C
 define VC C'sub as (hide x from sub(C));
 IF x is inherited in C
 THEN {superC := class that defines x;
FOR ALL SUBCLASSES Csub OF CLASS C INCLUDING C
        define VC C"sub as (refine superC:x for C'sub);
    }
}
```

Verandering van het type (domein) van een eigenschap

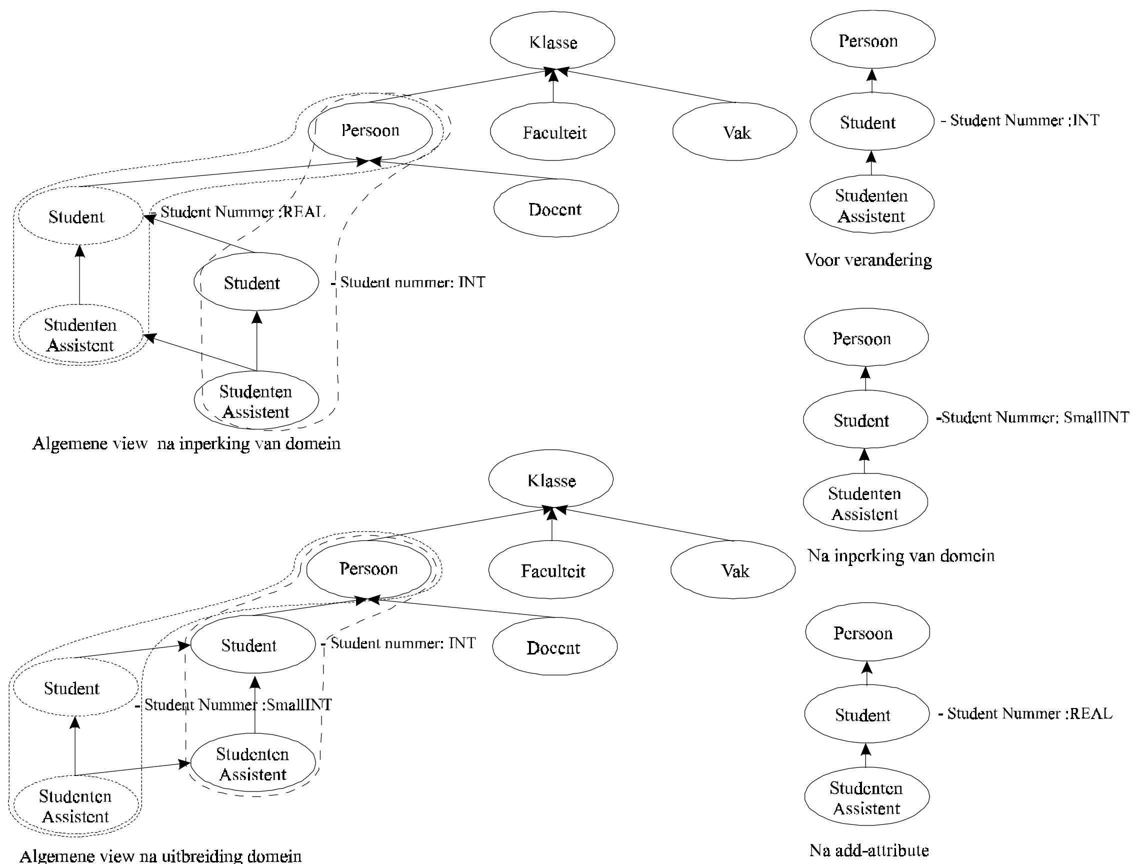
Hier hoeft alleen maar te worden gekeken naar het type van een attribuut. Een type van een methode kan namelijk alleen maar wijzigen als de gehele methode gewijzigd is. Om ervoor te zorgen dat een schema consistent blijft na een verandering van het type van een attribuut moet er onderscheid gemaakt worden tussen twee verschillende gevallen van domeins veranderingen:

- Het domein wordt uitgebreid. Dit wil zeggen dat er meer verschillende waarden van het attribuut mogelijk worden. Bij deze verandering blijven alle waarden uit de extensie valide.
- Het domein wordt gespecialiseerd. Er kunnen minder verschillende waarden worden ingevoerd. Na toepassing kan het gebeuren dat bepaalde waarden van de extensie niet meer voldoen. Deze hadden dan een waarde die nu niet meer wordt toegestaan. Deze objecten kunnen na de operatie niet meer worden opgeroepen. De gebruiker zal hier, net zoals bij de toevoeg-operatoren, de doorslag moeten geven.

In de code hoeft hiervoor geen onderscheid gemaakt te worden omdat MultiView dit zelf afvangt. Een verandering van het domein werkt net zover door in de subklasse totdat er geen subklassen meer zijn of totdat een subklasse zijn eigen domein aanpassing van het attribuut heeft. Het algoritme lijkt op dat van **Add_attribute** x **to** C met dat verschil dat er alleen een wijziging plaatsvindt
In MultiView

```
define change_domain_of x to new_domain in C as
{IF x does not exist in C
 THEN REJECT;
 ELSE define VC C' as (refine x:new_domain for C);
 set tmp:=C;
 FOR ALL subclasses (C_sub) OFCLASS tmp
        IF x not locally defined in tmp
        THEN define VC C'_sub as (refine C':x for C_sub);
};
```

Stel dat, in het voorbeeld de studentenadministratie, het toegevoegde student nummer (Figuur 4.3 linksboven) eerst als integer is opgeslagen en dat het domein gaan uitgebreid wordt door het op te slaan als REAL (Figuur 4.4 rechtsonder) Het resultaat hiervan is op algemeen niveau te zien in Figuur 4.4 linksboven. Mocht de administratie besloten hebben om het op te slaan als SMALLINT (Figuur 4.4 rechtsmidden) dan zou de globale view veranderd zijn naar Figuur 4.4 linksonder.



Figuur 4.4 domeinsverandering

Voeg superklasse toe

De operator om een superklasse toe te voegen kent aan een bepaalde klasse een nieuwe (extra) superklassenrelatie toe. Hierdoor erft deze klasse, en al zijn subklassen, de attributen en methoden van deze nieuwe superklasse. Deze operator kan voor meervoudige overervings-problemen zorgen als er in de nieuwe subklassen al eigenschappen zijn met dezelfde naam als eigenschappen uit de superklasse die niet locaal zijn gedefinieerd en die niet zijn geërfd van dezelfde superklasse als waarvan de nieuwe superklasse de eigenschap heeft geërfd. Bij TSE is het zo dat attributen en methoden met dezelfde namen overgeërfd kunnen worden naar een subklasse. Mocht dit zo zijn dan wordt de eigenschap van de subklasse intact gelaten.

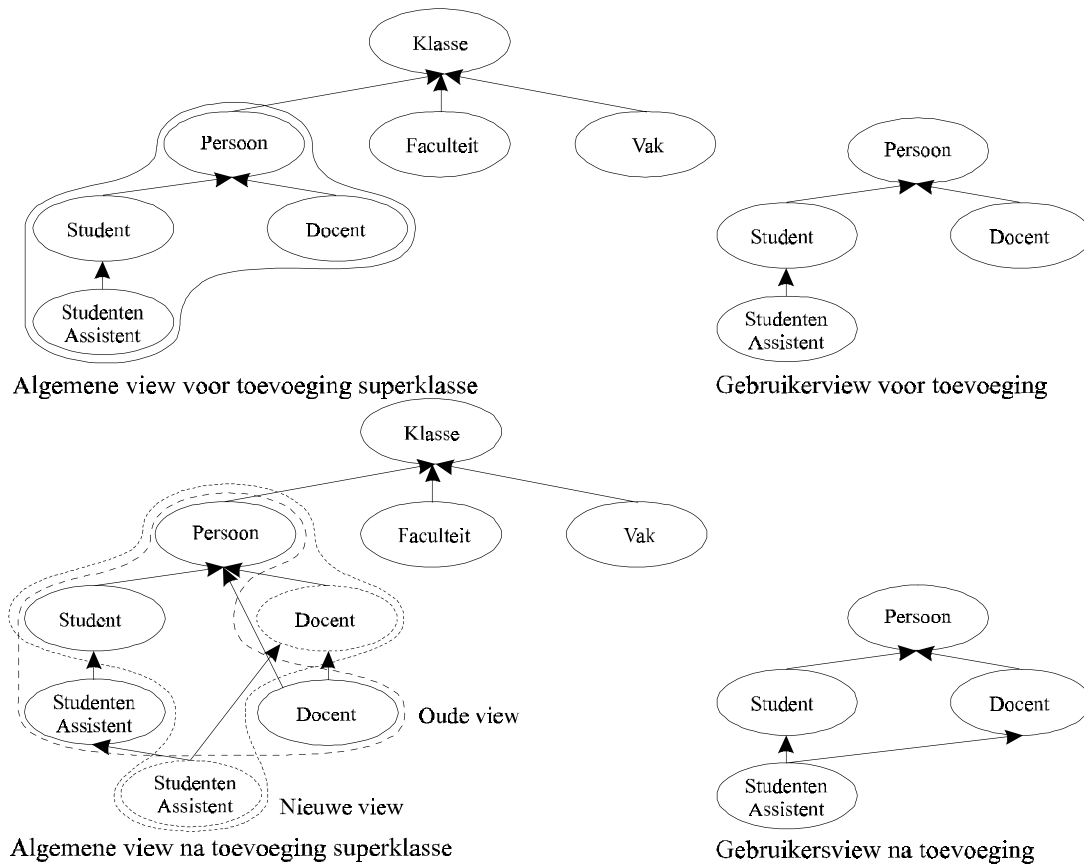
Het toevoegen van een superklasse past de extensie van een klasse zodanig aan dat de instanties van de klasse ook in zijn (nieuwe) superklasse voorkomen. Van de superklassen die nog geen superklassen waren van de nieuwe subklasse worden virtuele superklassen gemaakt waaraan de instanties van de nieuwe subklassen worden toegevoegd. Deze virtuele klassen zijn superklassen van hun basisklasse omdat ze algemener zijn. Van de subklassen die nog geen subklasse waren van de nieuwe superklasse worden virtuele subklassen gemaakt van hun basisklasse met daarin de nieuw overgeërfde eigenschappen. Deze wijzigingen worden ook toegepast op de extensies van de verschillende klassen.
In MultiView:

define add_edge C **to** C_{sup} **as**

{FOR ALL SUBCLASSES C_{sub} OF CLASS C INCLUDING C
define VC C'_{sub} **as** (**refine** properties of C_{sup} **for** C_{sub});
FOR ALL SUPERCLASSES C'_{sup} of C_{sup} that are not already superclasses of C INCLUDING C_{sup};
define VC C_{sub} **as** (**union** (C,C_{sup}));
}

Hierbij moet worden vermeld dat de **union** operator wel moet werken op de nieuwe instantie van de klasse waar de superklassenlink naar toe wordt gelegd. Dus de klasse in de nieuwe view. Dit omdat anders de verschillende views door elkaar gaan lopen.

Stel dat de studenten administratie Studenten Assistenten wil zien als docenten. Hiervoor moet er dan een superklasse relatie komen van Studenten Assistenten naar docenten. Zie figuur 4.5. De beginsituatie staat weer boven. Er wordt eerst een virtuele superklasse gemaakt van Docent. Dan wordt er een virtuele subklasse gemaakt van Studenten Assistent. Deze krijgt superklasse relaties naar de nieuwe klasse Docent en naar de klasse Student. Als laatste worden de instanties van Studenten Assistent toegevoegd aan de klasse Docent en de eigenschappen van Docent toegevoegd aan de instanties van Studenten Assistent (deze eigenschappen zijn leeg).



Figuur 4.5 Het toevoegen van een superklasse relatie.

Haal superklassenrelatie weg

Het resultaat van het verwijderen van een superklassen relatie is dat de eigenschappen van de superklassen binnen de view worden verwijderd bij die subklassen die deze, via de relatie hebben

geërfd. Dit geldt ook voor de extensies van deze klassen Mocht het zo zijn dat een klasse op een andere manier dezelfde eigenschappen van de superklasse erven (dus via een ander superklasse relatie) dan blijven deze eigenschappen wel bestaan.

Na het verwijderen van een superklassen relatie kan het zijn dat de klasse buiten de hiërarchie valt. Dit gebeurt als de klassen geen superklasse meer heeft. Bij TSE kun je bij de **delete_edge** operator een optie meegeven zodat in dat geval de meegegeven klasse de nieuwe superklasse wordt. Is deze optie niet ingevuld wordt de klassen een directe subklasse van de wortel van de hiërarchie.
In MultiView:

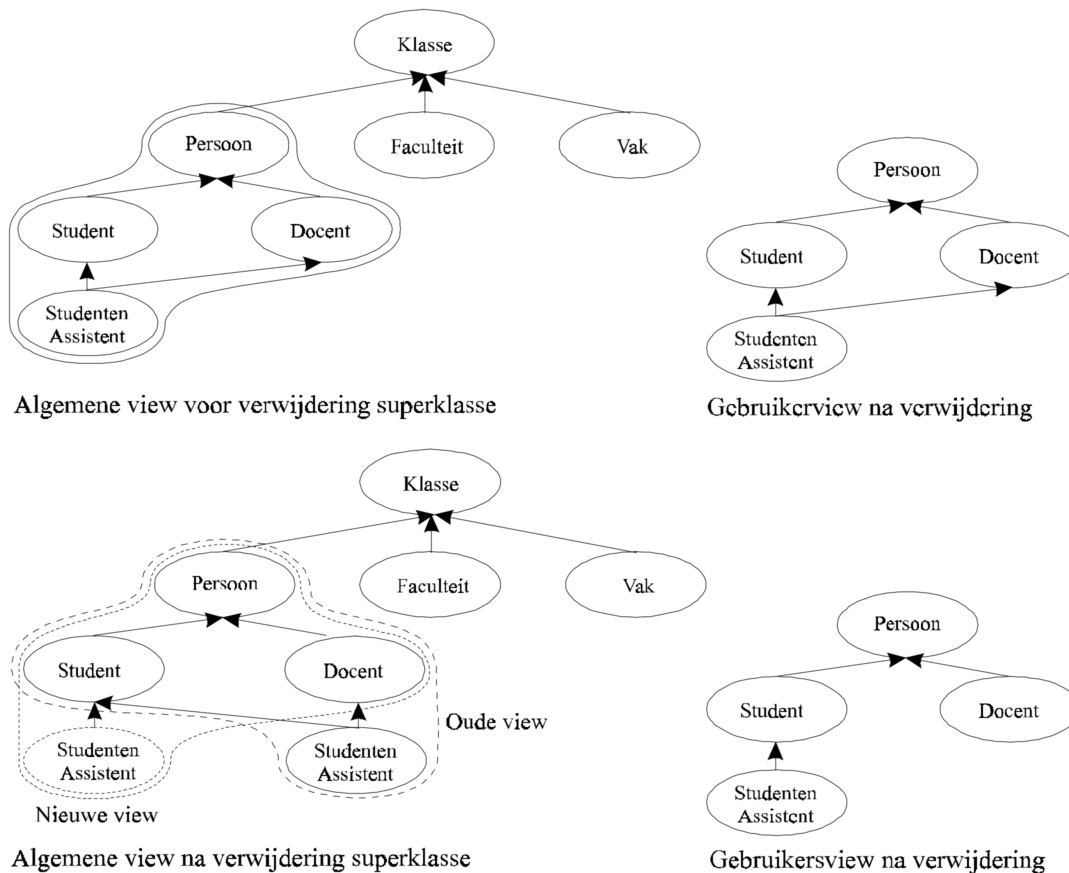
```
define delete_edge Csup, C [connected_to Cupper] as
{ FOR ALL superclasses v of Csup INCLUDING Csup that are not superklasses of C throught some other
relationship:
 {define VC x as union(commonSub (v, C, Csup-C));
        define VC v' as union(diff(v, S), X);
        }
        FOR ALL subclasses w of C' INCLUDING C:
        {y=findproperties(w, Csup-C);
        define VC w' as (hide y from w);
        }
}
```

De eerste loop van dit algoritme past de extensie van de klassen aan. Dit kan niet door zomaar de extensie van C uit de extensie van zijn superklassen te halen omdat C zijn extensies van zijn subklassen kan hebben en die subklassen kunnen via een andere relatie wel weer een subklasse van een van de superklassen van C zijn.

Wat het algoritme doet is kijken, via **commonSub**, wat de grootste gedeelde subklassen van C en v (v is de verzameling van C_{sup} en zijn superklassen) zijn, er vanuit gaande dat de superklassenrelatie C, C_{sup} verwijderd is. Dit zijn dan precies die extensies die door superklassen van C_{sup} worden geërfd. Dit gebeurt voor elke superklasse van C_{sup} die niet zelf een superklasse van C is. Aan het einde zijn alle overbodige elementen uit de extensie van de superklassen van C_{sup} gehaald.

De tweede loop zorgt ervoor dat de eigenschappen van C en zijn subklassen worden aangepast. **Findproperties** (w, C_{sup}-C) levert die eigenschappen die w alleen maar heeft geërfd door de superklassenrelatie C_{sup}-C. Deze eigenschappen worden dan, binnen de view, uit w verwijderd.

De studentenadministratie wil de studenten assistenten niet meer zien als docenten. (zie figuur 4.6) Er worden virtuele klassen gemaakt van Studenten Assistent en docent. Dan wordt er gekeken welke instanties van Studenten Assistent er zijn binnen Docent die daar niet meer horen. De eigenschappen van Docent die niet op een andere manier in Studenten Assistent zijn gedefinieerd worden verwijderd. Eén en ander wordt gepropageerd op de extensies van de nieuwe virtuele klassen.



Figuur 4.6 Het verwijderen van een superklasse relatie.

Voeg klasse toe

Als een klasse wordt toegevoegd is zijn extensie in het begin altijd leeg. Een klasse kan, bij TSE alleen maar onder aan de hiërarchie worden toegevoegd. Moet de klasse ergens anders in de hiërarchie dan zal dat door middel van de **add_edge** en de **delete_edge** operatoren moeten worden gedaan. Het type van de toegevoegde klasse is gelijk aan het type van zijn superklasse. Net als bij de **delete_edge** operator wordt een klasse direct aan de wortel verbonden als er geen andere klasse wordt aangegeven.
In MultiView:

define add-class C_{add} [**connected_to** C_{sup}]:
{FOR ALL the original classes (C_{origin}) of C_{sup}
    create a base class C_x as a direct subclass of C_{origin}';
    create a virtual class ($_{Cadd}$) based on the C_x classes by following the same
    derivation procedures of deriving C_{sub} from the C_{origin} classes;
}

Het algoritme zoekt eerst alle globale klassen waaruit C_{sup} is opgemaakt (C_{origin}) en maakt bij elk van deze klassen een klasse C_x. Dan maakt hij de virtuele klasse C_{add}, analoog aan C_{sup}

door, vanuit de C_{origin} klassen, hetzelfde afleidingsproces te volgen dat de andere kant op werd gebruikt om de globale klassen van C_{sup} te vinden.

Dus stel dat er in het voorbeeld een klasse Medewerker moet komen die een directe subklasse is van Persoon en een directe superklasse van Docent en Studenten Assistent. Dan zal deze klasse eerst gemaakt moeten worden als een subklasse van Persoon dmv de klasse toevoegoperatie en dan kan dmv de voeg superklasse relatie toe operatie deze klasse een superklasse gemaakt worden van Docent en Studenten Assistent.

Verwijder klasse

Dit is de **hide** operator van MultiView [20]. Deze operator haalt de klasse uit de view. Het globale schema blijft hierbij onveranderd.

Hernoemen

Het hernoemen van de verschillende eigenschappen en klassen gebruikt ook standaard functies van MultiView.[20] Het gaat hier om:

rename_attribute
rename_method
rename_class

Bij deze operaties is het zo dat de effecten ervan worden gepropageerd op de subklassen. Eerder overschreven eigenschappen of klassen worden hersteld. Wordt er een naam toegekend die al bestaat, dan wordt het aan de gebruiker overgelaten welke er daadwerkelijk gebruikt gaat worden.

Hoofdstuk 5 Schemaevolutie bij Pluriform

Inleiding

In dit hoofdstuk zal worden besproken hoe schemaevolutie bij Pluriform plaatsvindt. Hier zal gebruik gemaakt worden van het stelsel axioma's uit hoofdstuk 3.

Een belangrijk verschil tussen bijvoorbeeld TSE en Pluriform is dat er bij Pluriform geen sterke overerving is. Er is dan ook geen sprake van een echte object hiërarchie maar van een gerichte graaf. Deze geeft aan hoe de verschillende klassen aan elkaar gerelateerd zijn. Een klasse is gerelateerd aan een andere klasse door middel van een pointer-attibuut. Via zo'n pointer-attibuut kunnen alle eigenschappen van de klasse waarnaar verwezen wordt worden bereikt. Gevolg van zwakke overerving is dat het axioma van 'no cycles' niet direct is af te dwingen. Dit is omdat een pointer in principe over twee richtingen is af te lopen. Hierdoor is het stellen van het "no cycles" axioma niet mogelijk. Elke pointer zou dan in principe al een cycle zijn.
Een pointer relatie zal worden gedenoteerd als $x \uparrow t$ dit houdt in dat de klasse x een pointer-attibuut heeft dat verwijst naar de klasse t. De verzameling van pointer eigenschappen van een klasse t wordt weergegeven met S(t).

Het 3^e axioma voldoet altijd want de klasse Klasse is de basis van elk schema. Je kunt deze klasse dan ook niet verwijderen.

Verder zal er in dit hoofdstuk niet worden gelet op verschillende gebruikers met verschillende rechten. Het is met Pluriform namelijk mogelijk ervoor te zorgen dat bepaalde gebruikers maar bepaalde delen van het object schema mogen zien. Er is uitgegaan van slechts één gebruiker is die het gehele schema mag zien en aanpassen.

Het hoofdstuk is zo opgebouwd dat eerst een beschrijving wordt gegeven van de schemaevolutie operaties op klasse niveau en dan worden die van de eigenschappen van een klasse beschreven. Alle operaties zoals die in hoofdstuk 3 zijn geïntroduceerd zullen aan bod komen.

Veranderingen op klasse niveau

Klasse toevoegen.

Een nieuwe klasse wordt in een aantal stappen toegevoegd. Aan de hand van deze stappen is het gemakkelijk om te bewijzen dat het een correct schema oplevert. De nieuw toegevoegde klasse is t met $e_1 \ldots e_n$ als eigenschappen.

1) De eerste stap is het aanmaken van een klasse definitie. Tijdens het aanmaken van een klasse definitie wordt de klasse toegevoegd aan het klasse schema en wordt er een unieke ak toegewezen aan de klasse. Dus $T = T \cup t$.

Na deze stap is er nog steeds een correct schema omdat deze aktie alleen van invloed kan zijn op het eerste axioma en dat blijft kloppen omdat de klasse ook aan T wordt toegevoegd.

2) Dan worden de lokale eigenschappen gedefinieerd. Dit zijn zowel de attributen als de methoden. Dus $N(t) = \{e_1 \ldots e_n\}$ en $I(t) = I \cup \{e_1, \ldots, e_n\}$.

Na deze stap zijn er nog geen erfbare eigenschappen en dus is $N(x) = I(x) - H(x)$. Daarmee klopt het 5^e axioma.

3) De volgende stap is het definiëren van zijn pointer eigenschappen. Dus $S(t) = \{p_1 \ldots p_n\}$met $p_i = \{t \uparrow x | x \in T.\}$

Na deze stap zijn alle eigenschappen van zijn superklassen te bereiken via deze pointer. Hierdoor is H(t)=U I(P(t))

4) Via deze pointers kunnen methoden worden gedefinieerd die gebruik maken van niet lokale eigenschappen. Deze methoden worden toegevoegd aan de lokale eigenschappen van de klasse.

Ook deze methoden krijgen unieke ak's waaraan ze worden geïdentificeerd. Hierdoor kan er geen verwarring ontstaan met de geërfde eigenschappen.

Klasse verwijderen

Een klasse verwijderen kan alleen als deze klasse niet (meer) wordt gebruikt door andere klassen. Deze beperking is om ervoor te zorgen dat altijd alle methoden werken. Zou er namelijk in een subklasse een methode zijn die gebruik maakt van een lokale eigenschap van de te verwijderen klasse, dan zou deze methode illegaal worden. Dus voordat een klasse t kan worden verwijderd moet gelden: $\forall_{x \in T} [t \notin SUP(x)]$.

Pluriform geeft direct aan in welke klassen deze klasse nog wordt gebruikt. Als alle verwijzingen naar de klasse zijn verwijderd (zie ook het verwijderen van een eigenschap hiervoor) dat is de klasse een blad geworden in de object graaf. Dit houdt in dat er geen pijlen meer naar de knoop van deze klasse wijzen. Het maakt niet uit of er nog verwijzingen van deze klasse naar andere klassen gaan.

Nadat de klasse is verwijderd is er weer een correct schema over:

- Volledig: $\forall_{x \in T}[\forall_{t \in P(x)}[t \in T]$ geldt omdat er uiteindelijk een blad uit de graaf is verwijderd.
- De basis van het schema is niet veranderd. Dit is nog steeds de klasse met alle eigenschappen. De eigenschappen van de klasse zijn met de klasse mee verwijderd. Deze eigenschappen kunnen nergens ander invloed op hebben gehad omdat van tevoren vast stond dat ze nergens meer werden gebruikt. Hiermee is ook voldaan aan axioma's 5 en 6.

Voeg klassen samen

Om klassen A en B samen te voegen worden de eigenschappen van A toegevoegd aan B. Waarna klasse A wordt verwijderd. De klassen die gebruik maken van eigenschappen van A kunnen worden verwezen naar de nieuwe eigenschappen van B. Dit levert weer een correct schema op omdat het verwijderen van een klasse en het toevoegen van eigenschappen een correct schema opleveren.

Splits klasse

Om een klasse A te splitsen in klassen A en A' wordt er een nieuwe klasse A' gemaakt met een gedeelte van de eigenschappen van de originele klasse A. De verschillende verwijzingen die worden gedaan naar die eigenschappen van A kunnen probleemloos worden verlegd naar de eigenschappen van A'. Hierna kunnen de eigenschappen uit A worden verwijderd. Dit levert ook geen problemen op omdat alle relaties hierna artoe al zijn verwijderd. Dit levert een correct schema op omdat het verwijderen van eigenschappen en het creëren van een nieuwe klasse een correct schema opleveren.

Hernoemen van een klasse

Het hernoemen van een klasse kan simpelweg gebeuren door de naam van de klasse aan te passen. Dit heeft voor de rest geen gevolgen omdat de naam van een klasse slecht een attribuut is van deze klasse. De klasse zelf wordt geïdentificeerd aan de hand van zijn ak.

Voeg superklassenrelatie toe

Het toevoegen van een nieuwe superklassenrelatie gaat analoog met het toevoegen van een nieuwe eigenschap.

Haal superklassenrelatie weg

Voordat een superklassenrelatie, r van klasse x, kan worden weggelaten moet er zeker gesteld zijn dat de relatie niet meer wordt gebruikt. Er moet dus gelden dat $\forall_{t \in T}$ [r gebruikt in t $\Rightarrow$ t = x]. Dit wordt door Pluriform zelf afgedwongen. Het DCR geeft aan waar de relatie nog wordt gebruikt.
Er is hier een belangrijk verschil met de sterke overerving. Een relatie die niet wordt gebruikt mag dus wel worden weggehaald.

Verplaats een klasse

Het verplaatsen van een klasse gebeurt door het veranderen van zijn pointer eigenschappen. Sommige pointers worden verwijderd en anderen worden toegevoegd.

Schema operaties op eigenschappen niveau

Eigenschap toevoegen

Een eigenschap toevoegen gebeurt direct in de klasse definitie. Er zijn twee stappen waarin dat gebeurd:

- De eerste is het definiëren van de verschillende relaties tussen de klassen. Dit is in de vorige paragraaf behandeld en levert dus altijd een correct schema op.
- De tweede stap is het definiëren van de eigenschap. Hierbij kan gebruik gemaakt worden van de klasse graaf. Alle gelinkte eigenschappen kunnen worden gebruikt Ook dit levert een correct schema omdat de semantiek van een eigenschap niet van invloed is op de correctheid van een schema.

Eigenschap verwijderen

Een eigenschap verwijderen kan problemen opleveren als de eigenschap ergens anders nog gebruikt wordt. Ook hier moet dus gelden: $\forall_{t \in T}$ [r gebruikt in t $\Rightarrow$ t = x]. Ook hier geeft Pluriform dit zelf aan. DCR geeft weer aan waar de eigenschap nog gebruikt wordt.

Hernoemen van een eigenschap

Het hernoemen van een eigenschap heeft geen consequenties voor de correctheid van een schema omdat de naam van een eigenschap niet typerend is voor de eigenschap. Elke eigenschap wordt met een unieke, niet veranderbare ak aangeduid.

Hoofdstuk 6 TSE VS. Pluriform

Bij TSE wordt er nog veel overgelaten aan de gebruiker> het is niet duidelijk hoe dit op een consistente manier wordt bijgehouden en of dit überhaupt kan. (kijken waar die gebeurd en wat de consequenties hiervan zijn>)
View removal. Moeilijk zie andere artikel
Veel overhead bij TSE elke view weer opnieuw
Maar Pluriform heeft dit ook. Alle conversies moeten meegenomen worden.
Pluriform werkt in de praktijk, TSE alleen maar in test situaties. Wel gebaseerd op werkend Orion Systeem
Vanwege de algemene opzet van Pluriform is wat er gebeurd misschien wel minder mooi maar het blijft wel consequent. Bij TSE zijn er nog veel uitzonderingen die aan de gebruiker worden overgelaten. Dit zal, zeker bij veranderingen met een grote impact-boom ook voor de gebruiker erg onoverzichtelijk worden. Dit omdat er veel veranderingen wel automatisch gebeuren. Hierdoor kan de gebruiker het overzicht snel verliezen waardoor de keuze die dan aan de gebruiker gesteld wordt niet meer goed kan worden ingeschat en er zo de verkeerde beslissing kan vallen.

Referenties

[1] Young-Gook Ra, and A. Rundersteiner. “A Transparent Schema-Evolution System Based on Object-Oriënted View Technology” IEEECS 1997 Log number 104398.
[2] J. W. H. L. Booij. “Analyse Van Interactieve Application Development Methode Met Behulp Van PSM2” KUN 1998.
[3] J.J.T. van de Donk, “Pluriform Methode, Meer dan evolutionaire systeemontwikkeling. Veghel juni 1997.
[4] K. Jones. PEP Paper 22 CSC Index 1992
[5] Pluriform Application Framework. White paper, may 1998
[6] Meta Group evalueert Pluriform. Beschikbaar via www.pluriform.com/ned/7433.htm
[7] Kevan Jones. PEP Paper 22 ,May 1992;’Improving Development Performance throught Re-use.”. CSC Index, A company of Computer Sciences Corporation.
[8] Dietz, Prof dr ir J.L.G. “Van informatietechnologie naar organisatietechnologie” beschikbaar via http://www.sbi.nl/congresnet/catim/60493.htm
[9] S. Foppema “Schemaevolutie in de pratijk”. Afstudeerscriptie van de vakgroep Informatiesystemen aan de universiteit van Delft.
[10] Jorgensen, M (1995). “An Empirical Study of Software Maintenance Tasks:. Journal of Software Maintenance, Vol7, 1995.
[11] Sommerville, I (1995), vijfde druk, “Software engineering”, Addison-Wesley,
[12] Chikofsky , E & Cross, J (1990). “Reverse Engineering and Design Recovery: A Taxonomy”. IEEE Software, Vol 7 No 1. pp 13-17 January 1990
[13] Jones, K (1992) PEP paper 22 “Improving Development Performance throught re-use” CSC index 1992
[14] Atkinson, M.P., Sjoberg, D.I.K. and Morrison, R (1993).”Managing Change in persistent Object Systems” In:First JSSST international symposium on Object technologies for advanced Software, Kanazawa, Japan. Pp315-338 (1993)
[15] Marche, S (1993) “Measuring the stability of Data Models” European Journal on Information Systems, Vol 2, No1 pp37-47 1993
[16] Sjoberg, I.K. (1996) “Towards a Methodology for Evaluating Software Maintenance Technology” Department of Informatics, University of Oslo Norway.
[17] Humprey, W.S. (1990) “Managing the Software Process” Reading, Massachusetts, Addison-Wesley
[18] Ferraby , L (1991) “Change Control During Systems Development” Prentence-Hall
[19] Sjoberg, D.I.K. (1993) “Quantifyinng Schema Evolution” Information and Software Technology, vol35, no1, pp35-44, January 1993.
[20] Rundensteiner, E. A., “MultiView: A methodology For Supporting Multiple View Schemata In OODBs” 1996
[21] Kuno, H, et al, “The MultiView OODB View System: Design and Implementation”
[22] Bertino, E. “A view mechanism for object-oriënted databases.” 3rd International Conference on Extending Database Technology, pp 136-151 March 1992.
[23] J. Banerjee, W. Kim, H.J. Kim and H.F. Korth, “Semantics and implementation of schema evolution in OO Databases” Proc. Third international conference extending database technology, pp 136-151 Mar 1987
[24] H. C. Howard et al, “Versions, Configurations and Constraints in CEDB” march 29 1994

Termen en afkortingen

Algemeen niveau	Dit is het niveau waarop de verschillende views tegelijk worden weegegeven (basis + virtueel).
Attribuut	Een toestand van een klasse.
Basis database	De database die alle gegevens bevat. Alle view schemata zijn hiervan afgeleid.
Basis klasse	Een klasse uit de basis database.
BOL	Business Object Library. Een set van re-useable business objecten. Een gedeelte van de PAF.
Bron klasse	De klasse waar een virtuele klasse van afgeleid is. Deze klassen kunnen zowel basis als virtuele klassen zijn.
Business Object Library	De herbruikbare klassen en objecten, eventueel samengevoegd tot complete processen van Pluriform.
Capacity Augmenting	Het uitbreiden van de globale klassen in virtuele klassen Zie [22].
Directe Subklasse	Een klasse die één relatie lager ligt in de klasse hiërarchie.
Directe Superklasse	Een klasse die precies één relatie hoger ligt in de klasse hiërarchie.
Eigenschap	Een attribuut of methode.
Extensie	De verzameling van instanties van een klasse.
Globale schema	Het schema dat alle basis en virtuele klassen omvat.
Kernel Systeem	Dit is de tool die PS ontwikkelt. Zij omvat de BOL en de Pluriform engine.
Klasse	Entiteit met een aantal getypeerd door een aantal eigenschappen.
Methode	Het gedrag van een object.
Object	Een instantie van een klasse.
Object hiërarchie	De hiërarchie van een OO-schema geeft de overerving van de verschillende klassen aan. De wortel in de hiërarchie definieert alleen zichzelf. Objecten onderin de hiërarchie erven de eigenschappen van hun superklassen.
PAF	Pluriform Application Framework.
Pluriform	De tool ontwikkeld door Pluriform Software. Ook wel PAF Genaamd.
PS	Pluriform Software.
Subschema evolutie	Evolutie op een virtueel schema.
TSE	Transparent Schema Evolution.
Type	Het domein van toegestane waarden.
View	Zie view schema.
View schema	Een schema van een database opgebouwd door klassen, dit kunnen zowel virtuele als basis klassen zijn.
Virtuele klasse	Een klasse die (direct of indirect) is afgeleid van de basisdatabase.

MultiView een samenvatting

Inleiding

Het MultiView OODB systeem is één van de eerste database systemen die dynamische en aanpasbare database views realiseert. MultiView is ontwikkeld op de Universiteit van Michigan [21]. Het is gebaseerd op GemStone [2] dat o.a. zorgt voor de database programmeertaal en het transactie management.

MultiView biedt de volgende mogelijkheden:

- Gebruikers kunnen virtuele klassen op elk moment aanmaken
- Gebruikers kunnen vragen en updates toepassen op zowel basis als virtuele klassen
- Gebruikers kunnen virtuele schema's maken en aanpassen op elk moment
- Virtuele klassen zijn voor de gebruiker niet te onderscheiden van globale klassen
- Capaciteits uitbreiding (capacity augmenting) [22] is mogelijk. Dit houdt in dat er in één virtuele klasse data gebruikt word die niet af te leiden is uit haar basis klassen.

Beschrijving van de operatoren

Om niet te veel in detail te treden, volgt hier alleen een beschrijving van de operatoren die ik gebruikt heb:

- De define VC operator:

De **define VC** operator wordt aangeroepen met (**define VC** <klasse> **as** <operatie>). Deze maakt een klasse met de naam klasse en als eigenschappen de uitkomst van operatie. De nieuw aangemaakte klasse zal in het vernieuwde schema de klasse die in operatie gebruikt wordt vervangen. Dit wordt automatisch door MultiView afgevangen.

- De select operator:

De **select** operator wordt aangeroepen met (**select from** <klasse> **where** <predicaat>). Dit levert die set van objecten van klasse die voldoen aan het predicaat.

- De hide operator:

De **hide** operator wordt aangeroepen met (**hide** <eigenschappen> **from** <klasse>). Dit levert een superklasse van klasse waar de eigenschappen uit zijn verwijderd. Alle objecten uit de invoer set zijn ook objecten in de uitvoer set.

- De refine operator:

De **refine** operator wordt aangeroepen met (**refine** <eigenschappen-def> **for** <klasse>). Dit levert een set met dezelfde objecten als Klasse maar dan met een nieuwe eigenschappen: eigenschappen-def

Set operatoren:

- De union set-operator:

De **union** operator is een operator die twee object verzamelingen samenvoegt. Het type is het laagste type dat beide verzamelingen gemeenschappelijk hebben. Dit bestaat altijd, want in het uiterste geval zijn beide verzamelingen van het type object.

- De diff set-operator:

[2] GemStone is een geregistreerd product van GemStone Systems Inc.

De **diff** operator levert een subset van zijn eerste argument op. Deze subset bevat die argumenten die ook in het tweede argument van de **diff** operator voorkomen.

Deel 2 Splitsing van de business objecten uit de volledige omgeving.

Afstudeeronderzoek bij Pluriform Software

Gert-Jan Haverkamp 9435670

Inhoudsopgave

HOOFDSTUK 1: INHOUDSOPGAVE 36

HOOFDSTUK 2: INLEIDING 37

HOOFDSTUK 3: DE PLURIFORM OMGEVING: PAF 38

INLEIDING 38
DE LAGENSTRUCTUUR 38
APPLICATIES EN APPLICATIE ONTWIKKELOMGEVING 39
PLURIFORM EVOLUTIE 39

HOOFDSTUK 4: GEBRUIKERSSYSTEEM OP DE SCHOOL VOOR INFORMATICA 41

DOELSTELLING 41
VRAAGSTELLING 41
INTRODUCTIE VAN TERMEN 42

HOOFDSTUK 5: DE METHODEN 44

METHODE 1 44
Inleiding 44
Splitsing op basis van licenties 44
Uitzonderingen 45
Aanpak 46
Problemen 46
Conclusies 46
METHODE 2 47
Inleiding 47
Aanpak 47
METHODE 3 48
Inleiding 48
Splitsing op basis van methoden per klasse 48
Conclusies 48
Beschrijving resultaat 50

HOOFDSTUK 6: CONCLUSIES 51

HOOFDSTUK 7: REFERENTIES 52

Hoofdstuk 1: Inleiding

Inleiding

Pluriform Software is een bedrijf, gevestigd in Mariaheide, dat zich met een state-of-the-art IS ontwikkelomgeving bezig houdt. Deze tool is objectgeoriënteerd en gericht op hergebruik. PS is een bedrijf dat zich geheel wil richten op de ontwikkeling van deze tool en de daadwerkelijke exploitatie uitbesteedt aan partners zoals Cap Gemini. Op deze manier hoopt het de tool beter te kunnen evolueren dan wanneer men zich direct afhankelijk maakt van de exploitatie ervan.

De School voor Informatica heeft in principe dezelfde opvatting. Ook de school is niet direct afhankelijk van het gebruik van het product wat ze leveren (kennis) en richt zich ook voornamelijk op het ontwikkelen hiervan (onderzoek).

PS en de School voor Informatica hebben het idee dat er een vruchtbare samenwerking kan ontstaan tussen deze twee partijen. PS ziet hierin een mogelijkheid om ideeën op te doen voor verbeteringen aan hun tool en om bekendheid bij studenten te verwerven. De School voor Informatica kan door de samenwerking moderne, praktische ondersteuning geven aan het onderwijs en krijgt een middel om nieuwe ideeën in de praktijk te brengen.

De School voor Informatica kampt, in het kader van de samenwerking, met een tekort aan resources. De ontwikkelversies van Pluriform vragen snelle PC's met veel geheugen. Het aantal PC's op de KUN die dit aankunnen is echter beperkt. Om goed te kunnen samenwerken moest er een versie van de Pluriform omgeving komen die speciaal gericht was op de School voor Informatica. Dit houdt in dat er zo veel mogelijk functionaliteit verwijderd moest worden die niet door de School voor Informatica gebruikt zou worden om het gebruik van resources te minimaliseren. In dit gedeelte van de scriptie wordt een verslag gedaan van het verkrijgen van zo'n systeem. Het uiteindelijk ontwikkelde systeem heeft de naam Pluriform PDE (Pure Delevoment Environment) gekregen.

Eerst zal een overzicht gegeven worden van hoe Pluriform is opgebouwd en hoe het in de loop van de tijd is geevolueerd (hoofdstuk 2). Dan wordt verder gespecificeerd wat het precieze doel is van dit onderzoek (hoofdstuk 3). Nadat de verschillende methoden van werken zijn besproken (hoofdstuk 4), wordt er een beschrijving gegeven van het eindresultaat en volgen de conclusies (hoofdstuk 5)

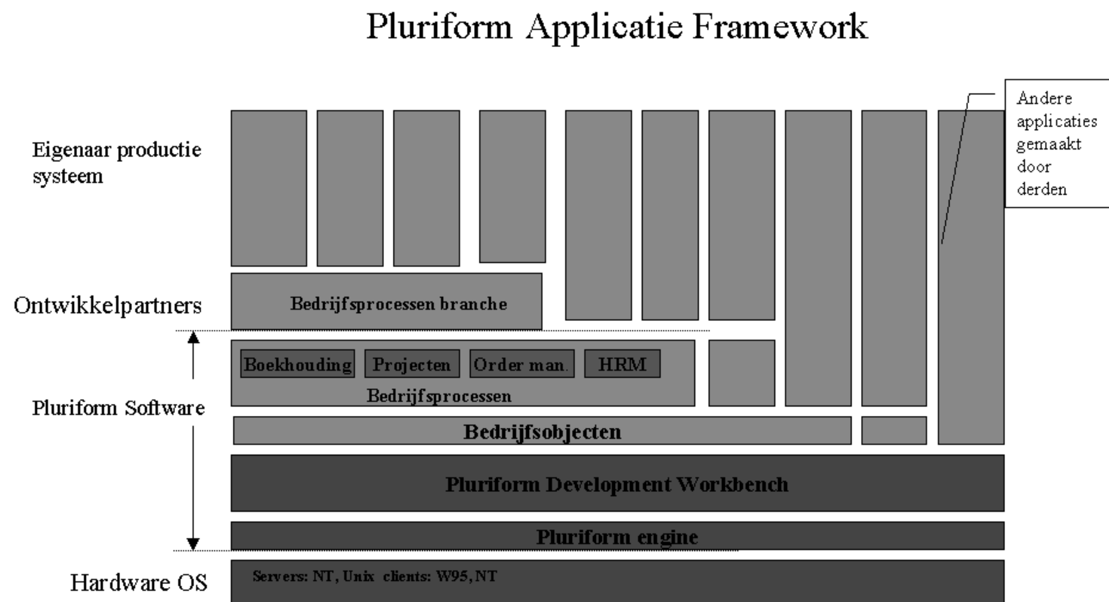
Hoofdstuk 2: De Pluriform omgeving: (PAF)

Inleiding

PS heeft een objectgeoriënteerde ontwikkelomgeving ontwikkeld op basis van herbruikbare componenten namelijk het Pluriform Application Framework (PAF, ook wel Pluriform genoemd). Dit is geen revolutionair idee maar heeft tot nu toe weinig spectaculaire resultaten opgeleverd. PS heeft echter in de afgelopen jaren bewezen dat er met hun tool wel degelijk goede resultaten te behalen zijn. Dit blijkt o.a. uit het feit dat Cap Gemini de tool nu actief aan het gebruiken is.

De lagenstructuur

De Pluriform ontwikkelomgeving bestaat uit een aantal lagen. Deze lagen zijn sterk met elkaar verbonden en voor een gedeelte geïntegreerd. Hij is opgebouwd uit vier verschillende lagen (zie figuur 2.1). Voor een uitgebreide uitleg van de lagen zie [2].



Figuur 2.1 De lagenstructuur van Pluriform

De onderste laag is de Pluriform engine. Deze C++ laag executeert de applicatie- definities uit de repository en vormt een abstractielaag voor verschillende besturingssystemen, netwerken en databases. Bij executie wordt er geen code gegenereerd. De engine zelf is een executable die de beschikbare definities gebruikt om te besturen hoe de applicatie zich gedraagt en eruit komt te zien. In de repository wordt dus uitsluitend een definitie van de applicatie opgeslagen.

Boven de engine zit de Development Workbench. Deze laag bevat de algemene ondersteuning voor de applicatie segmenten. Hierin zitten een aantal applicatie ontwikkelgereedschappen zoals een scripting editor, een rapport generator, performance tuners en een distributiesysteem. De Workbench ondersteunt de gehele software ontwikkelcyclus (zowel bij klanten als binnen Pluriform zelf) met gereedschappen die niet alleen zorgen voor het vastleggen van modellen, de (automatische) definitie van databasetabellen en het creëren van definities voor een automatische gegenereerde GUI, maar ook voor het beheersbaar aanbrengen van wijzigingen in bestaande systemen. Zo is de Dynamische Cross-Reference (DCR) een belangrijk voorbeeld uit de Workbench.

Hierop ligt de laag bedrijfsobjecten (ook wel Business Object Library (BOL) genoemd). Deze bevat een aantal verschillende applicatie segmenten. Voorbeelden hieruit zijn: Persoon, Onderneming, Artikel en Project. Momenteel bevat deze laag rond de 300 klassen.

De bovenste laag is de bedrijfsprocessen laag. Dit zijn typische high-level frameworks zoals: Ordermanagement, Human Recource Management en Grootboek. Het gaat hier om bedrijfsprocessen die gebaseerd zijn op de transactie-, systeem- en abstractieconcepten volgens de Dynamic Essential Modelling of Organisations (DEMO) methode. Deze methode levert een model op dat geheel onafhankelijk van de manier waarop het, zowel in organisatorische als in informatorische zin is gerealiseerd. In dit model komen dus geen afdelingen en hiërarchische structuren voor.

Applicaties en applicatie ontwikkelomgeving

Zoals eerder is vermeld is het grootste gedeelte van Pluriform is met behulp van Pluriform zelf gemaakt. Alleen de Pluriform engine is in C++ geschreven. De Workbench is helemaal met behulp hiervan gemaakt. Hetzelfde geldt voor de lagen die daar boven op liggen; deze zijn allemaal gemaakt binnen Pluriform en dus zonder gebruik van extra C++ code gemaakt.

Pluriform is echter van mening dat als je wilt zorgen voor een goed onderhoudbare omgeving, de applicaties die je maakt met dezelfde componenten moeten worden gerealiseerd als je omgeving. Dit omdat je op die manier de kloof tussen applicatie makers en ontwikkelomgeving ontwikkelaars zo klein mogelijk houdt. Mocht er iets zijn wat een applicatie-ontwikkelaar wil in zijn omgeving dan kan hij dat zelf realiseren en in volgende applicaties weer gebruiken alsof het een gedeelte van de tool is.

Dit idee heeft PS uitgewerkt door het maken van applicaties in een repository. Deze repository is een uitbreiding van de ontwikkelomgeving. In elke applicatie zit dus de gehele ontwikkelomgeving. Het is dan ook onjuist om te spreken van een applicatie. Men zou het eigenlijk moeten hebben over een specifiek uitgebreid Pluriform systeem.

Pluriform evolutie

Omdat Pluriform een snel evoluerende omgeving is en je toch telkens met de meest actuele versie wilt werken, zal er ook voor gezorgd moeten worden dat Pluriform PDE kan mee evolueren. Pluriform kent twee verschillende manieren om een applicatie up to date te houden buiten het "met de hand" bijwerken.

- De eerste is door middel van fixregels. Dit zijn objecten die per object gemaakt kunnen worden. In een fixregel leg je de nieuwe definitie van het object vast. Door deze fixregel te exporteren, en in te lezen op de te vernieuwen applicatie kan de nieuwe definitie worden overgebracht naar het nieuwe systeem zonder dat de nieuwe definitie opnieuw hoeft te worden ingevoerd. Ook kan op deze manier ervoor gezorgd worden dat er aanpassingen worden gedaan op meta niveau waarvoor de applicatie niet is gelicenceerd. Het voordeel van het maken van fixes is dat de gebruiker er niets van hoeft te merken. En fixes kunnen zich opstapelen en je kunt ze verspreiden naar alle versies tegelijk. Op deze manier dan een probleem wat zich bij één klant voordoet en wordt opgelost meteen bij andere klanten worden opgelost voordat die het probleem daadwerkelijk tegen komen.
- De andere methode is door het maken van een update. Dit houdt in dat alle objecten die door de pluriform omgeving worden ondersteund worden ge-update. Vooral voor systemen die al lang draaien zonder veranderingen en die op een gegeven moment toch de nieuwe functionaliteit willen is deze methode erg handig. Fixregels hebben namelijk meestal conversie taken bij zich. Het effect als er meerdere conversietaken op een object plaatsvinden wordt nogal onoverzichtelijk. Op een gegeven moment zegt PS dat vanaf een bepaalde versie de oudere versies niet meer direct geconverteerd kunnen worden maar dat er gebruik gemaakt moet worden van een update. Dit was bijvoorbeeld het geval met de overgang van de character gerichte versie naar de grafische versie. De overgang hier was zo groot dat het niet efficiënt meer was om dat met fixregels te doen.

Hoofdstuk 3: Gebruikerssysteem op de School voor Informatica

Inleiding

Pluriform is een brede ontwikkelomgeving voor informatie systemen. Met een grote Business Object Library (BOL) die door de jaren heen is geëvolueerd naar een niveau dat door maar weinig ander IS ontwikkelomgevingen wordt geëvenaard. Door te werken met verschillende klanten uit verschillende industrieën is Pluriform Software in staat geweest om een grote hoeveelheid kant en klare business logic op te nemen. Zo is er bijvoorbeeld gewerkt met GTI installatie techniek en Van Rossum & Partners. Maar ook met het non profit organisaties zoals het gerechtshof in Den Bosch en Artsen zonder Grenzen. Per industrie is er een gedeelte van de functionaliteit aan Pluriform toegevoegd. Dit zijn dìe delen die geïdentificeerd zijn als algemeen voor die industrie.

Nu is het de bedoeling dat de Pluriform omgeving ook op de School voor Informatica gebruikt gaat worden. Hier zal geen gebruik gemaakt worden van de business functionaliteit. Dit omdat er op de School voor Informatica geen informatie systemen geschreven zullen worden voor het type klanten zoals Pluriform dat kent. Deze BOL neemt wel een grote hoeveelheid ruimte in beslag. Omdat op de School voor Informatica de resources beperkt zijn is het wenselijk om een systeem te hebben waar deze functionaliteit niet meer in aanwezig is. Dit komt ook ten goede aan de overzichtelijkheid. Dit is zeker voor pas beginnende Pluriform modelleurs een voordeel.

Nadat de business logic is verwijderd blijft er een informatiesysteem ontwikkelomgeving over die door een gebruiker gemakkelijk kan worden uitgebreid op dezelfde manier als Pluriform zijn business objecten heeft ontwikkeld. Deze versie van Pluriform heeft de naam Pluriform PDE (Pure Development Environment) gekregen.

Pluriform PDE heeft dus de meest elementaire functionaliteit, met als doel dat deze door de gebruikers uitgebreid kan worden. Hiervoor biedt het dezelfde mogelijkheden die Pluriform Software nu heeft. Dit houdt in dat Pluriform PDE kan worden uitgebreid en er zo een goed onderhoudbare en herbruikbare bibliotheek van objecten en functionaliteit onstaat die is geoptimaliseerd naar de wensen van de gebruiker(s) die er op dat moment mee werken.

Doelstelling

De doelstelling van het te ontwikkelen Pluriform PDE systeem is het maken van een zo kaal mogelijk Pluriform systeem. Dit systeem zal dus alle karakteristieke eigenschappen moeten hebben van Pluriform maar zal de voor hergebruik meegeleverde objecten niet meer omvatten.

Vraagstelling

Uit de doelstelling volgt de volgende vraag: ‘Hoe kunnen meta-objecten onderscheiden worden van business objecten?”. Door antwoord te geven op deze vraag kan, de doelstelling worden bereikt door deze manier van identificeren toe te passen op de verschillende objecten binnen PAF en dan alleen de meta-objecten te exporteren naar het Pluriform PDE systeem.

Introductie van termen

Nu zal kort een aantal termen worden geïntroduceerd dat vaak voorkomt.

Klasse: een klasse is een definitie van een structuur die instanties van de klasse allemaal moeten hebben. Een klasse bevat o.a.

- een aantal methodes
- een aantal permissies
- een module pointer
- een aantal eigenschappen
- een reeks van tests

Voor een compleet overzicht zie [1]. Elke klasse (behalve de klasse Klasse zelf) is dus een instantie van de klasse Klasse.

Object: Een object is een instantie van een klasse. Neem bijvoorbeeld de klasse Persoon. Deze klasse legt dan bijvoorbeeld vast dat een Persoon een Achternaam en een Voornaam heeft. Janssen, Jan is dan een voorbeeld van een object van de klasse Persoon.

Attributen: attributen zijn de eigenschappen (velden) van een klasse. Een attribuut is van een bepaald type (bijv.text of date). Een belangrijk attribuut van elke klasse in de ak (afkorting van archief code). Dit attribuut wordt intern gebruikt als identificatie. Gebruikers kunnen hier, gesteld dat ze de goede licenties hebben, natuurlijk ook gebruik van maken.

Relations: Met een relation definieert u een verband tussen twee klassen. Eén klasse verwijst naar een andere klasse met een pointer-attibuut. Deze kan hooguit naar één object tegelijk wijzen.

Eigenschappen: dit zijn flags die meegegeven kunnen worden aan een klasse. De belangrijkste eigenschappen op dit moment zijn:

- Export: dit geeft aan dat de objecten van de klasse meeworden mee geëxporteerd met een update. Het maken van een update word later nog besproken.
- Private: geeft aan dat een object alleen bedoeld is voor een bepaalde (groep) gebruiker(s)/rol(len).

Permissies: hierin wordt vastgelegd welke rollen of gebruikers de attributen van de klasse mogen wijzigen.

Gebruiker: iemand die een login heeft tot een Pluriform systeem

Rol :gebruikers kunnen worden gegroepeerd in rollen. Per rol wordt vastgelegd welke permissies de gebruikers hebben.

Modules: Binnen Pluriform zijn er een aantal (op dit moment 98) modules. Deze modules vormen een opsplitsing van Pluriform die objecten indeelt per categorie. Zo is er een module “azg” die de objecten voor Artsen Zonder Grenzen omvat. Maar er is ook een module “financieel” waar objecten inzitten zoals boekhouding, journaalpost en aangifte. In principe is deze opsplitsing er voor klassen, maar per object kan dit pointer-attibuut ook worden gebruikt.

Tests: hiermee kunnen constraints worden aangegeven waaraan een object moet voldoen.

Flags: dit zijn markeringen die aan een object meegegeven kunnen worden.

Licenties: licenties geven aan welke modulen een Pluriform systeem mag gebruiken. Een licentie moet aangevraagd worden bij PS en zal telkens voor maar een bepaalde tijd gelden. Na dat een licentie is verlopen kunnen er geen nieuwe objecten meer worden toegevoegd. Er zal dan een nieuwe licentie aangevraagd moeten worden.

Ak: elk object in Pluriform wordt automatisch een uniek ak (afkorting van archief code) gegeven. Aan de hand van dit ak wordt een object geïndetificieerd.

Hoofdstuk 4: De methoden

Inleiding

Globaal zijn er twee manieren om een groep in tweeën te splitsen. De eerste is van het geheel uitgaan en dan te kijken wat er niet nodig is. De andere is uitgaan van niets en kijken wat er nodig is. In dit geval is de eerste methode de meest veilige. De kans dat je iets vergeet is nihil. Er zal eerder een werkend systeem zijn. In principe is deze methode al succesvol voordat je eraan begint. De tweede methode is netter. De kans dat er objecten teveel meegaan is kleiner. Het nadeel hieraan is dat er moeilijk een begin gevonden kan worden. Verder is het ook moeilijk toe te passen op grote onoverzichtelijk verzamelingen als essentiële objecten overal kunnen zitten.

De splitsing methode moet generiek zijn omdat er op elk moment een nieuwe versie van Pluriform PDE gemaakt moet kunnen worden. De methode moet overdraagbaar zijn omdat het PDE systeem mee moet kunnen evolueren met het kernel systeem. Omdat de splitsing methode generiek moet zijn en overdraagbaar naar andere systemen zal de methode moeten worden gemaakt in het Pluriform kernel systeem. Dit is de versie van Pluriform waar de ontwikkelaars aan werken en van waaruit alle veranderingen centraal worden gecoördineerd en gedistribueerd.

Het eerste probleem is dat er niet zomaar objecten uit de huidige versie weggegooid kunnen worden. Dan zou het kernel systeem ook alle functionaliteit verliezen. Dit probleem is opgelost door een methode te maken die een kopie maakt van het werkende systeem en per object gaat kijken of het wel of niet mee moet worden gekopieerd. Deze manier van werken is analoog aan die van het maken van een update. Deze methode geeft ook al een eerste voorzet voor een splitsingsmethode. Dit omdat er daar al een filter ingebouwd is die objecten niet exporteert die nog niet voor distributie gereed zijn, waar bijvoorbeeld nog aan ontwikkeld wordt of die voor intern gebruik zijn.

Methode 1

Inleiding

De eerste manier van splitsen is gebaseerd op het licentie principe. In Pluriform worden er licenties verstrekt die bepalen welke objecten een systeem wel en welke objecten een gebruiker niet zelf mag gebruiken. Het idee achter deze methode is dat de kleinste licentie alleen die objecten bevat die nodig zijn om te modelleren

Van de objecten wordt per object gekeken of het een relatie heeft met een module of een klasse die niet meegaat. Is dit het geval dan zal dit object ook niet worden meegenomen. Verder zal er gebruik gemaakt worden van de features die het update mechanisme al had. Dit houdt in dat bijvoorbeeld alleen die objecten worden meegenomen waarvan de klasse de eigenschap export heeft.

Splitsing op basis van licenties

Een licentie is opgebouwd uit modules. Elke licentie omvat een aantal modules. Een aantal objecten heeft zo'n module pointer. De klasse Klasse is de belangrijkste klasse die zo'n module pointer heeft. Er kan dus van een aantal klassen worden bepaald of ze meegaan of niet.

Aangezien alles tot een klasse behoort geeft dit meteen weer een extra beperkingmogelijkheid. Dit omdat er een aanzienlijk aantal objecten een pointer heeft naar een klasse. Deze extra mogelijkheid is erg sterk omdat veel klassen een pointer hebben naar een klasse. Dit kan zijn een pointer naar een klasse die aangeeft dat object echt direct bij die klasse hoort. Een mooi voorbeeld hiervan is een View. Een View is een object dat o.a. aangeeft hoe de opbouw van de attributen is geregeld en hoe het attribuut moet worden weergegeven. Elke View moet dus direct gerelateerd zijn aan een klasse wil hij nog nut hebben.

Verder kan een pointer naar de klasse Klasse ook nog aangegeven wat de resultklasse is van het object. Mocht het een klasse opleveren die niet geëxporteerd wordt moet het object zelf ook niet worden geëxporteerd. Dit omdat er anders fouten op kunnen treden.

De aanname is bij deze methode dat de objecten die zijn toegedeeld aan een module de meta-objecten zijn. Met deze stelling pretendeer ik niet de vraagstelling volledig te beantwoorden. Dit omdat er waarschijnlijk meer objecten in zitten dan alleen meta-objecten. Het voordeel van deze methode is wel dat het een goede start levert om vanuit daar verder te gaan. Als deze methode succes heeft dan hoeven alleen de objecten van deze modulen nog maar te worden gescheiden.

Uitzonderingen

Er zijn een aantal groepen die een uitzondering vormen. Nu volgt een overzicht van deze groepen en de reden waarom ze een uitzondering vormen. Ook zal worden uitgelegd hoe het probleem is opgelost.

De klasse module. Alle instanties van deze klasse zullen worden meegenomen omdat anders er conflicten komen met het licentie systeem. Dit heeft namelijk de alle modules echt nodig. Verder is het ook handig voor de uiteindelijke gebruikers. Die kunnen dan gemakkelijk zien welke modules er al te krijgen zijn en (via hun hulp) kunnen zien wat de verschillende modules omvatten. Mocht een gebruiker besluiten om een extra module aan te schaffen dan kan dit door middel van hetzelfde algoritme gemakkelijk worden toegevoegd. Dit kan dan zonder dat de al ingevoerde applicaties worden verwijderd.

Marked internal. Er zijn een aantal klassen gemarkeerd als internal. Deze markering wordt door een ontwikkelaar zelf aangegeven. Dit doet hij om ervoor te zorgen dat de klasse niet wordt geëxporteerd naar een distributie systeem. Veel voorkomende redenen hiervoor zijn dat de klasse nog niet helemaal af is of omdat hij slechts tijdelijk gebruikt gaat worden.

Marked kernel_only. Deze objecten zijn door de ontwikkelaars geïdentificeerd als kernel_only. Dit kan verschillende redenen hebben maar de belangrijkste is het consistent houden van het permissie mechanisme. Er zijn voornamelijk methoden gemarkeerd die wildcard permissies hebben. Mocht een klant deze gebruiken dan zou hij objecten aan kunnen roepen die misschien wel buiten zijn licentie liggen. Om dit te voorkomen worden deze objecten niet geëxporteerd.

Objecten zonder module/klasse pointer. Deze objecten zijn niet op te delen in meta en business klassen. Deze zullen dus allemaal meegenomen worden. Er zal later ook hier een verdere splitsing moeten worden gemaakt.

Aanpak

Zoals eerder vermeld is er gewerkt met een methode die analoog is aan de methode die wordt gebruikt voor het maken van een update. Deze methode is nagemaakt met de mogelijkheid voor het werken als er andere stations actief zijn. Eerst is bepaald welke modules er in de kleinste licentie zitten. Dit zijn:

- Eigenaarschap
- Eigen-model
- Instellen
- Kantoor
- Systeem
- Tweetalige data

Deze modules zijn opgenomen in een aparte groep die wordt meegegeven aan het algoritme, dit heeft als voordeel dat als je de groep modules wilt aanpassen, je dit centraal kunt doen en dat je niet de hele algoritme structuur af hoeft te gaan op zoek naar waar de groep wordt gebruikt.

Aan de hand van deze modules wordt bepaald welke kan worden bepaald welke klassen er mee moeten worden geëxporteerd. Dit zijn 86 klassen.

Problemen

Het probleem waar ik bij deze methode tegenaan liep was dat het erg moeilijk te achterhalen is waar het fout gaat. Op het moment dat een test systeem niet werkt is er maar weinig informatie beschikbaar. Alleen de thread tot het object dat de fout opleverde is beschikbaar. Er moet dan worden gekeken van welke objecten dit object gebruik maakt en per object moet worden bekeken of het wel al dan niet mee is gegaan. Per object zijn er gemiddeld zo'n vijftig objecten die worden aangeroepen. Is het betreffende object eenmaal gevonden dan moet zìjn thread misschien ook nog mee worden genomen.

Is er een object in de thread gevonden dat niet meegaat moet er gekeken worden of dit al dan niet terecht is gebeurd. Hiervoor moet worden gekeken naar de semantiek van zijn pointers. In het geval dat het object wel mee had gemoeten zal een aanpassing gedaan moeten worden op het object. Als het object terecht niet mee is gegaan moet er een ander object worden gezocht wat de fout heeft veroorzaakt.

Na een groot aantal aanpassingen is besloten dat deze methode niet werkbaar is vanwege de grote onoverzichtelijkheid in de module / klasse relations.

Conclusies

- Het eerste probleem waar tegenaan gelopen wordt is de onzekerheid over de semantiek van de pointers. Als er een pointer is met bijvoorbeeld hoort NIET bij klasse. Dan zou het algoritme op basis hiervan het object niet meenemen terwijl het misschien juist wel meegenomen moet worden. Er is gekeken of dit soort pointers er nu in voorkomen en dat is niet het geval. Alleen zegt dit niets over de toekomst.
  Voor de rest is dit geen echt probleem omdat het niet waarschijnlijk is om zo'n relatie te leggen. Na enig onderzoek is gebleken dat zo'n p ointer op dit moment niet bestaat. Dit is echter geen garantie voor de toekomst.

- Het algoritme is niet specifiek. Op het moment dat een object geen relaties heeft wordt het toch meegenomen. Er wordt bij deze objecten op geen enkele manier gekeken of er op een andere manier uitsluitsel geboden kan worden over of het object wel of niet meegenomen moet worden.
- De opdeling in modulen is gebaseerd op waar de ontwikkelaar vindt dat hij bij hoort. Deze indeling is niet gemaakt specifiek voor het maken van een gestripped systeem. Er kunnen dus een aantal objecten meegenomen worden die hierin dan ook niet thuis horen. Aan de andere kant geeft het wel een goede reden om met meer aandacht te kijken naar de module pointers.

Methode 2

Inleiding

Na de resultaten van de eerste methode kwam de stelling dat er misschien een derde soort objecten is. Deze laag heb ik de informatie objecten genoemd. Dit zijn objecten die op basis van hun semantiek onder de business objecten zouden vallen maar die het Pluriform systeem wel nodig heeft.

Er zijn verschillende redenen waarom een object nodig is voor het PAF. De belangrijkste en meest algemene is dat als een object wordt aangeroepen in een script en het object wordt daarna verwijderd dan zal het script ongeldig worden. Hierdoor verandert het type van het script naar een foutmelding. Door dit type klopt het type van het script niet meer in de scripts die hiervan gebruik maken en die zullen dus ook weer ongeldig worden. Deze objecten zullen dus ook mee moeten worden geëxporteerd

Aanpak

Er is uitgegaan van de methode zoals die in methode 1 wordt beschreven. Dit is gebeurd vanuit de optiek dat er eerst een werkend systeem moet komen en dat er later verder zal worden uitgefilterd. Aangenomen wordt dat de groep objecten uit de eerste methode de meta-objecten zijn. Vanuit deze objecten moet dan worden gekeken van welke objecten dit object gebruik maakt. Is er een object gevonden dat nog niet is meegegaan dan zal dit object ook mee moeten worden geëxporteerd.

Nu volgt er het probleem dat ook de context van dit object meegenomen zal moeten worden. Stel dat het hier gaat om iets van een type van een klasse die niet mee gaat dan zal deze klasse toch meegenomen moeten worden. De objecten van deze klasse hoeven dan niet worden meegnomen maar zijn methoden bijvoorbeeld dan weer wel. Deze methoden zullen dan weer zorgen voor een groot aantal nieuwe objecten die dan weer wel meegenomen moeten worden. Door deze methode zal er een zodanig aantal objecten extra meemoeten als informatie objecten dat dit het voordeel van de splitsing teniet zullen doen. Na het bekijken van een benadering door middel van een teller script bleek dat alle objecten mee zouden worden geëxporteerd. Er is dan ook gekozen om niet verder te gaan met deze methode.

Methode 3

Inleiding

Het grote probleem bij de eerste methode was de onoverzichtelijkheid. Er waren teveel redenen waarom een object te onrechte niet meegegaan was. De precieze semantiek van veel pointers is moeilijk te achterhalen zonder diep in de objectstructuur te duiken.
Om toch meer overzicht te krijgen is er gekeken naar een manier om het probleem op te splitsen in kleinere problemen. Een analyse van de resultaten van de eerste methode was dat door de 6 modules uit de kleinste licentie er 86 klassen werden geëxporteerd. Van deze 86 zijn er maar 42 waarvan de objecten worden geëxporteerd. Ten koste van een gedeelte van de generiek is toen besloten om bij elk van deze 42 klassen een methode te maken die bepaald welke objecten wel en welke objecten niet worden geëxporteerd.

Het voordeel van deze methode is dat de semantiek van de pointers bekend is. Er zijn pointers waar dat nog steeds niet het geval is omdat het daar direct afhangt van de semantiek waarin het object zich bevindt. Maar dit kan met deze methode dan al op het hogere hoger niveau worden onderkend. Deze pointers zullen dan ook niet worden gebruikt om de splitsing te realiseren.

Splitsing op basis van methoden per klasse

Er zal dus per klasse worden gekeken hoe de pointers kunnen worden gebruikt om te selecteren welke klassen er wel of niet meegaan naar Pluriform PDE. Het voordeel hiervan is tweeledig. Aan de ene kant is het overzicht veel beter. Per klasse liggen de pointer relaties vast en ligt de semantiek voor een groot gedeelte ook vast. Er wordt per pointer gekeken of het een goed criterium bied voor de splitsing. Verder kunnen ook pointer naar andere klassen dan de klasse Klasse of de klasse Module worden gebruikt. Een voorbeeld hiervan is de klasse Viewline. Deze klasse bepaald de opbouw van een veld van een klasse. Elke Viewline hoort bij een View. Een View bepaald de opbouw van de gehele klasse. Elke View heeft een pointer naar een klasse. Op basis van deze pointer kan er goed besloten worden of een View wel of niet mee moet. Met behulp van de nieuwe methode kan de pointer naar de View dus ook gebuikt worden om te kijken of een Viewline mee moet. Een Viewline hoeft alleen mee als de View ook mee gaat. Dit kan dus simpel worden getest door in zijn View de selectie methode aan te roepen.

Aan de ene kant wordt het dus specifieker. Dit gaat ten koste van de generiek. Er zal nu per klasse waarvan de objecten worden geëxporteerd een methode moeten worden aangemaakt die de selectie maakt. Dit moet gebeuren ook zijn er twee van deze methodes gelijk. Een methode is namelijk aan een klasse gebonden.

Ik heb alleen voor de klasse die nodig zijn voor het Pluriform PDE systeem zo'n methode gemaakt. Mocht er worden besloten om ook voor andere groepen van klassen een afsplitsing te maken zal er voor de extra klassen nog zo'n methode moeten worden gemaakt.

Conclusies

Bij het testen van deze methode kwam al snel de conclusie dat de engine niet bestand was tegen niet bestaande scripts. De type checker van Pluriform gaf dan direct een type error omdat een niet bestaand script niet noodzakelijk van het juiste type is.
IF nee

THEN niet bestaand script
ELSE ja
Leverde dus al problemen op. Om dit probleem op te lossen is er een aanpassing gekomen in de engine die alleen nog maar een probleem in het systeem oplevert als een foutief script daadwerkelijk wordt aangeroepen. Het niet bestaande gedeelte wordt vervangen door de melding 'illegaal script". Dus bovenstaand script gaat dus nu wel goed.

Deze oplossing leverde meteen het probleem op bij het testen dat niet bekend is wel object een eventueel probleem oplevert. Alleen de melding 'illegaal script" wordt gegeven. Dit is opgelost door ook aan te geven object er illegaal is door middel van zijn ak. De klasse van dit object is te vinden door middel van de type gebondenheid.

Beschrijving resultaat

Het uiteindelijke Pluriform PDE systeem is een goed werkend systeem geworden. Het is o.a. getest door een nieuwe medewerker van Cap Gemini de handleiding 'inleiding modelleren" [1] door te laten werken op het PDE systeem. Hierin komen de meeste handelingen voor die je met Pluriform kunt doen. Hij is niet gestuit op enige systeem fouten.
Dit is, helaas, geen bewijs dat alle noodzakelijke objecten zijn meegekomen. Maar geeft wel een indicatie dat er een systeem is ontstaan waar goed op gewerkt kan worden. Mochten er toch nog objecten gevonden zijn die niet zijn meegegaan dan kan dit, zonder verlies van informatie d.m.v. een fixregel gemakkelijk worden toegevoegd. Ook kan er simpel voor gezorgd worden dat zo'n object in de volgende versies van Pluriform PDE wel meegenomen wordt.
Aan de andere kant is het wel zo dat er nog een redelijk aantal objecten is wat ten overvloede is meegenomen. Dit komt door twee redenen:

- De eerste groep van objecten die ten onrechte in Pluriform PDE terecht zijn gekomen zijn de objecten die door hun makers een verkeerde module of klasse pointer hebben gekregen. Dit was voor de ontwikkelaars op dat moment van minder belang omdat het slechts een richtlijn aangeeft. De intentie van een module pointer bijvoorbeeld was niet dat die op een gegeven moment zou worden gebruikt om een splitsing te maken.
- De splitsing is nu bepaald aan de hand van eigenschappen van een object. Er is niet gekeken naar de semantiek van een object. Wil je echt een scheiding maken zal er van elk object moeten worden gekeken naar zijn semantiek en naar de semantiek van de relaties die hij heeft met andere objecten.

Het uiteindelijk Pluriform PDE systeem heeft maar 93 van de 271 klassen over. Het systeem is hiermee van 88.473.000 naar 47.906.816 bytes gegaan, dit is slechts 54 %. Hierdoor is het gebruik van het geheugen aanzienlijk minder geworden.

Hoofdstuk 5: Conclusies

Pluriform PDE

Het uiteindelijk Pluriform PDE systeem is een systeem dat ongeveer de helft van een normaal systeem in neemt. Hierdoor is de benodigde hardware een stuk verminderd en de snelheid van het systeem opgevoerd. Met het systeem is gemakkelijk een simpel informatiesysteem te maken. Verder kun je er ook snel mee leren modelleren in Pluriform.

Er kan door de gebruiker zelf een eigen bibliotheek opgebouwd worden van nieuwe herbruikbare componenten. Mocht een gebruiker toch een aantal objecten willen gaan gebruiken die Pluriform Software al binnen zijn volledige pakket heeft dan kunnen deze componenten gemakkelijk worden toegevoegd door middel van fixregels. Verder kan het systeem gemakkelijk mee evolueren met de versies van het volledige systeem door gebruik te maken van het update mechanisme.

Naar mijn mening is het Pluriform PDE systeem een gemakkelijk systeem om te gaan gebruiken op de School voor Informatica omdat het zo snel te leren is en het een goede ondersteuning biedt aan een RAD of RIAD methode zoals die wordt geleerd op de School voor Informatica.

Verder onderzoek

Het Pluriform PDE systeem bevat nog een aantal objecten die niet noodzakelijk zijn. Hier zijn verschillende oorzaken voor aan te wijzen. Om deze objecten er ook nog uit te filteren zou er bijvoorbeeld gekeken moeten worden naar klassen die nog geen filter methode hebben en daar een extra module pointer aan toevoegen. Deze pointer moet dan per objectgevuld worden.

Verder kan het ook zo zijn dat ontwerpers per ongeluk de verkeerde module hebben ingevuld. Deze pointer was in beginsel niet bedoeld om een splitsing mee te maken. Deze pointer is in sommige gevallen puur een keuze die de ontwerper maakt. In deze gevallen kan er geen semantiek van afgeleid worden en is het een kwestie van de ervaring en het inzicht van de ontwerper die bepaald wat de waarde van de module wordt.

Een ander punt wat nog gedaan moet worden is van de andere modulen een selectie methode te maken. Op het moment dat een gebruiker meer modulen wil hebben dan die in dit onderzoek zijn geïdentificeerd dan zullen voor de klassen die vallen onder de nieuwe modulen ook methoden moeten worden gemaakt die bepalen welke objecten er wel en niet direct bij deze module horen.

Hoofdstuk 6: Referenties

[1] Inleiding modelleren G. Haverkamp 1999
[2] Pluriform Application Framework White Paper T. van Hugte may 1998