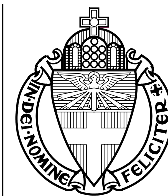
Een Model voor de Evolutie van Legacy Systemen naar OO

P.J. Koning



Katholieke *Universiteit* Nijmegen

Master's Thesis No. 335, 12 december 1994

Department of Informatics
Faculty of Mathematics and Informatics

Information
Systems

Een Model voor de Evolutie van Legacy Systemen naar OO

Afstudeerscriptie 335,

ter verkrijging van de graad doctorandus
aan de Katholieke Universiteit Nijmegen,
Faculteit der Wiskunde en Informatica,
Afdeling Informatiesystemen

door:

P.J. Koning

geboren te 12-12-1970

begeleider(s):

dr.ir. Th.P. van der Weide
ir. P.Chr. van Galen

12 december 1994

Inhoudsopgave

Samenvatting **1**

Voorwoord **3**

Dankwoord **5**

1 Het Object Model **7**

1.1 Objecten 7

1.2 Klassen 8

1.2.1 Attributen 8

1.2.2 Operaties 8

1.3 Links en Associaties 8

1.4 Generalisatie en Overerving 9

2 Hergebruik **11**

2.1 Een Introductie in Hergebruik 11

2.2 Hergebruik in de Object Georiënteerde Methode 12

2.2.1 De Compositionele Benadering 12

2.3 Verschillende Vormen van Hergebruik in de Object Georiënteerde Methode 12

2.3.1 Hergebruik binnen het Ontwikkelde Systeem 13

2.3.2 Hergebruik van Delen van andere Object Georiënteerde Systemen 13

2.4 Het Dilemma 13

3 Een Eerste Aanzet tot een Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd **15**

3.1 Een Eerste Aanzet tot een Architectuur 15

3.2 Een Eerste Aanzet tot een Methode 18

3.3 Voor- en Nadelen van de Methode 19

4 Een Verdere Afbakening **21**

5 Beschrijving van een RDBMS met SQL 23

5.1 RDBMS Logische Data Structure 23

5.2 RDBMS Operatoren 24

5.2.1 DDL Zinnen 25

5.2.2 DML Zinnen 27

5.2.3 Module Language Zinnen 28

5.3 RDBMS Integriteit 29

6 Een Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd 31

6.1 De Architectuur 31

6.2 De Methode 33

6.3 Voor- en Nadelen van de Methode 34

7 Stap 1: Analyseer Huidige Situatie 35

7.1 Spoor Programma's P_i op 35

7.2 Leg Tabel Structuren Vast 35

7.3 Leg Afhankelijkheid Vast tussen Programma's P_i en Tabellen TB_i 36

8 Stap 2: Maak Object Model Toekomstige Situatie 37

8.1 Volledige Analyse 37

8.2 Van Relationeel Model naar Object Model 38

8.2.1 De Methode 38

8.2.2 De Voordelen 38

8.2.3 De Nadelen 38

9 Stap 3: Omzetten Object Model naar Relationeel Model 39

9.1 Twee Mogelijke Implementaties van het Object Model in SQL 39

9.1.1 One Table Approach 39

9.1.2 More Table Approach 39

9.1.2.1 Van Klassen naar Tabellen 40

9.1.2.2 Van Binaire Associaties naar Tabellen 40

9.1.2.3 Van Ternaire Associaties naar Tabellen 41

9.1.2.4 Van Qualified Associaties naar Tabellen 41

9.1.2.5 Van Generalisatie naar Tabellen 41

9.2 Voor- en Nadelen 43

9.3 Maak Tabellen 44

10 Stap 4: Maak Nieuwe en Virtuele Oude Database 49
10.1 Een Inventarisatie . . . 49
10.1.1 De Oude Tabellen . . . 49
10.1.2 De Nieuwe Tabellen . . . 49
10.2 De Mapping . . . 50
10.2.1 Attribuut Mapping . . . 50
10.3 De Migratie . . . 52
10.3.1 Maak Nieuwe Tabellen uit de Oude Tabellen . . . 52
10.3.2 Maak Virtuele Oude Tabellen uit de Nieuwe Tabellen . . . 52

11 Stap 5: Migreer Oude Software 55
11.1 Migratie van Programma's P_i . . . 55
11.2 Migratie van een Deel van P_i . . . 55

12 Tips voor Management van de Methode 57
12.1 Doelgebied . . . 57
12.2 Risico Factoren . . . 58
12.3 Project Scheduling . . . 60

13 Verder Onderzoek 61
13.1 Mapping Tools . . . 61
13.2 Forward Gateway . . . 61
13.3 Specificatie Extractor . . . 61
13.4 Automatische Afhankelijkheids Analysator . . . 61
13.5 SQL-3 en SQL-4 . . . 61
13.6 Organisatorische Eisen voor Slagen van Migratie . . . 62
13.7 Performance van View Mechanisme . . . 62
13.8 Integratie Verschillende DBMS-en . . . 62
13.9 Definitie Problemen bij Integratie van Verschillende Databases . . . 62

14 Conclusies 63

Auteurs Index 67

Samenvatting

Een belangrijke vorm van hergebruik binnen de object georiënteerde methode, hergebruik van oude, niet object georiënteerde systemen, wordt tot nu toe steeds over het hoofd gezien. Ondanks de andere vormen van hergebruik die de object georiënteerde methode claimt te ondersteunen, hergebruik van componenten van andere object georiënteerde systemen en hergebruik van componenten binnen het te ontwikkelen systeem zelf, wordt de object georiënteerde methode in het bedrijfsleven nog weinig toegepast. Juist het gemis aan onderzoek naar hergebruik van oude, niet object georiënteerde systemen is één van de belangrijkste oorzaken dat bedrijven een afwachtende houding aannemen ten opzichte van de object georiënteerde methode. Bedrijven zitten met duizenden regels code, die in de loop van de jaren ontwikkeld zijn. Ze hebben miljoenen guldens geïnvesteerd in het ontwikkelen van deze programmatuur. Worden al deze investeringen weggegooid bij een omschakeling naar de object georiënteerde methode?

In deze scriptie wordt een architectuur ontwikkeld, die zowel de oude systemen en de nieuwe (object georiënteerde) systemen naast elkaar laat bestaan en de mogelijkheid biedt om de oude systemen stap voor stap te migreren naar een object georiënteerde omgeving. Bedrijven kunnen op deze manier doorgaan met functioneren en de oude investeringen blijven zo behouden. Vandaar dat het doel van deze scriptie is: het ontwerpen van een architectuur en methode die het mogelijk maken om legacy systemen (stap voor stap) te migreren naar een object georiënteerde omgeving.

In het eerste hoofdstuk zal het object model, zoals dat in deze scriptie gebruikt wordt, beschreven worden. Het zal veel overeenkomsten vertonen met Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91], omdat deze methode taal-onafhankelijk is. Het tweede hoofdstuk gaat eerst in op hergebruik in het algemeen, spitst zich daarna toe op het object georiënteerd model en sluit vervolgens af met een pleidooi voor onderzoek naar een nieuwe vorm van hergebruik; hergebruik van oude, niet object georiënteerde systemen. Het derde hoofdstuk maakt een eerste aanzet tot een model waarin de oude systemen kunnen blijven bestaan naast het nieuwe systeem in de object georiënteerde omgeving. Hoofdstuk vier spoort een aantal gevallen op waarin de eerste aanzet die in hoofdstuk drie gegeven is, toegepast kan worden. Dit hoofdstuk sluit af met een beredeneerde keuze voor het *RDBMS naar RDBMS* geval. Hoofdstuk vijf beschrijft het relationele database model. Hoofdstuk zes bekijkt de invloed van de keuze voor het relationele naar relationele geval op respectievelijk de architectuur en de methode. De hoofdstukken zeven tot en met elf geven een gedetailleerde beschrijving van de methode zoals die in hoofdstuk zes voorgesteld is. Hoofdstuk twaalf geeft het doelgebied, de risico factoren en de project scheduling van de methode, die in deze scriptie beschreven wordt, weer. Hoofdstuk dertien kijkt naar gebieden voor mogelijk verder onderzoek. Het laatste hoofdstuk zal een aantal conclusies geven over de architectuur en de methode die in het kader van mijn afstudeerscriptie ontwikkeld zijn.

Voorwoord

Zo'n drie jaar geleden kwam ik in aanraking met het fenomeen object oriëntatie. Aan de Katholieke Universiteit Nijmegen volgde ik alle vakken uit de GIP[1]-lijn. Eerst was ik lid van een project-team dat een object georiënteerd ontwerp maakte. Daarna werd ik technisch manager van een project-team dat dit ontwerp ging implementeren en in GIP4 was ik lid van een Quality Assurance Team. Dit team bood ondersteuning aan een GIP-groep die een object georiënteerd ontwerp maakte. Om mijn kennis van het object georiënteerd paradigma te vergroten, heb ik het vak SE2[2] bij prof. C.H.A. Koster gevolgd en vervolgens de vakken Object Oriented Systems en Software Analyse Ontwerp bij dr.ir. M. Aksit aan de Technische Universiteit Twente. Langzamerhand raakte ik onder de indruk van de voordelen van het object georiënteerd paradigma. Wat mij echter opviel in gesprekken met mensen uit het bedrijfsleven was het feit dat het object georiënteerd paradigma nog weinig in bedrijven toegepast wordt. Men wil wel overschakelen maar een overschakeling wordt bemoeilijkt door het feit dat alle oude conventionele systemen gewoonweg niet weggegooid kunnen worden. Weggooien van deze systemen zou een enorme kapitaalvernietiging met zich meebrengen. Vaak is het ook nog zo dat bedrijven afhankelijk zijn geworden van deze systemen; voor deze bedrijven is het al helemaal niet mogelijk om met een schone lei te beginnen. Bij mij ontstond zo de behoefte om onderzoek te doen naar een mogelijke oplossing voor dit probleem en ik heb deze behoefte gekoppeld aan een afstudeeronderzoek bij de vakgroep van dr.ir. Th.P. van der Weide.

De doelstelling van mijn onderzoek is:

> Het ontwerpen van een architectuur en methode die het mogelijk maken om legacy systemen (stap voor stap) te migreren naar een object georiënteerde omgeving.

Dat dit een groot probleem is voor het bedrijfsleven blijkt uit mijn stage bij PTT Telecom B.V. in Groningen. Ik mocht meekijken bij het Tripoli-1 project dat kijkt hoe men oude systemen kan migreren naar een nieuwe, in dit geval component georiënteerde, omgeving. Dat het succes van de migratie niet alleen afhankelijk is van technische, maar vooral ook van organisatorische, kennis en politieke factoren, blijkt uit het Tripoli-1 project. De omschakeling naar een nieuw paradigma, in het Tripoli-1 geval een component georiënteerd paradigma, is meer dan alleen het leren van dit paradigma. De bevindingen van het Tripoli-1 project zullen in deze scriptie verweven zijn. De stage is twee maanden eerder dan gepland beëindigd in overleg met projectleider ir. C.J.H. Nieuwenhuis en stagebegeleider drs. A. Last. Door de problemen die het Tripoli-1 project onderweg ondervond, zijn de overeenkomsten tussen mijn scriptie en het Tripoli-1 project dusdanig uit elkaar gegroeid dat tot een beëindiging van de stage is besloten. Het contact tussen het Tripoli-1 project-team en mij zijn wel behouden gebleven in de vorm van een verificatie van mijn scriptie door een aantal leden van het project-team.

Ook heb ik mijn onderzoeksresultaten gepresenteerd op een werkbijeenkomst van FENIT in Maarssenbroek. De titel van deze werkbijeenkomst was 'Aan de slag met Object Oriëntatie'. Uit de discussies die op deze werkbijeenkomst ontstonden bleek dat het 'legacy' probleem voor een heleboel bedrijven een *hot* item is. Ik denk dan ook dat er meer onderzoek naar dit probleem gedaan moet worden. Ik hoop dat mijn scriptie een 'eerste' goede aanzet is in de 'legacy' problematiek.

[1] GeÏntegreerd Practicum
[2] Software Engineering 2

Dankwoord

Tijdens mijn afstuderen heb ik met veel mensen over het 'legacy' probleem gepraat. Op deze plaats wil ik hen dan ook allemaal bedanken:

- dr.ir. Th.P. van der Weide, afstudeerbegeleider K.U.N.
- ir. P.Chr. van Galen, FENIT, voor het bezorgen van een stageplaats bij PTT Telecom B.V. afdeling I&AT en het geven van waardevolle op- en aanmerkingen op mijn scriptie.
- prof. dr. W. Gerhardt-Häckl van de TU in Delft voor haar opmerkingen.
- dr. H. Li, PTT Telecom B.V. afdeling I&AT Leidschendam voor zijn op- en aanmerkingen.
- drs. F.A. Grootjen, medewerker K.U.N., voor zijn opmerkingen.
- W.J. Dikkers, PTT Telecom B.V. afdeling I&AT Groningen.
- drs. A. Last, mijn stagementor bij PTT Telecom B.V. afdeling I&AT Groningen.
- ir. C.J.H. Nieuwenhuis, projectleider Tripoli-1 project bij PTT Telecom B.V. afdeling I&AT Groningen.
- Het hele Tripoli-1 project team van PTT Telecom B.V. afdeling I&AT Groningen:
  - R.M.P. de Bruijn
  - drs. E.J.H. Couwenberg
  - J. Dreteler
  - J.H. Hemelt
  - drs. A. Last
  - ing. A. van der Meulen
  - ir. R.E. Siegel
  - ir. D. Valk
  - drs. S.G. Vrind

Hoofdstuk 1

Het Object Model

In dit hoofdstuk wordt het object model (OM) beschreven zoals dat in deze scriptie gebruikt wordt. Het OM zal veel overeenkomsten vertonen met het OM van Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91]. Bij Rumbaugh et al. is het OM één van de drie modellen die door de Object-oriented Modeling and design Technique (OMT) gebruikt worden. Het kan vergeleken worden met de data modellen die gebruikt worden in de functionele benaderingen. De grootste verschillen tussen een data model en een object model zijn de extra concepten a) operaties en b) identiteit. In een data model zijn de operaties geen onderdeel van het model, terwijl in een object model operaties wel een onderdeel zijn van het model. Identiteit is die eigenschap van een object dat hem onderscheidt van andere objecten.

De keuze om het OM van Rumbaugh et al. als leidraad te nemen, is te verklaren uit het feit dat het OMT van Rumbaugh et al. programmeertaal-onafhankelijk is. Dit in tegenstelling tot bijvoorbeeld Meyer[Mey88] die bepleit dat de gebruikte taal in het ontwerp proces geïntegreerd moet zijn omdat zo oppervlakkige resultaten vermeden worden. Maar steeds zal getracht worden om de gebruikte concepten zo te definiëren, dat ze in overeenstemming zijn met de bekende methoden zoals die van Booch[Boo91], Jacobson, Christerson, Jonsson en Van Overgaard[JCJvO92], Rumbaugh et al.[RBP+91] en Meyer[Mey88].

Het OM wordt gebruikt om een deel van de reële wereld, het *Universe of Discourse* (UoD), te beschrijven. In het UoD zijn entiteiten (objecten) aanwezig. Sommige van deze objecten hebben dingen gemeen. De objecten met gemeenschappelijke eigenschappen worden gegroepeerd in een klasse. Over zo'n klasse willen we informatie vastleggen. Als we bijvoorbeeld vast willen leggen dat alle personen namen hebben, dan is de **naam** één eigenschap (attribuut) van de klasse **persoon**. De klasse **persoon** geeft dan informatie over een object in deze klasse. Andere informatie kan weer afgeleid worden uit wat een object in een klasse kan doen (operaties). Bijvoorbeeld, een persoon kan **lopen**, **slapen** of **praten**. Ook willen we informatie vast kunnen leggen over relaties tussen klassen (associaties). Bijvoorbeeld, een **persoon** kan werken voor een **bedrijf**. De woorden die tussen haakjes staan zijn de belangrijkste concepten van een OM en zullen in de volgende paragrafen behandeld worden.

1.1 Objecten

Het middelpunt van het OM zijn *objecten*. Een object is een abstractie in het UoD; "Objects are things with crispy boundaries" (zie Booch[Boo91]). Alle objecten hebben een unieke *identiteit*, wat wil zeggen dat ze altijd van andere objecten te onderscheiden zijn. "Identity is that property of an object which distinguishes it from all other objects", zoals Khoshafian en Copeland[KC86] definiëren. Een consequentie hiervan is dat objecten, die dezelfde eigenschappen hebben, toch verschillende objecten zijn.

De consequentie van het hebben van een identiteit is dat, ook als alle eigenschappen van het object veranderen, het object nog steeds dat ene unieke object blijft. Het woord *object* wordt

vaak gebruikt met verschillende betekenissen. Soms betekent object één ding, andere keren weer duidt het op een groep gelijksoortige dingen. Als in deze scriptie gesproken wordt over precies één ding dan wordt het woord *object* gebruikt. We gebruiken de term *klasse* als we het over een groep objecten met dezelfde eigenschappen hebben.

1.2 Klassen

Een klasse beschrijft een groep objecten met overeenkomstige eigenschappen en gedrag. Jacobson et al.[JCJvO92] definiëren het begrip klasse als volgt:

> A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure.

Het concept klasse is puur een abstractie middel en stelt ons in staat om van een paar specifieke voorbeelden een generalisatie te maken. Meyer[Mey88] geeft een duidelijk verschil aan tussen klassen en objecten:

> Het is belangrijk het onderscheid tussen classes en objecten goed voor ogen te houden. Objecten zijn elementen die tijdens de executie van een systeem worden gecreëerd; classes zijn een puur syntactische beschrijving van een verzameling mogelijke objecten - de instanties van de class. Tijdens uitvoering van een systeem bestaan er alleen objecten; in een programma zien we alleen maar classes.

Hiermee wordt meteen het nut van klassen duidelijk. Door het groeperen van objecten in klassen abstraheren we het probleem. Abstractie geeft modelering zijn kracht en de mogelijkheid om te generaliseren van een paar specifieke objecten naar één klasse die alle betreffende objecten omvat. Soortgelijke definities worden maar één keer per klasse opgeschreven in plaats van voor elke instantie van die klasse (objecten).

1.2.1 Attributen

De reden waarom objecten gegroepeerd worden in klassen is dat ze soortgelijke eigenschappen hebben. Deze eigenschappen worden *attributen* genoemd. Attributen relateren data-waarden aan objecten. Elk attribuut behoort tot een bepaalde klasse en heeft een unieke naam binnen de klasse.

1.2.2 Operaties

Operaties zijn functies of transformaties die uitgevoerd kunnen worden op objecten in een klasse. We noemen dit het gedrag van objecten. Alle objecten in een klasse delen dezelfde operaties.

1.3 Links en Associaties

Tot nu toe hebben we alleen groepen objecten (klassen) beschreven. Klassen in het UoD staan niet op zich. Klassen kunnen onderling relaties hebben.

Een *link* is een fysieke of conceptuele verbinding tussen objecten. Een *associatie* beschrijft een groep links met overeenkomstige structuur en semantiek. Een link is dus een instantie van een associatie. Een link speelt zich op het object niveau en een associatie op het klasse niveau af.

Een *qualified associatie* relateert aan twee objecten een *qualifier*. Een qualifier is een speciaal attribuut dat de meervoudigheid van een associatie reduceert. One-to-many associaties en many-to-many assocaties kunnen qualified zijn. Een qualified associatie kan gezien worden als een soort ternaire associatie.

Een *ternaire associatie* is een relatie tussen drie klassen.

De begrippen associatie en link zijn afkomstig van Rumbaugh et al.[RBP+91]. Booch[Boo91] heeft een soortgelijke verbinding tussen klassen die relaties worden genoemd. Meyer[Mey88] praat niet over links, associaties of relaties. Hij is ervan overtuigd dat alles als objecten gezien moet worden, dus ook relaties tussen klassen.

1.4 Generalisatie en Overerving

Generalisatie is de relatie tussen een klasse en één of meerdere verfijnde versies ervan. De klasse die verfijnd wordt, wordt *superklasse* genoemd. Elke verfijnde versie ervan wordt *subklasse* genoemd. Het voordeel van generalisatie is dat elke operatie maar één keer gemaakt hoeft te worden, alle andere klassen erven deze gewoon.

Overerving (inheritance) kan ook meervoudig gebeuren, dit wordt *multiple inheritance* genoemd. Een klasse kan meer dan één superklasse hebben. Het voordeel van meervoudige overerving is dat je krachtiger kunt modelleren. Het brengt het object model dichter bij hoe de mens zelf denkt. Het nadeel ervan is het verlies van conceptuele en implementatie eenvoud.

Hoofdstuk 2

Hergebruik

In het midden van de jaren '70 ondervonden software ontwikkelaars een groot aantal problemen; projecten waren te laat klaar, software was te duur, software werd te complex om met de huidige methoden aan te kunnen en de ontwikkelde software voldeed niet aan de eisen van de gebruikers (zie Sommerville[Som89]). Om deze zogenaamde *Software Crisis* te kunnen stoppen, is men op zoek gegaan naar nieuwe ontwikkelmethoden. De software die met behulp van zo'n nieuwe methode ontwikkeld zou gaan worden, zou aan een aantal eisen moeten voldoen:

- De software zou makkelijk onderhoudbaar en uitbreidbaar moeten zijn (maintainability en extendibility).
- Delen software moeten herbruikbaar zijn (reusebility) .
- Software zou makkelijk gecombineerd moeten kunnen worden met andere software (compatibility).
- Software moet een goed user-interface bieden (usebility).

In dit hoofdstuk wordt het begrip herbruikbaarheid naderbekeken.

2.1 Een Introductie in Hergebruik

Dusink en Hall[DH89] definiëren de term *re-use*:

> Re-use is considered as a means to support the construction of new programs using in a systematical way existing designs, design fragments, program texts, documentation, or other form of program representation.

Ook Meyer[Mey88] heeft een soortgelijke definitie:

> Herbruikbaarheid is de eigenschap van een software produkt dat het opnieuw gebruikt kan worden, geheel of in delen, voor nieuwe, andere toepassingen

Een systeem bestaat uit delen (componenten). Componenten kunnen oplossingen representeren van ontwerpproblemen, architectuurproblemen of coderingsproblemen. Ze kunnen gebonden worden aan een specifiek toepassingsgebied en implementeren bepaalde functies en operaties in dat domein. Deze delen kunnen op twee manieren hergebruikt worden:

- *transformationele benadering*; programma's worden geschreven als abstracte specificaties in een speciale taal. Deze talen zijn domein specifiek en transformeren de abstracte specificaties automatisch in efficiënte programma's door gebruik te maken van domein kennis in de transformatie regels en door gebruik te maken van bestaande componenten.

- *compositionele benadering*; software componenten worden gebruikt als basis bouwstenen in het ontwikkelingsproces van software. Programma's worden geconstrueerd door bestaande software componenten te combineren.

Het voordeel van hergebruik bevindt zich op verschillende vlakken. Tijdens ontwerp en implementatie komen minder fouten voor doordat betrouwbare componenten gebruikt kunnen worden. De software kan sneller opgeleverd worden doordat het ontwerpen niet vanaf een nulpunt hoeft te beginnen. Ook worden er minder fouten gemaakt doordat complexiteit verborgen is binnen de componenten.

2.2 Hergebruik in de Object Georiënteerde Methode

De object georiënteerde methode claimt ondersteuning van hergebruik en verhoging van de efficiëntie van het ontwerpen van software door het verlagen van ontwerp-, implementatie- en testkosten. Hergebruik in de object georiënteerde methode gaat volgens de *compositionele benadering*. Vandaar dat deze nader bekeken wordt.

2.2.1 De Compositionele Benadering

In de compositionele benadering zoekt en combineert de ontwikkelaar kant-en-klare componenten om grotere componenten of programma's te maken. De algorithmen en specificaties moeten begrepen worden door de programmeur.

Binnen de compositionele benadering kan hergebruik toegepast worden als black-box, glass-box of als white-box proces. Als de black-box methode gebruikt wordt, zijn alleen de interface en de specificatie van de componenten bekend. Met de glass-box methode is ook de inhoud van de componenten zichtbaar maar dit alles kan niet veranderd worden. De white-box methode laat ook verandering toe van de componenten (zie Dusink en Hall[DH89] voor een uitgebreide beschrijving van deze terminologie). Deze verandering kan gedaan worden door gebruik te maken van een editor. Ook kunnen delen van de implementatie overschreven worden door nieuwe code.

Dat deze vorm van hergebruik niet nieuw is blijkt uit een citaat van McIlroy[McI76]:

> Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules, as there are catalogs of VSLI devices: when we build a new system, we should be ordering components from these catalogs and combining them, rather than reinventing the wheel every time. We would write less software, and perhaps do a better job at that which we do get to develop. Wouldn't then some of the problems that everybody laments - the high costs, the overruns, the lack of reliability - just go away? Why isn't it so?

Hij uitte reeds in 1968, tijdens de nu beroemde NATO-workshop over de software crisis, de droom van in massa geproduceerde softwarecomponenten.

2.3 Verschillende Vormen van Hergebruik in de Object Georiënteerde Methode

Tot nu toe werden er binnen het object georiënteerd paradigma twee belangrijke vormen van hergebruik onderscheiden:

- Hergebruik binnen het ontwikkelde systeem.
- Hergebruik van delen van andere object georiënteerde systemen.

In Meyer[Mey88] leidt het thema hergebruik zelfs tot de technieken van zijn object georiënteerde methode.

2.3.1 Hergebruik binnen het Ontwikkelde Systeem

Operaties en attributen worden maar één maal ontworpen en geïmplementeerd.

Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91] zeggen hierover:

> Sharing of code within a project is a matter of discovering redundant code sequences in the design and using programming language facilities, such as procedures or methods, to share their implementation. This kind of code sharing almost always pays off immediately by producing smaller programs, faster debugging, and faster iteration of the design.

Booch[Boo91] schat zelfs dat een programma 33 procent kleiner wordt door hergebruik:

> From our experience, a reasonably complete problem reporting system consists of about 20-30,000 lines of C++. Without C++, and without our fierce dedication to seeking out commonality among key abstractions and mechanisms, we would expect a non-object-oriented version to be about 50 percent larger.

2.3.2 Hergebruik van Delen van andere Object Georiënteerde Systemen

Klassen die al eens geschreven zijn, kunnen hergebruikt worden in het nieuwe, te ontwikkelen systeem.

Deze vorm van hergebruik wordt in alle ontwikkelmethoden nagestreefd. Rumbaugh et al.[RBP+91] zeggen in de analyse fase:

> Reuse a module from a previous design if possible, but avoid forcing a fit. Reuse is easiest when part of the problem domain matches a previous problem. If the new problem is similar to a previous problem but different, the original design may have to be extended to encompass both problems. Use your judgement about whether this is better than building a new design.

Booch[Boo91] zegt:

> Actively looking for reusable software components that are relevant to a new system is a very important activity in any development. This process is facilitated by rich class libraries that are typically available for object-based and object-oriented programming languages.

Jacobson, Christerson, Jonsson en Van Overgaard[JCJvO92] pleiten zelfs voor nog meer hergebruik:

> The necessity of reusability is of course applicable during coding, since it can influence productivity significantly. This is the usual context when software people talk about reuse. So there is no reason that prevents us having it on code level, but it is actually not the only interesting type of reuse in software engineering. What can give even higher productivity enhancement is reuse in other development phases. Other parts of the construction phase may benefit when reusing entire designs in several systems. Additionally reuse should also be viewed as natural during analysis and testing.

2.4 Het Dilemma

In de meeste bedrijven zullen er informatie systemen aanwezig zijn die niet object georiënteerd zijn. Deze systemen zijn door de vele veranderingen zo complex geworden dat elke verandering aan het programma zorgvuldig voorbereid moet worden. Vaak worden veranderingen niet uitgevoerd omdat men de gevolgen van deze veranderingen niet meer kan overzien. Een methode die deze

situatie probeert te voorkomen, zou een mogelijke uitkomst bieden voor die bedrijven. De object georiënteerde methode claimt de uitkomst te zijn.

Aan de ene kant willen de bedrijven graag overschakelen van conventionele methoden naar de object georiënteerde methode, maar aan de andere kant vragen de bedrijven zich af waar alle investeringen blijven die gedaan zijn in de bestaande software. Moet alles van de grond af aan opgebouwd worden met de object georiënteerde methode? Worden zo alle investeringen dus weggegooid? Vaak kunnen de bedrijven zo'n omschakeling zich niet eens veroorloven omdat er een te grote afhankelijkheid bestaat tussen de bestaande software en het functioneren van het bedrijf. Dat deze situatie niet ideaal is en moet veranderen, zal de lezer duidelijk zijn. Vandaar dat er onderzoek gedaan moet worden naar deze problematiek.

Er moet een manier gevonden worden om de conventionele methoden en de object georiënteerde methode te combineren. De term **hergebruik** speelt naar mijn mening de belangrijkste rol in de oplossing van deze problematiek. Door het hergebruik van zoveel mogelijk van deze bestaande systemen (niet object georiënteerde systemen) zou een aanzienlijke verbetering verwezenlijkt kunnen worden. Men zou een omschakeling kunnen maken van starre systemen naar flexibele (object georiënteerde) systemen.

Hoofdstuk 3

Een Eerste Aanzet tot een Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd

In dit hoofdstuk zal een eerste aanzet gegeven worden om te komen tot een model voor de evolutie van legacy systemen naar OO. Er zal een architectuur worden toegelicht die het mogelijk maakt om zowel de conventionele wereld als de object georiënteerde wereld naast elkaar te gebruiken. Vervolgens wordt een methode voorgesteld die aan de hand van eenvoudige stappen deze architectuur realiseert.

Er wordt vanuit gegaan dat er een object model aanwezig of te ontwerpen is. Er is geen onderzoek gedaan naar de kosten van de invoering van zo'n architectuur. Ook wordt er in de verdere scriptie vanuit gegaan dat de oude situatie één database kent. Hierdoor zijn er geen definitieproblemen in de oude situatie. Dit maakt het toepassingsgebied natuurlijk smaller, maar een beperking is noodzakelijk vanwege de beperkte tijd die aan deze scriptie besteed kan worden.

3.1 Een Eerste Aanzet tot een Architectuur

We zien elk systeem opgebouwd zoals in Figuur 3.1.

We hebben dus:

$Programma's : P_1....P_n$ (Er bestaat geen afhankelijkheid tussen de P's)

De verzameling van Programma's P_i noemen we P:

$P = \{P_1...P_n\}$

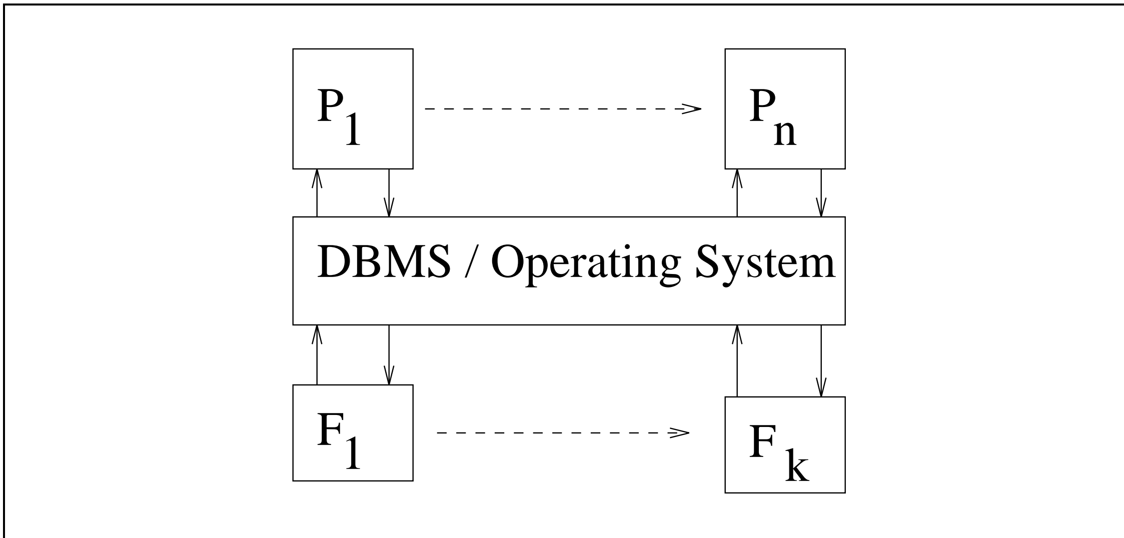
$Files : F_1...F_k$

De verzameling van Files F_i noemen we de DataBase DB:

$DB = \{F_1...F_k\}$

Een file F is een sequentie van tuples:

$F = \{< t_1...t_n > | t_i \in T\}$



Figuur 3.1: Elk systeem is als volgt opgebouwd.

Waarbij T de verzameling van alle mogelijke tuples is. Een tuple $t_i \in T$ is een sequentie van gegevens:

$t_i = \{< g_1 ... g_n > | g_i \in Pop(G_i)\}$

Waarbij G_i de naam van de kolom is en $Pop(G_i)$ de verzameling van alle mogelijke invullingen van de kolom.

Uiteindelijk willen we van deze architectuur omschakelen naar een architectuur waarbij alle programma's $P_1 ... P_n$ vervangen zijn door programma's $P'_1 ... P'_o$ die object georiënteerd zijn. Doordat er een object model gemaakt wordt, kan het zijn dat ook de DB omgezet moet worden naar een DB' die het object model opslaat. Daar we onder andere de gegevens van DB willen hergebruiken moeten we een mapping maken tussen DB en DB':

$mapping : DB \rightarrow DB'$

De database DB wordt aangesproken door een query-taal Q. Deze Q kan ook gewoon het operating system zijn. De nieuwe database DB' wordt aangesproken door een query-taal Q'. Er moet nu gelden dat voor elke zin uit Q er een zin uit Q' is, zodat het resultaat van de twee zinnen gelijk is.

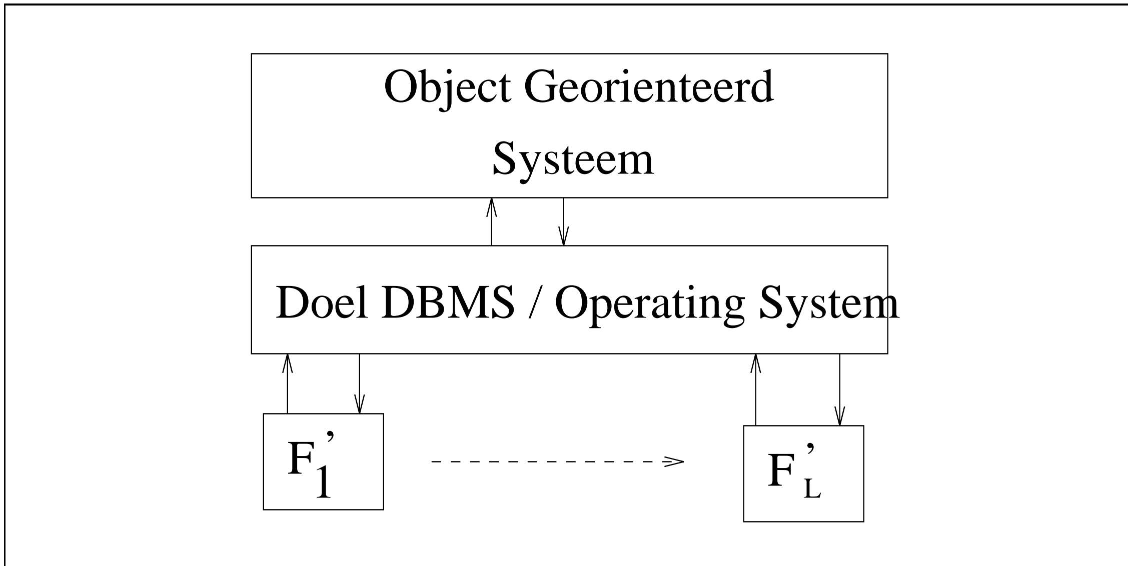
voor elke $q \in Q$ is er een $q' \in Q'$ zodat $resultaat(q) = resultaat(q')$

In Figuur 3.2 zien we de architectuur die we uiteindelijk willen realiseren.

Daar we niet alleen de oude database DB willen hergebruiken maar ook de programma's $P_1 ... P_n$ moeten we een *Forward Gateway* maken. Het idee van een Forward Gateway wordt in Brodie en Stonebraker[BS93] beschreven.

> By a Gateway we mean a software module introduced between operational software components to mediate between them.

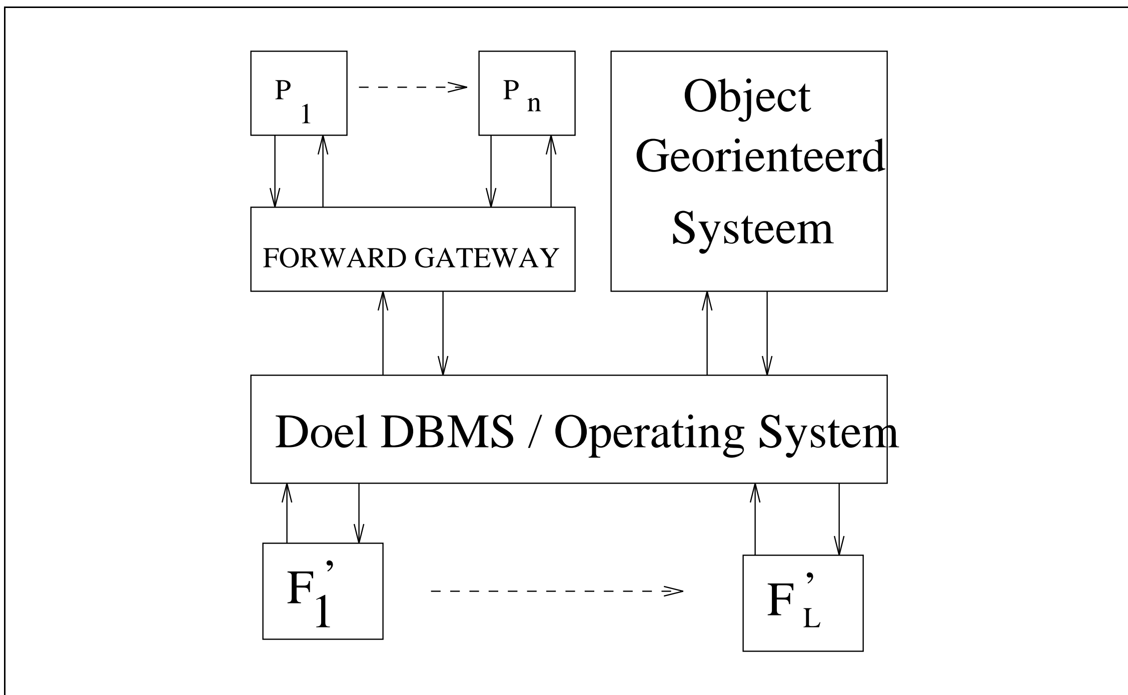
In deze scriptie wordt de Forward Gateway gebruikt als een tolk. Het vertaalt de oude call's van een programma op een database naar nieuwe call's op de nieuwe target database. Formeel beschreven, hebben de oude programma's $P_1 ... P_n$ via een acces a toegang tot de oude DB. We moeten nu acces a op DB omvormen tot een acces a' op DB'.



Figuur 3.2: De Uiteindelijke Architectuur; deze situatie willen we in de toekomst realiseren.

We doen dit met behulp van een Forward Gateway FG.

$FG : a_{DB} \rightarrow a'_{DB'}$



Figuur 3.3: De Tussen Fase Architectuur; als tussen-fase willen we de oude programma's P_i naast de nieuwe situatie laten bestaan.

In Figuur 3.3 zien we dat de DB hergebruikt is en dat we geleidelijk elke P_i kunnen herimplementeren naar een object georiënteerde omgeving.

Aan de hand van de architectuur kunnen we een aantal stellingen formuleren:

1. DB is hergebruikt.
2. Programma's $P_1...P_n$ kunnen hergebruikt worden, als er een *Forward Gateway* is.
3. Elke P_i kan naar een object georiënteerde omgeving gemigreerd worden.
4. Er is een situatie waarbij alle $P_1...P_n$ gemigreerd zijn (Dit volgt automatisch uit bewering 3).

3.2 Een Eerste Aanzet tot een Methode

In de vorige paragraaf is een architectuur voorgesteld die de oude database en de oude code hergebruikt en het mogelijk maakt om geleidelijk over te schakelen naar OO. In deze paragraaf wordt een plan voorgesteld dat, aan de hand van duidelijk geformuleerde stappen, een geleidelijke overschakeling mogelijk maakt van oude starre systemen naar flexibele (object georiënteerde) systemen. Deze eerste aanzet zal het *Interface Approach Methode* genoemd worden.

Het *Interface Approach Method* omvat de volgende stappen:

1. Analyseer Huidige Situatie. In deze stap worden code en data van het oude systeem geanalyseerd. Dat deze stap in de praktijk een moeilijke en tijdrovende bezigheid kan zijn, blijkt uit mijn ervaringen in het Tripoli-1 project. Hier zaten delen van de data specificaties verborgen in de code zelf. Het analyseren van de data is dan zeker geen triviale bezigheid.
2. Maak Object Model Toekomstige Situatie. De toekomst wordt gemodelleerd in een object model.
3. Kies Gewenst DataBase Management Systeem.
4. Maak Mapping Tussen Oude en Nieuwe DataBase. Maak een **mapping table** die een vertaling maakt van oude database service calls naar de nieuwe service calls, alsook van oude database items naar nieuwe attribuutwaarden. Het maken van zo'n mapping kan in bepaalde situaties zeer ingewikkeld zijn. Bij het Tripoli-1 project deden zich definitieproblemen voor waarbij het vinden van een algemene definitie die voor zowel de oude als de nieuwe situatie een moeilijke zaak was, vooral ook omdat de oude situaties moesten kunnen blijven draaien.
5. Maak Forward Gateway
6. Migreer Oude Software. Delen van de oude software kunnen gemigreerd worden naar de nieuwe omgeving.

De Interface Approach Method bestaat eigenlijk uit drie delen:

1. *Hergebruik van data*: Stap 1 tot en met 4
2. *Hergebruik van code*: Stap 5
3. *Vervanging van slechte code*: Stap 6

Het is mogelijk om alleen stap 1 tot en met 4 uit te voeren. Dit is aan te raden als men in één keer naar object georiënteerde omgeving wil overstappen. Dan ontstaat de situatie van Figuur 3.2.

3.3 Voor- en Nadelen van de Methode

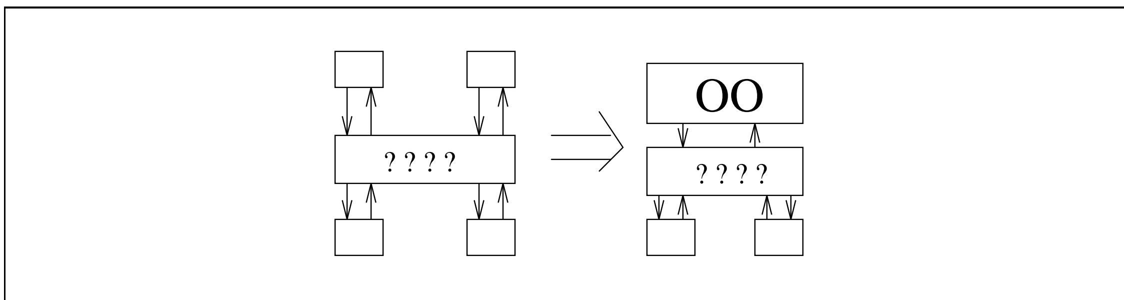
De Interface Approach Method voorgesteld in deze scriptie kent een aantal belangrijke voordelen ten opzichte van andere methoden (*reusable code recovery* in Ning, Engberts en Kozaczynski[NEK94], *encapsulating the whole system* in Ning, Engberts en Kozaczynski[NEK94] en de verschillende methoden beschreven in Dusink en Hall[DH89]).

- **Voordelen:**
  - *Veiligheid.* Omdat de oude software niet verandert, blijven de oude functies gelijk.
  - *Weinig inspanningen in oude code.* De oude code hoeft niet nauwkeurig geanalyseerd te worden.
  - *Kort migratie pad.* Het is niet nodig om de oude code nauwkeurig te analyseren en te begrijpen.
  - *Er is een stapsgewijze migratie mogelijk van het oude systeem naar een object georiënteerde omgeving.*
  - *Er ontstaat een flexibel systeem.* Doordat er stap voor stap gemigreerd wordt naar een object georiënteerde omgeving.
- **Nadelen:**
  - *Forward Gateway is moeilijk te maken.*
  - *Meenemen van ontwerpfouten niet mogelijk.* Doordat de oude code zo min mogelijk geanalyseerd wordt, leert men de tekortkomingen van de oude systemen niet kennen.

Hoofdstuk 4

Een Verdere Afbakening

In de voorgaande hoofdstukken is een eerste aanzet gegeven tot een model voor de evolutie van legacy systemen naar OO. Deze eerste aanzet kan voor verschillende situaties verder onderzocht worden. Als we naar Figuur 4.1 kijken, zien we aan de vraagtekentjes welke delen nog ingevuld moeten worden om het verhaal concreet te kunnen maken.



Figuur 4.1: Een eerste aanzet.

We kunnen onder andere de volgende situaties onderscheiden:

- van RDBMS naar ODBMS
- van RDBMS naar RDBMS
- van Flat Files naar ODBMS
- van Flat Files naar RDBMS

De oude situatie kent legio mogelijke invullingen van de vraagtekentjes. In deze scriptie gaan we uit van een RDBMS met SQL als de oude situatie, omdat bij de oude situaties in de praktijk een RDBMS met SQL vaak aanwezig is.

Kijken we echter naar de toekomstige situatie dan zien we dat we óf naar een ODBMS óf naar een RDBMS willen gaan. De overschakeling naar een ODBMS heeft uiteraard de voorkeur omdat het uiteindelijke systeem object georiënteerd zal zijn. Een overschakeling naar een RDBMS heeft als nadeel dat er een vertaling gemaakt moet worden van de OO wereld naar de relationele wereld (*impedantie probleem*). Het voordeel van de overschakeling naar een RDBMS is dat bij veel bedrijven deze faciliteit al aanwezig is. Er hoeft dus geen extra investering gedaan te worden in een dure ODBMS. De Forward Gateway blijkt te vervallen door gebruik te maken van het *view* mechanisme van SQL. RDBMS-en bestaan al vele jaren en is daardoor een bewezen concept. Het gebruik van een ODBMS kent een aantal belangrijke nadelen. De ODBMS-markt staat nog in de kinderschoenen. De meeste ODBMS-en bevinden zich nog in een expermimentele fase. Veel

ODBMS-produkten beloven veel, maar of ze dat in de praktijk kunnen waarmaken moet nog blijken. Alhoewel er aan een standaardisatie van ODBMS-en gewerkt wordt , is er op dit moment nog geen standaard. Als een bedrijf een ODBMS aanschaft voor vele guldens en het blijkt dat deze te sterk afwijkt van de te ontwikkelen standaard, moet je als bedrijf overschakelen naar een nieuwe ODBMS. Als we de voor- en nadelen nog eens opsommen, krijgen we het volgende overzicht:

- **Voordelen**
  - **ODBMS**
    - object model en ODBMS sluiten goed op elkaar aan
  - **RDBMS**
    - vaak al aanwezig in een bedrijf
    - geen Forward Gateway nodig
    - standaard
    - is een bewezen concept
- **Nadelen**
  - **ODBMS**
    - geen standaard
    - veel verschillende te koop
    - onduidelijk wat ze echt doen en in hoeverre ze OO zijn
    - bevindt zich in experimentele fase
  - **RDBMS**
    - impedantie probleem

We zien aan de hand van de voor- en nadelen dat het gebruik van een ODBMS, waar de eerste voorkeur naar uitgaat, op dit moment niet aan te raden is voor bedrijven. Vandaar dat in deze scriptie het geval *van RBDMS naar RDBMS* verder onderzocht wordt. Hiertoe wordt in hoofdstuk 5 het relationele model beschreven. In hoofdstuk 6 wordt de invloed op de in hoofdstuk 3 voorgestelde architectuur en de methode onderzocht. Vervolgens wordt de methode voor het *van RDBMS naar RDBMS* geval verder uitgediept.

Hoofdstuk 5

Beschrijving van een RDBMS met SQL

Het *relationeel data model* is uitgevonden door Codd[Cod90] en is gebaseerd op een simpel concept - de tabel. Een *relationeel* DBMS (RDBMS) is een computer programma dat deze tabellen aan kan. Een RDBMS, zoals Codd definieerde, heeft drie belangrijke delen:

- De data, gepresenteerd in tabellen
- Operatoren, om de tabellen te kunnen manipuleren
- Integriteitsregels

Elk van deze delen worden toegelicht in de volgende paragrafen.

5.1 RDBMS Logische Data Structure

Een relationele database is logisch gezien een collectie van tabellen:

$$RDB = \{< tb_1, ..., tb_l > | tb_i \in TB\}$$

Waarbij TB de verzameling van alle mogelijke tabellen is.

In Van der Weide[Wei88] wordt een tabel gedefinïeerd. Er wordt uitgegaan van een verzameling D waarvan we de elementen *domeinen* noemen.
Bijvoorbeeld:

$$D = \{int, real, char[20], ...\}$$

De verzameling D^* bevat alle waarden uit deze domeinen:

$$\mathcal{D}^* = \bigcup \mathcal{D} = \bigcup_{D \in \mathcal{D}} D$$

Daarnaast is er een verzameling A, waarvan we de elementen *attributen* noemen. Aan elke attribuut is een domein geassocieerd:

$$Dom : A \rightarrow D$$

Bijvoorbeeld: $A = \{studcode, studnaam, woonplaats, ...\}$, en

$$Dom(studcode) = char[6]$$

$Dom(studnaam) = char[40]$
$Dom(woonplaats) = char[32]$

Een *relationeel schema* is van het type tabel. R is een relationeel schema indien $R \subseteq A$.

In het *relationeel gegevens model* wordt (een deel van) de werkelijkheid beschreven in de vorm van een aantal typen van tabellen. Dit deel van de werkelijkheid wordt aangeduid als het *Universe of Discourse*, afgekort *UoD*. We noemen $\mathcal{C}$ een conceptueel schema indien:

1. elke $C \in \mathcal{C}$ een relationeel schema is en
2. $\bigcup \mathcal{C} = A$.

Als R een relationeel schema is, dan is een *tupel* t over schema R een afbeelding:

$t : R \rightarrow D^*$

zodanig dat in alle kolommen een waarde van het goede type staat:

$[t(A) \in Dom(A)]$

We noemen ρ een relatie over R indien ρ een (eventueel lege) verzameling van tupels over R is.

$t =< studcode : Jans32, vakcode : A1, cijfer : 9 >$

$$\rho = \{ \begin{array}{l} < studcode : Jans32, vakcode : A1, cijfer : 9 >, \\ < studcode : Jans32, vakcode : M1, cijfer : 7 >, \\ < studcode : Klaa29, vakcode : A1, cijfer : 8 > \end{array} \}$$

Het relationele schema van een relatie r wordt aangegeven met $Attr(r)$.

5.2 RDBMS Operatoren

SQL heeft verschillende soorten zinnen, die verdeeld worden in drie taal groepen:

1. Data Definition Language (DDL): Deze zinnen worden gebruikt om database structuren, integriteitsregels en de privileges van gebruikers te definiëren.
2. Data Manipulation Language (DML) : DML zinnen worden gebruikt om data bij te werken of om data te selecteren.
3. Module Language: Module Language zinnen zijn DML zinnen die binnen in een programma aangeroepen kunnen worden.

In de volgende paragrafen zullen alleen de voor deze scriptie belangrijke zinnen behandeld worden. Voor de volledigheid wordt de rest alleen opgesomd. Voor meer gedetailleerde informatie over SQL kan men Van der Lans[vdL89] en Abcouwer en Geesink[AG89] raadplegen. De formele syntax beschrijving van de SQL zinnen is overgenomen uit Van der Lans[vdL89]. De formele syntax wordt beschreven in de *Backus Naur Form (BNF)*. BNF beschrijft een taal met behulp van substitutie regels of productie regels bestaande uit een aantal symbolen. In elke productie regel wordt één symbool gedefinieerd. Een symbool kan een SQL statement, een tabel naam of een puntkomma zijn. De BNF kent een aantal metasymbolen:

- <> Non-terminal symbolen worden binnen de < en > gedefinieerd.
- **::=** Dit symbool wordt gebruikt om het non-terminal symbool (links van het ::= teken) te scheiden van de defninitie (rechts van het ::= teken). De ::= moet gelezen worden als *wordt gedefinieerd als*.

- | Alternatieven worden met behulp van de | aangegeven.
- [ ] Alles tussen de rechte haken is optioneel.
- **...** De drie puntjes tonen dat iets één of meerdere keren herhaald mag worden.
- { } Alle symbolen tussen { en } vormen een groep.

5.2.1 DDL Zinnen

De SQL standaard kent de volgende DDL zinnen:

- *CREATE SCHEMA*
- *CREATE TABLE*
- *CREATE VIEW*
- *GRANT*

CREATE SCHEMA

In de SQL standaard hoort bij elke *tabel* of *view* een *schema*. Op het moment dat een schema gemaakt wordt, worden ook alle tabellen, views en privileges gemaakt. Ook wordt de naam van de gebruiker gespecificeerd. Deze is de eigenaar van het schema en alle tabellen en views die erbij horen. Elke gebruiker mag maar één schema bezitten. Een database mag meerdere schema's bevatten.

```
<schema> ::=
   CREATE SCHEMA AUTHORIZATON <authorization identifier>
   [<schema element>...]

<schema element> ::=
   <table definition> |
   <view definition> |
   <privilege definition>
```

CREATE TABLE

Een tabel is de enige plaats in SQL waarin data opgeslagen kan worden.

```
<table definition> ::=
   CREATE TABLE <table name>
   (<table name> [{,<table element>}...])

<table name> ::=
   [<authorization identifier>.]<table identifier>

<table element> ::=
   <column definition> |
   <unique constraint definition>

<column definition> ::=
   <column name> <data type> [NOT NULL [UNIQUE]]

<unique constraint definition> ::=
   UNIQUE <column list>
```

```
<column list> ::=
   (<column name> [{,<column name>}...])

<data type> ::=
   CHARACTER [(<length>)} |
   CHAR      [(<length>)} |
   NUMERIC   [(<precision> [,<scale>])] |
   DECIMAL   [(<precision> [,<scale>])] |
   DEC       [(<precision> [,<scale>])] |
   INTEGER   |
   INT       |
   SMALLINT  |
   FLOAT     [(<precision>)] |
   REAL      |
   DOUBLE PRECISION
```

Tabellen of views mogen niet dezelfde naam hebben. Ook de kolommen van een tabel mogen niet dezelfde namen hebben. Elke kolom die als UNIQUE wordt gespecificeerd, moet ook als NOT NULL gespecificeerd worden.

NOT NULL is een integriteitsregel waarmee aangegeven kan worden dat een bepaalde kolom geen NULL waarden mag hebben.

UNIQUE is een integriteitsregel waarmee aangegeven wordt dat twee gelijke waarden niet in een bepaalde kolom of combinatie van kolommen mag voorkomen.

De integriteitsregels NOT NULL en UNIQUE worden na elke (relevante) SQL zin getest. Als een UPDATE of INSERT zin de integriteitsregels overtreden, accepteert SQL deze zin niet en wordt een negatieve waarde toegekend aan de SQL-CODE variable.

CREATE VIEW

Een view is een virtuele tabel waarvan de virtuele inhoud afgeleid is van één of meer andere tabellen of views. Een SELECT wordt gebruikt om de inhoud te definiëren. Een view is een *virtuele tabel.*

```
<view definition> ::=
   CREATE VIEW <table name>
   [ <column list> ]
   AS <query specification>
   [WITH CHECK OPTION]

<table name> ::=
   [<authorization identifier>.]<table identifier>

<column list> ::=
   (<column name> [{,<column name>}...])

<query specification> ::=
   SELECT [ ALL | DISTINCT ] <select list>
   <table expression>

<select list> ::=
   <value expression> [ {,<value expression>}... ] | *

<table expression> ::=
   <from clause>
   [ <where clause> ]
```

```
   [ <group by clause> ]
   [ <having clause> ]
```

De naam van een view mag niet gelijk zijn aan de naam van een view of een andere tabel in dat schema.

Als een view met de WITH CHECK OPTION wordt gedefinieerd, worden alle updates van een UPDATE en INSERT zin getest.

5.2.2 DML Zinnen

Voor het manipuleren van data kent SQL twee soorten DML zinnen: *cursor afhankelijk* en *cursor onafhankelijk*. De cursor afhankelijke zinnen zijn:

- *DECLARE*
- *OPEN*
- *FETCH*
- *CLOSE*
- *DELETE*
- *UPDATE*

Een cursor is een mechanisme waarmee het resultaat van een zoek conditie van rij naar rij gelezen kan worden.

De cursor onafhankelijke zinnen zijn:

- *SELECT*
- *INSERT*
- *UPDATE*
- *DELETE*
- *COMMIT*
- *ROLLBACK*

Deze zinnen werken direct op de inhoud van de tabellen zonder tussenkomst van een cursor.

SELECT

Een SELECT wordt gebruikt om één of meer rijen van een tabel te selecteren.

```
<select statement> ::=
   SELECT [ ALL | DISTINCT ] <select list>
   INTO <target specification>
        [ {,<target specification>}... ]
   <table expression>

<select list> ::=
   <value expression> [ {,<value expression>}... ] | *

<target specification> ::=
   <parameter specification> |
```

```
   <variable specification>

<parameter specification> ::=
   <paramter name> [[INDICATOR] <indicator parameter> ]

<variable specification> ::=
   <variable name> [[INDICATOR] <indicator variable> ]

<table expression> ::=
   <from clause>
   [ <where clause> ]
   [ <group by clause> ]
   [ <having clause> ]
```

INSERT

De *INSERT* kent twee vormen. De eerste voegt één rij van nieuwe waarden aan de tabel toe. De tweede voegt een set van rijen toe aan de tabel.

```
<insert statement> ::=
   INSERT INTO <table name>
   [<column list>]
   { VALUES ( <value> {,<value>}... ) | <query specification> }

<table name> ::=
   [ <authorization identifier>. ] <table identifier>

<column list> ::=
   ( <column name> [{,<column name>}...] )

<value> ::=
   <value specification> | NULL

<query specification> ::=
   SELECT [ALL | DISTINCT] <select list>
   <table expression>

<select list> ::=
   <value expression> [{,<value expression>}...] | *

<table expression> ::=
   <from clause>
   [ <where clause> ]
   [ <group by clause> ]
   [ <having clause> ]
```

5.2.3 Module Language Zinnen

De SQL standaard kent de volgende Module Language zinnen:

- *MODULE*
- *PROCEDURE*

DML zinnen kunnen op twee manieren gebruikt worden:

- In een module: Een SQL Module is een collectie van SQL procedures. Elke procedure bevat één DML statement. Door een programma, geschreven in een andere taal, te linken aan een module, kunnen DML zinnen die in procedures gedefinieerd worden, aangeroepen worden vanuit het programma.
- In een programma geschreven in een andere taal (bekend als *embedded SQL*).

5.3 RDBMS Integriteit

Twee aspecten van integriteit in Codd's[Cod90] model zijn *entiteit integriteit* en *referentiële integriteit*.

Entiteit integriteit zegt dat elke tabel precies één *primary key* mag hebben. Een *primary key* is een combinatie van één of meer attribuut waarden, waarmee elke rij in een tabel gelokaliseerd kan worden.

Referentiële integriteit eist dat de RDBMS de *foreign key* consistent houdt met zijn corresponderende primary key. Een *foreign key* is een primary key van een tabel die ingebed is in een andere (of dezelfde) tabel.

Hoofdstuk 6

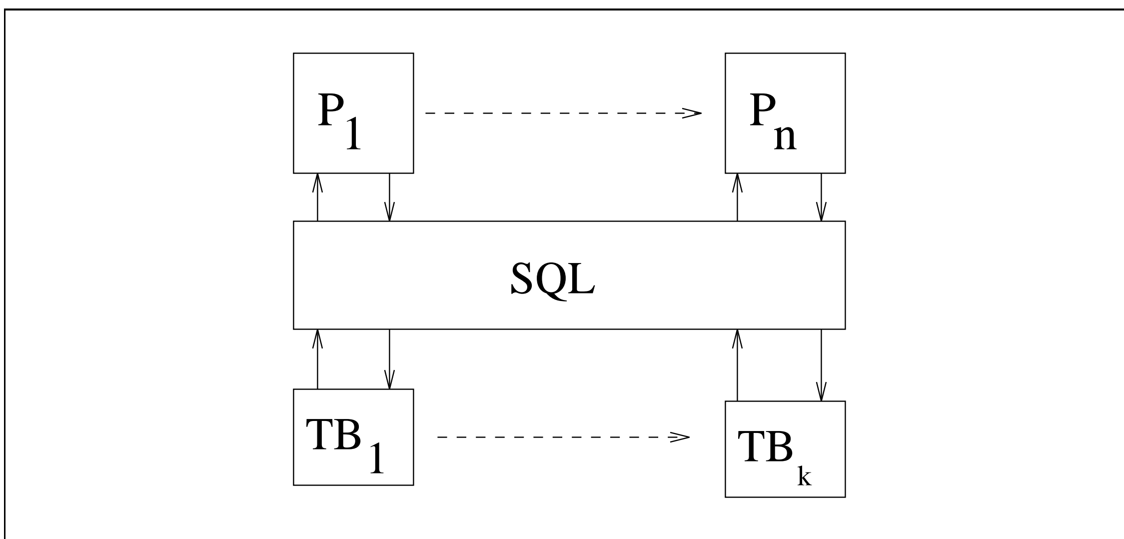
Een Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd

In hoofdstuk 4 is een verdere beperking gedaan op het toepassingsgebied. Aan de hand van een duidelijke motivatie is gekozen om het *RDBMS naar RDBMS* geval verder te onderzoeken. Hiertoe is in hoofdstuk 5 het relationeel data model beschreven. In dit hoofdstuk wordt de invloed van deze verdere beperking op de architectuur en de methode bekeken.

6.1 De Architectuur

De keuze van SQL als OS/DBMS-laag heeft een aantal gevolgen voor de architectuur beschreven in Hoofdstuk 3. In deze sectie wordt de architectuur beschreven zoals die er met de invulling van SQL uitziet.

We zien elk systeem opgebouwd als in Figuur 6.1.



Figuur 6.1: Elk systeem is als volgt opgebouwd.

We hebben dus:

$Programma's : P_1....P_n$ (Er bestaat geen afhankelijkheid tussen de P's)

De verzameling van Programma's P_i noemen we P:

$P = \{P_1...P_n\}$

$Relationele\ tabellen : tb_1....tb_k$

De verzameling van de tabellen tb_i noemen we de relationele database (RDB):

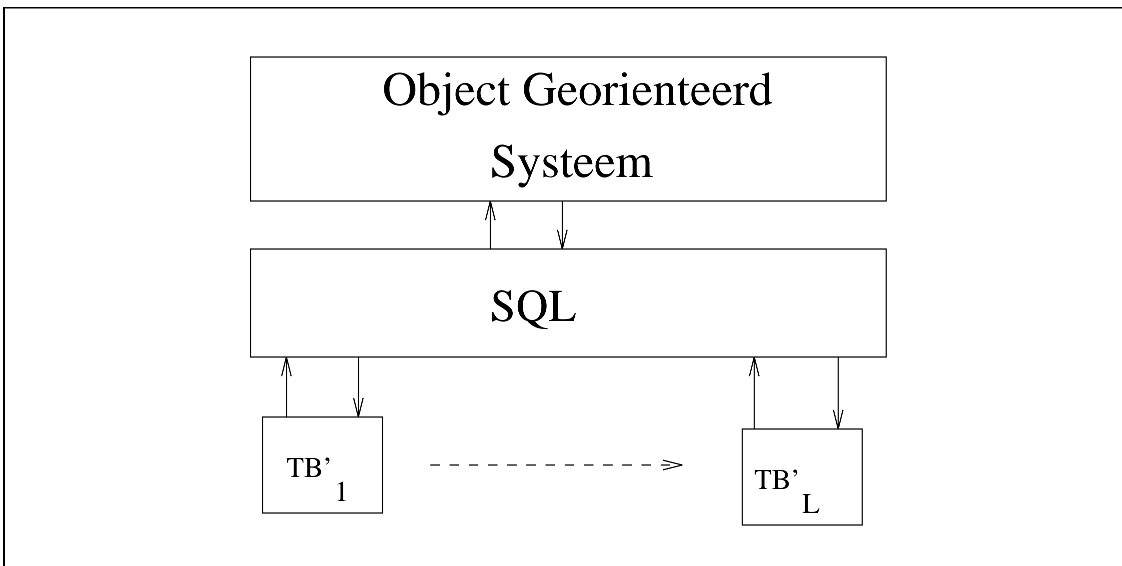
$RDB = \{< tb_1, ..., tb_l > | tb_i \in TB\}$

Waarbij TB de verzameling van alle mogelijke tabellen is.

Uiteindelijk willen we van deze architectuur omschakelen naar een architectuur waarbij $P'_1...P'_o$ object georiënteerd zijn. Doordat er een object model gemaakt wordt, moet er een vertaling gemaakt worden van het object model naar een relationele database model RDB', die de attributen van de objecten opslaat. Daar we onder andere de gegevens van RDB willen hergebruiken, moeten we een mapping maken tussen RDB en RDB':

$mapping : RDB \rightarrow RDB'$

In Figuur 6.2 zien we de architectuur die we uiteindelijk willen realiseren.



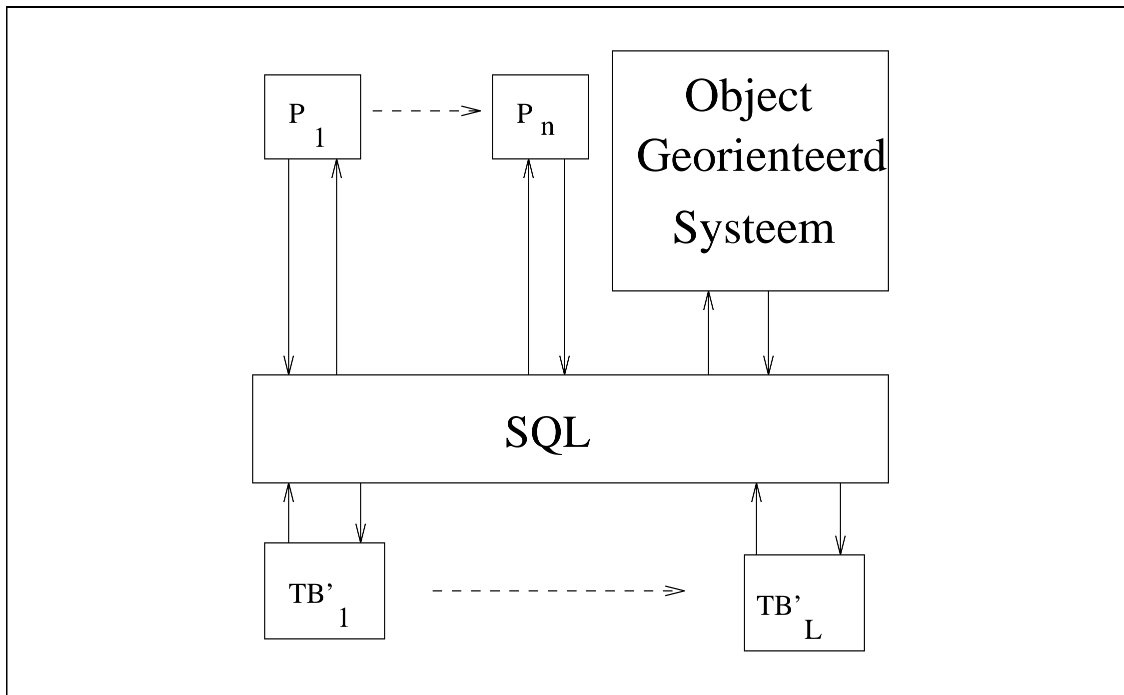
Figuur 6.2: De Uiteindelijke Architectuur; deze situatie willen we in de toekomst realiseren. Alles is dan object georiënteerd.

Daar we niet alleen de oude DataBase RDB willen hergebruiken, maar ook de programma's $P_1...P_n$ moet de oude RDB virtueel geïmplementeerd worden met behulp van het *view* mechanisme van SQL. Hierdoor blijven de SQL zinnen in de programma's $P_1...P_n$ geldig en kunnen deze dus hergebruikt worden. Als tussen-fase ontstaat dan het plaatje van Figuur 6.3.

We kunnen nu geleidelijk elke P_i herimplementeren.

Aan de hand van de architectuur kunnen we een aantal stellingen formuleren:

1. RDB is hergebruikt.



Figuur 6.3: De Tussen Fase Architectuur; als tussen fase willen we de oude programma's P_i naast de nieuwe situatie laten bestaan.

2. Programma's $P_1 ... P_n$ kunnen hergebruikt worden door de oude tabellen met behulp van het view mechanisme virtueel te definiëren uit de nieuwe tabellen.
3. Elke P_i kan naar OO gemigreerd worden.
4. Er is een situatie waarbij alle $P_1 ... P_n$ gemigreerd zijn (Dit volgt automatisch uit bewering 3).

6.2 De Methode

Door het gebruik van SQL als OS/DBMS-laag vervalt de *Forward Gateway* en hiermee dus ook de stap *Maak Forward Gateway*. In plaats van de Forward Gateway gebruiken we het *view* mechanisme van SQL. Extra stap wordt dus het definiëren van de views. Deze stap wordt opgenomen als substapje van *Maak Nieuwe en Virtuele Oude DataBase*.

De *SQL Interface Approach Method* omvat de volgende stappen:

1. Analyseer Huidige Situatie. In deze stap worden de code en de data van het oude systeem geanalyseerd.
2. Maak Object Model Toekomstige Situatie. De toekomst wordt gemodelleerd in een Object Model.
3. Omzetten Object Model naar Relationeel Model .
4. Maak Nieuwe en Virtuele Oude DataBase. Maak een **mapping table** die een vertaling maakt van oude database service calls en de nieuwe service calls, alsook van oude database items naar de nieuwe attribuut waarden. Vervolgens worden de nieuwe tabellen gedefinïeerd gevuld en wordt de oude situatie virtueel hersteld.
5. Migreer Oude Software. Delen oude software kunnen gemigreerd worden naar de nieuwe omgeving.

6.3 Voor- en Nadelen van de Methode

Doordat SQL als uitgangspunt wordt genomen, vervalt het nadeel van de constructie van een Forward Gateway.

Voor de volledigheid worden alle voor- en nadelen opgesomd:

- **Voordelen:**
  - *Veiligheid.* Omdat de oude software niet verandert, blijven de oude functies gelijk.
  - *Weinig inspanningen in oude code.* De oude software hoeft nauwelijks geanalyseerd te worden.
  - *Kort migratie pad.* Het is niet nodig om de oude software nauwkeurig te analyseren en te begrijpen.
  - *Er is een stapsgewijze migratie mogelijk van het oude systeem naar een object georiënteerde omgeving.*
  - *Er ontstaat een flexibel systeem.* Doordat er stap voor stap gemigreerd wordt naar een object georiënteerde omgeving.
- **Nadelen:**
  - Door het gebruik van SQL ontstaat er het *impedantie probleem.*
  - Alle architecturen die geen relationeel databasemanagement systeem met SQL als OS/DBMS-laag hebben, worden niet omvat door deze beperkte methode.

Hoofdstuk 7

Stap 1: Analyseer Huidige Situatie

Het doel van deze stap is het verkrijgen van inzicht in de huidige situatie. Hiervoor worden alle code en data bekeken.

De code hoeft niet in detail onderzocht te worden, wat een voordeel is ten opzichte van andere *reverse engineering* methoden. Alleen de code die toegang verschaft tot de data moet bekeken worden.

De data structuur wordt vastgelegd in schema's om zo een duidelijk beeld van de structuur te verkrijgen.

Daar de methode uitgaat van de architectuur zoals die afgebeeld is in Figuur 3.2, is het belangrijk dat er een afspiegeling plaatsvindt van de huidige situatie op die architectuur. Vandaar dat deze stap onder te verdelen is in sub-stapjes:

1. Spoor Programma's P_i op
2. Leg Tabel Structuren Vast
3. Leg Afhankelijkheid Vast tussen Programma's P_i en de Tabellen TB_i

7.1 Spoor Programma's P_i op

De verschillende programma's P_i moeten opgespoord en beschreven worden. In hoofdstuk 3.1 is er vanuit gegaan dat er geen afhankelijkheden tussen de verschillende programma's P_i bestaan. De afhankelijkheden binnen de programma's zelf worden later bekeken.

7.2 Leg Tabel Structuren Vast

Het doel van deze stap is het vastleggen van de tabel structuren in de huidige situatie. Vul voor elke file F_i Figuur 7.1 in:

Het resultaat van deze stap is de verzameling van de schema's.

Veld	Veld Naam	Type	Grootte	Omschrijving

Figuur 7.1: Tabel om gegevensstructuren vast te leggen.

7.3 Leg Afhankelijkheid Vast tussen Programma's P_i en Tabellen TB_i

In de voorafgaande stappen hebben we de afzonderlijke delen van de architectuur benoemd. Nu is het noodzaak om de afhankelijkheid tussen de afzonderlijke delen vast te leggen. In Figuur 3.1 is de afhankelijkheid tussen de drie verschillende lagen duidelijk te zien. Programma's P_i hebben toegang tot de files F_i via de DBMS/OS laag.

In Figuur 7.2 kunnen we de afhankelijkheid vastleggen tussen de programma's P_i en de files F_i.

File maakt gebruik van	P_1	P_2	- - - - - - - - ->	P_k
TB_1	X	X		
TB_2				X
↓				
TB_L	X			

Figuur 7.2: Tabel om de afhankelijkheid tussen Programma's P_i en Tabellen TB_i vast te leggen.

Hoofdstuk 8

Stap 2: Maak Object Model Toekomstige Situatie

Deze stap analyseert de toekomstige situatie en stelt het object model hiervan op. De uitvoering van deze stap is te vinden in de meeste methoden en wordt daarom niet gedetailleerd beschreven in deze scriptie. Bij de keuze van een methode moet men in gedachten houden dat er nog geen methoden zijn, die zover uitgewerkt zijn, dat deze door iedereen op een eenduidige manier gebruikt, hetzelfde resultaat opleveren. Als leidraad nemen wij de analyse fase van Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91]. Ook is het mogelijk om aan de hand van de tabellen die uit stap 1 rollen een object model te construeren, zie Premerlani en Blaha [PB94].

8.1 Volledige Analyse

Het doel van de analyse is het ontwikkelen van een model dat aangeeft wat het systeem moet doen. We gaan uit van een gedetailleerde beschrijving van het probleem. Door de analyse van de probleemomschrijving komen we aan de hand van een aantal stappen tot een object model, zie Rumbaugh et al.[RBP+91].

- Spoor object klassen op.
- Maak een datadictionary met beschrijvingen voor de klassen, attributen en associaties.
- Voeg attributen toe voor objecten en links.
- Reorganiseer en simplicificeer de object klassen met behulp van inheritance.
- Test acces paden door middel van scenario's en herhaal indien nodig de bovenstaande stappen.
- Groepeer klassen in modules, gebaseerd op koppeling en gerelateerde functies.

Andere veel gebruikte methoden zijn die van: Jacobson, Christerson, Jonsson en Van Overgaard[JCJvO92], Kristen[Kri93], Rumbaugh et al.[RBP+91] en Meyer[Mey88].

8.2 Van Relationeel Model naar Object Model

In het artikel van Premerlani en Blaha[PB94] wordt er onderzoek gedaan naar het hergebruik van Relationele Databases.

8.2.1 De Methode

Premerlani en Blaha[PB94] hebben een aantal stappen ontwikkeld, die een relationeel model omvormen tot een object model.

- Maak een initieel object model.
- Zoek de candidate keys.
- Zoek de foreign keys.
- Verfijn de klassen.
- Spoor de generalisaties op.
- Spoor de associaties op.
- Maak de transformatie.

8.2.2 De Voordelen

Het voordeel van de methode van Premerlani en Blaha[PB94] is dat er in relatief korte tijd een object model geconstrueerd kan worden.

8.2.3 De Nadelen

Een nadeel van het gebruik van deze methode is dat het onmogelijk is om eventuele gebreken van de oude applicatie te corrigeren omdat het de oude gegevensstructuren als basis neemt.

Een ander probleem is dat de ontwerpers grote kennis en ervaring moeten hebben van de object georiënteerde methode.

De methode wordt alleen aangeraden als er omgeschakeld moet worden van een oude omgeving naar een nieuwe omgeving, omdat dan de oude database een goede leidraad is voor de gegevens die opgeslagen moeten worden.

Hoofdstuk 9

Stap 3: Omzetten Object Model naar Relationeel Model

In *Stap 2: Maak Object Model Toekomstige Situatie* is een object model gemaakt dat de toekomstige situatie beschrijft. Het is nu nodig om het object model om te zetten naar een relationeel model.

9.1 Twee Mogelijke Implementaties van het Object Model in SQL

Er zijn twee mogelijke manieren om het object model in SQL tabellen op te slaan:

- One Table Approach (zie Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91])
- More Table Approach (zie Rumbaugh et al.[RBP+91])

In de volgende paragrafen zullen beide mogelijkheden uitgewerkt worden.

9.1.1 One Table Approach

Gebruik een RDBMS en bewaar het gehele object model als de tuple <entity-naam, key, attribuut-naam, waarde>. Het is dan mogelijk om alle klassen, associaties en generalisaties van het hele object model in één tabel te bewaren.

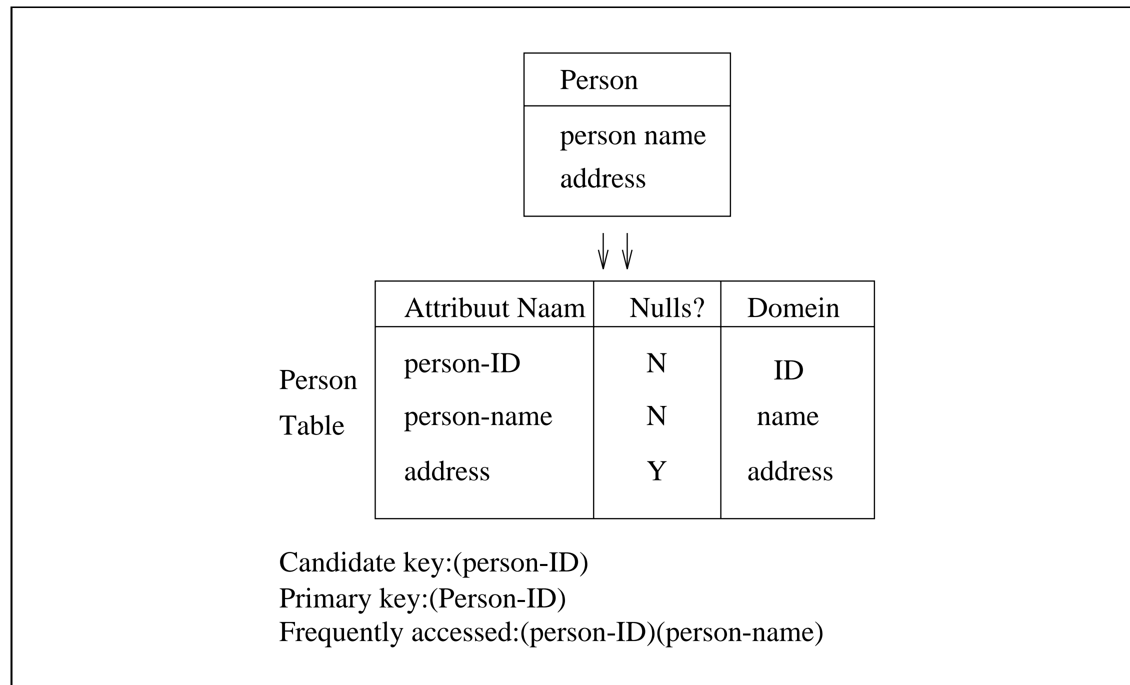
In het algemeen is het niet ideaal om alle entiteiten in één tabel te stoppen. Een database is meer dan alleen een opslag voor data, deze moet ook zelf-beschrijvend zijn. Een database slaat niet alleen data op, maar ook de structuur van de data (meta-data). De *One Table Approach* verwijdert metadata uit de database. Deze metadate moet dan in de code gebracht worden die met die ene lange tabel moet handelen.

9.1.2 More Table Approach

Een object model bevat vele klassen, associaties, generalisaties en attributen. Om een object model te vertalen in ideale tabellen, moet er gekozen worden tussen verschillende mapping alternatieven. Er zijn bijvoorbeeld twee manieren om associaties om te zetten in tabellen en vier manieren om generalisaties om te zetten in tabellen. De rest van deze paragraaf laat zien hoe object modellen vertaald kunnen worden naar relationele tabellen. In de tabellen worden alleen de attribuut waarden van de objecten opgeslagen.

9.1.2.1 Van Klassen naar Tabellen

Elke klasse wordt in één of meerdere tabellen weergegeven (zie Figuur 9.1). De objecten in een klasse kunnen horizontaal en/of verticaal gepartitioneerd zijn. Als een klasse veel instanties heeft die maar weinig gebruikt worden, kan horizontale partitionering de efficiëntie verhogen doordat veel geraadpleegde objecten in één tabel en de andere objecten in een andere tabel kunnen staan. Natuurlijk moet de applicatie weten in welke tabel er gezocht moet worden. Soortgelijk kan de efficiëntie verhoogd worden met verticale partitionering in het geval dat een object attributen heeft met verschillende toegangs-patronen.



Figuur 9.1: Elke klasse wordt een tabel.

9.1.2.2 Van Binaire Associaties naar Tabellen

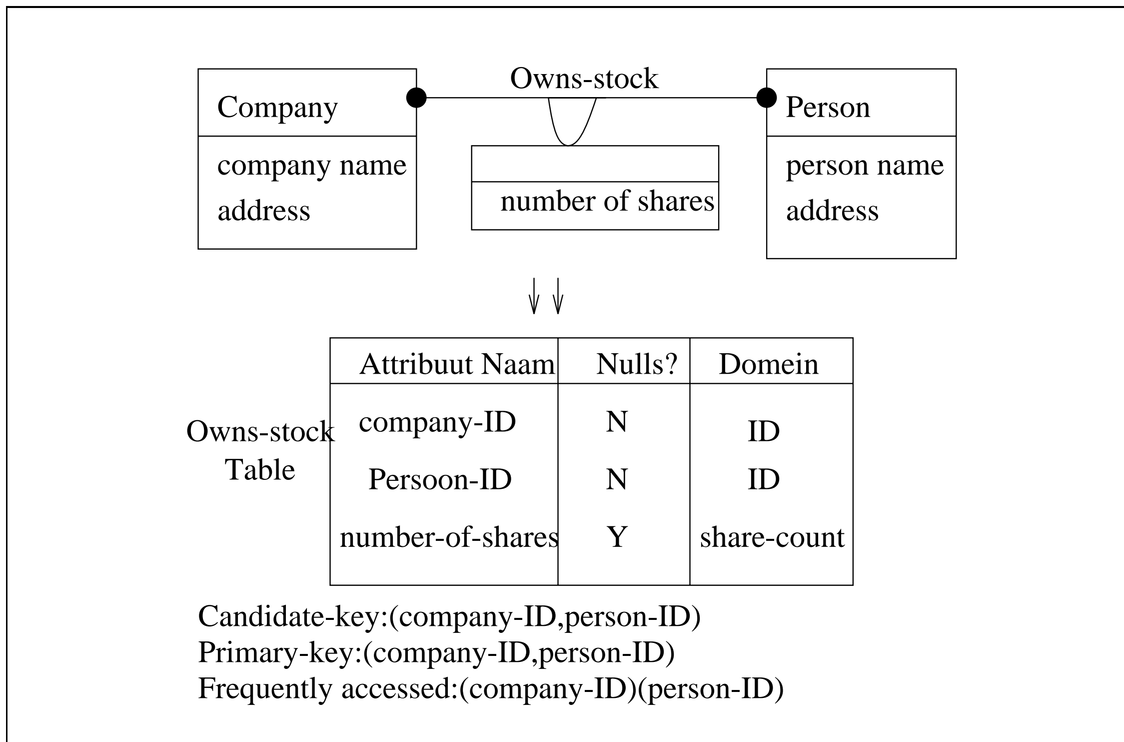
Vaak worden binaire associaties als klassen bekeken en dus volgens de vorige subsectie gerepresenteerd. Het voordeel hiervan is:

- Er zijn minder tabellen nodig.
- Snellere navigatie omdat er minder tabellen zijn.

De nadelen hiervan zijn:

- Associaties bestaan tussen onafhankelijke objecten. Het is niet netjes om objecten kennis te geven van andere objecten; *encapsulatie* wordt doorbroken. Juist *encapsulatie* is één van de voordelen van object georiënteerde systemen.
- Uitbreidbaarheid van een object model vermindert. Bij verandering van de structuur van een associatie werkt dit door tot in de data-laag.
- De representatie van associaties in tabellen bemoeilijkt het zoeken en updaten.

De uiteindelijke keuze om een associatie als een klasse te bekijken, hangt af van de vereiste snelheid van de navigatie. Figuur 9.2 geeft weer hoe je een binaire associatie als een klasse ziet.



Figuur 9.2: Een binaire associatie als een klasse.

De primary-keys van de gerelateerde klassen en alle link attributen worden attributen van de associatie tabel. Een associatie tabel zet altijd alle foreign-keys van de gerelateerde objecten op *NOT NULL*; de definitie van een associatie vereist dit. Een link tussen twee objecten vereist dat beide objecten bekend zijn.

Figuur 9.3 geeft aan hoe een one-to-many associatie omgezet wordt in tabellen.

Het is ook mogelijk om een one-to-one associatie in één tabel te zetten. De associatie *land heeft een hoofdstad* en zijn gerelateerde objecten kunnen in één tabel gezet worden. Het in één tabel vastleggen van zo'n associatie verhoogt de performance en vermindert het benodigd aantal geheugen-bytes ten koste van een verminderde uitbreidbaarheid. Figuur 9.4 geeft deze mogelijkheid weer in de trend van de voorgaande voorbeelden.

9.1.2.3 Van Ternaire Associaties naar Tabellen

Figuur 9.5 toont een RDBMS implementatie van een ternaire associatie.

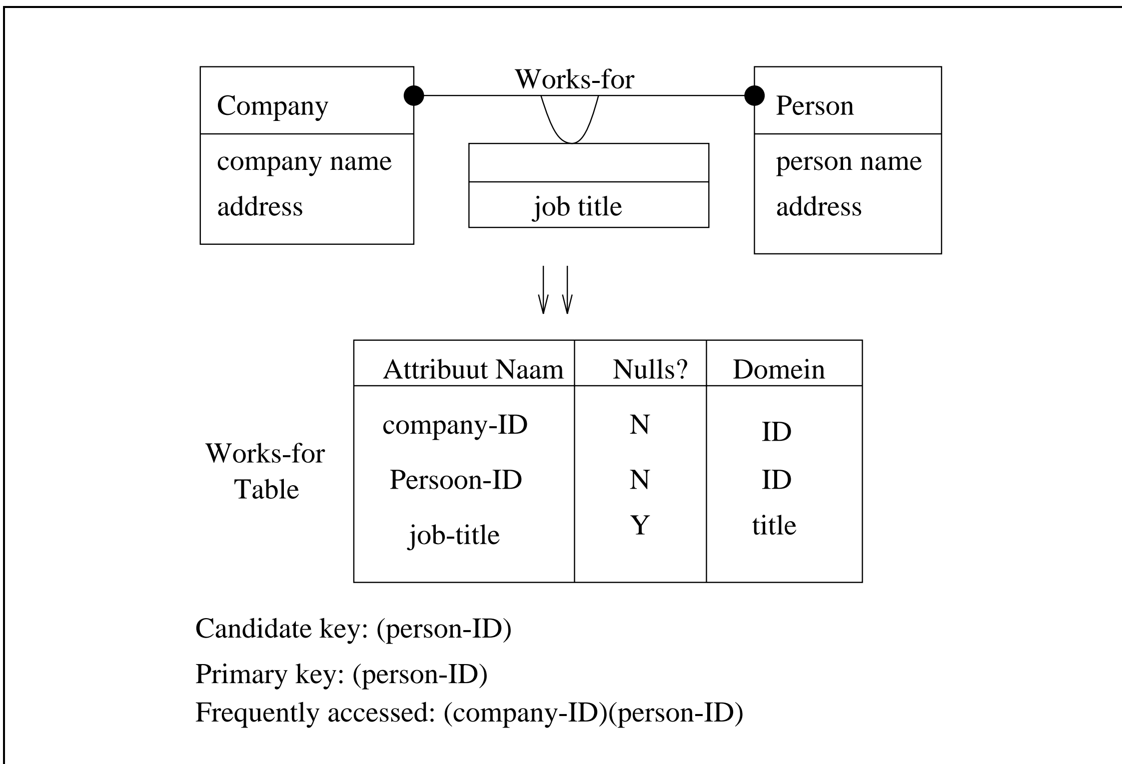
Er is een tabel voor elke klasse die participeert in de ternaire associatie. De link attributen mogen eventueel *NULL* zijn.

9.1.2.4 Van Qualified Associaties naar Tabellen

Bij een qualified associatie worden de primary-keys van de gerelateerde klassen en de *qualifier* in één tabel gerepresenteerd. Zowel de primary-keys van de gerelateerde klassen als de qualifier mogen niet *NULL* zijn. Figuur 9.6 geeft weer hoe een qualified associatie in een tabel weergegeven kan worden.

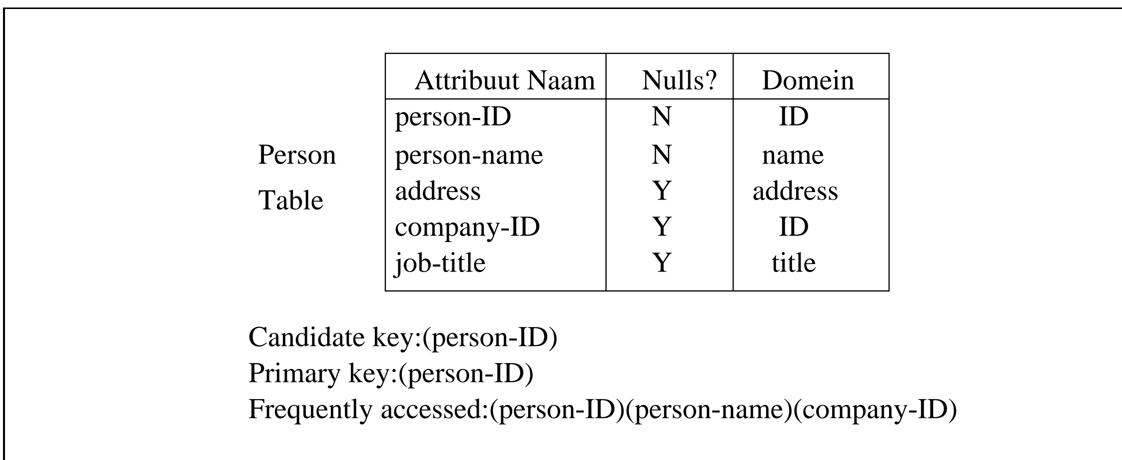
9.1.2.5 Van Generalisatie naar Tabellen

Er zijn vier mogelijkheden om generalisaties om te zetten naar tabellen. Figuur 9.7 zal als basis dienen om de vier strategiën te onderzoeken.



Attribuut Naam	Nulls?	Domein
company-ID	N	ID
Persoon-ID	N	ID
job-title	Y	title

Figuur 9.3: One-to-may associatie in een tabel.



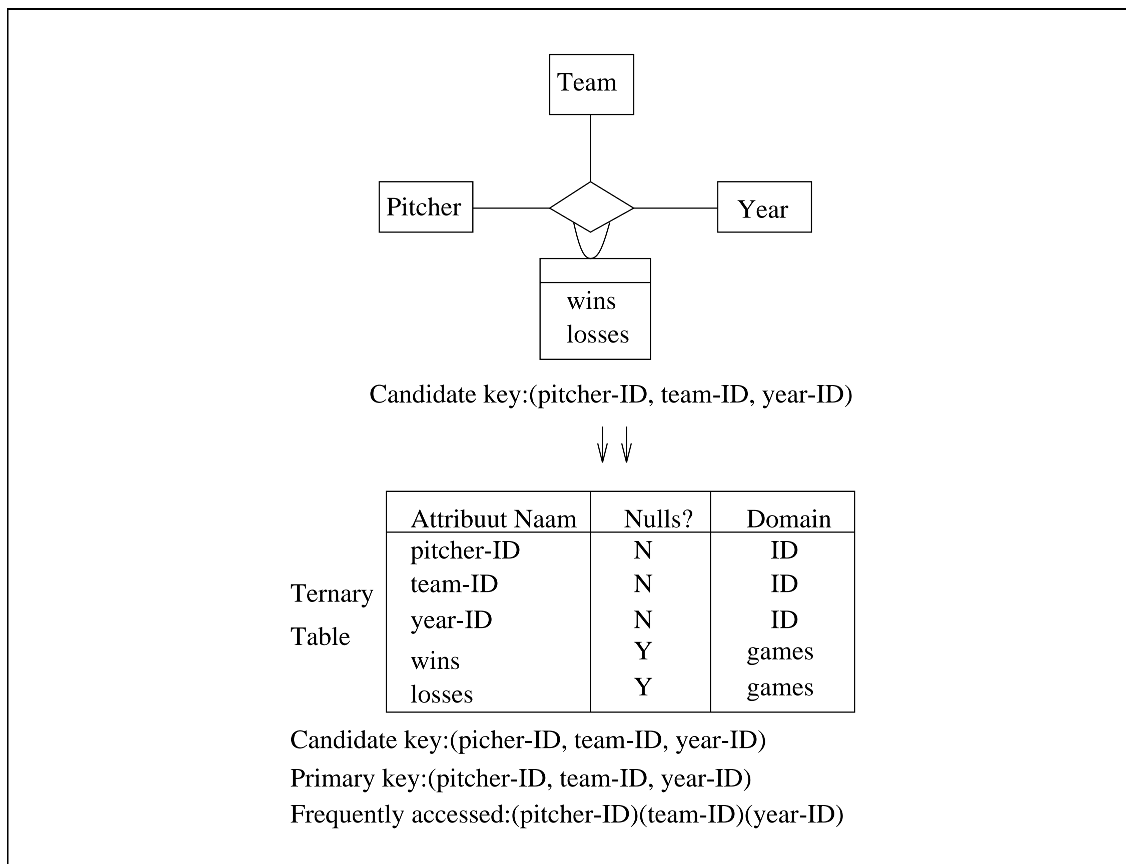
Attribuut Naam	Nulls?	Domein
person-ID	N	ID
person-name	N	name
address	Y	address
company-ID	Y	ID
job-title	Y	title

Figuur 9.4: One-to-one associatie in een tabel.

Bij de normale mapping worden de superklassen en de subklassen elk in een aparte tabel gerepresenteerd. De identiteit van een object wordt gegarandeerd door het gebruik van een gedeelde identifier. Figuur 9.8 geeft dit weer.

Deze manier is logisch gezien netjes en uitbreidbaarheid blijft mogelijk. Een nadeel is het grote aantal tabellen dat op deze manier nodig is om alle generalisaties vast te leggen. Hierdoor kan de navigatie traag zijn.

Een andere manier is de *many subclass approach* (zie Figuur 9.9). De motivatie om deze manier te kiezen, is het elimineren van de *superklasse naar subklasse navigatie* waardoor de snelheid wordt verhoogd. Deze manier elimineert de superklasse tabel en zet alle superklasse attributen in elke subklasse tabel. Deze manier zou gebruikt kunnen worden als de subklasse veel en de superklasse weinig attributen heeft. De applicatie moet dan natuurlijk wel weten in welke subklasse hij moet



Figuur 9.5: Ternaire associatie in een tabel.

zoeken.

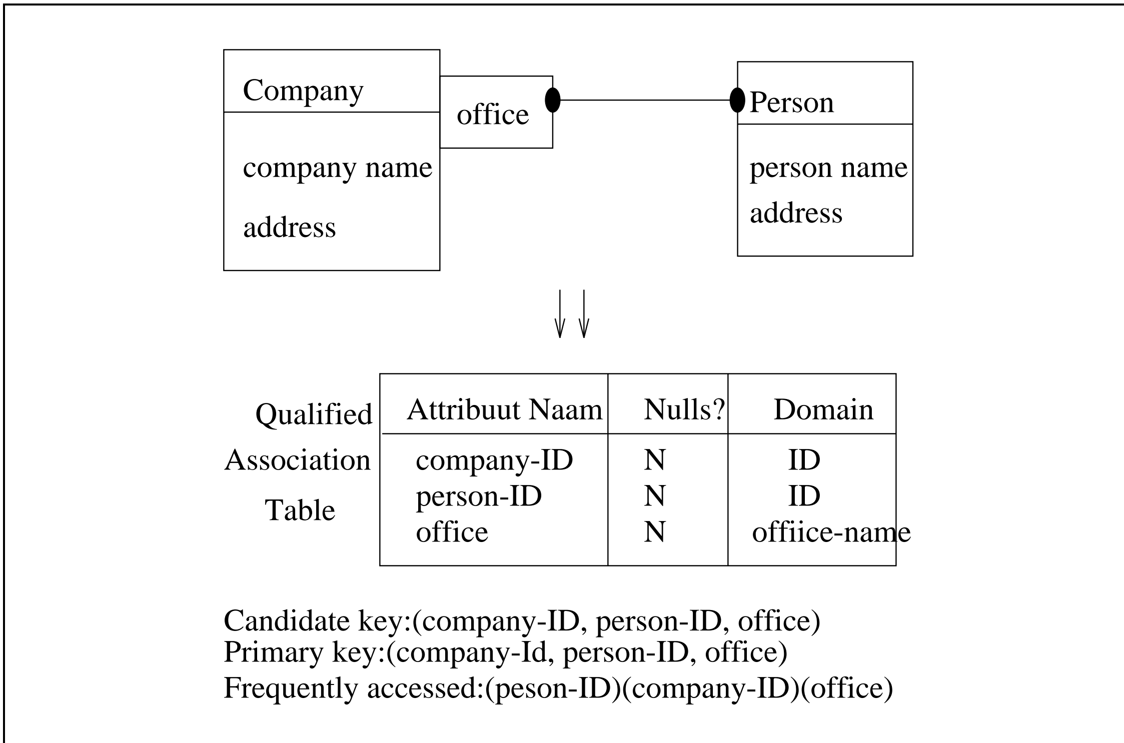
De derde mogelijkheid is de *one superclass table approach* (zie Figuur 9.10). Deze brengt alle subklasse attributen omhoog naar het superklasse nivo. Elk record gebruikt alleen de attributen van zijn subklasse, de andere attributen zijn *null*. Deze mogelijkheid kan bruikbaar zijn als er maar twee of drie subklassen zijn met weinig attributen.

De beste manier om generalisaties met disjoint multiple inheritance te representeren is de standaard manier zoals die aan het begin van deze sectie beschreven is. De beste manier om overlapping multiple inheritance te representeren is door één tabel per superklasse, één tabel per subklasse en één tabel voor de generalisatie te gebruiken.

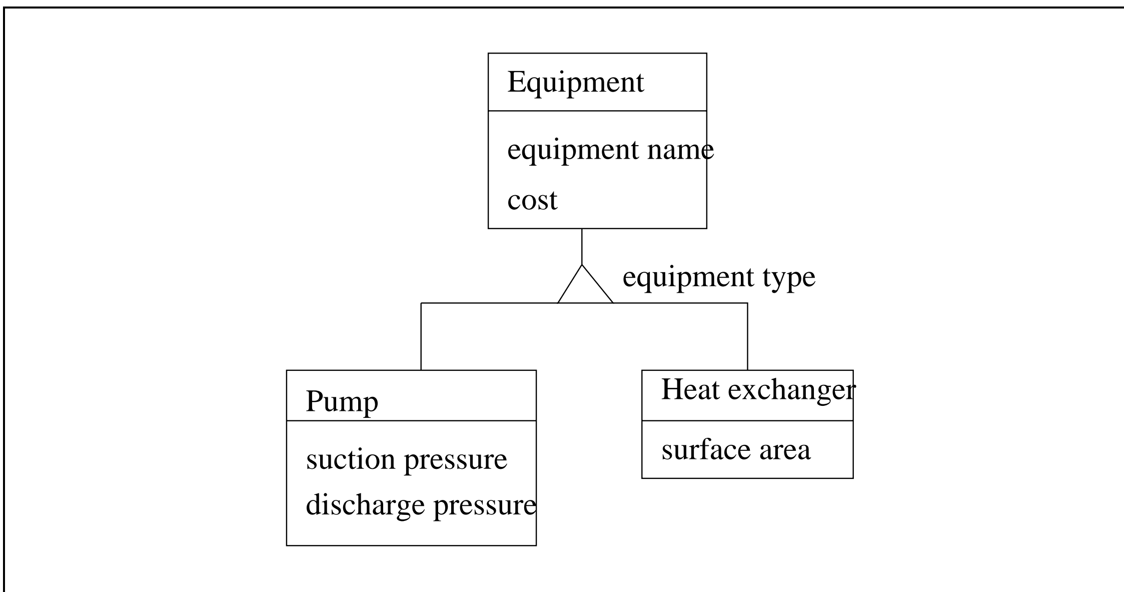
9.2 Voor- en Nadelen

Met de One Table Approach is een eenvoudige en snelle vertaling van het object model naar het relationele model mogelijk. Een nadeel, doordat er één lange tabel komt, is dat bij grote object modellen de snelheid waarmee informatie opgezocht kan worden aanzienlijk kleiner is dan bij de More Table Approach. Deze zoekt gerichter in de bijpassende kleinere tabellen en kan dus aanzienlijk sneller het gewenste resultaat opleveren. De vertaling van het object model naar een relationeel model via de More Table Approach is een ingewikkelder proces dat meer tijd kost dan bij de One Table Approach. De One Table Approach kent maar één tabel met een vaste structuur, wat als voordeel heeft dat bij een verandering van het object model de relationele structuur hetzelfde blijft. Verandering van het object model heeft bij een, via het More Table Approach verkregen, relationele structuur invloed op de tabelstructuren.

Een goede afweging van de keuze is noodzakelijk en afhankelijk van het probleemgebied.



Figuur 9.6: Qualified associatie in een tabel.



Figuur 9.7: Standaard voorbeeld generalisatie.

9.3 Maak Tabellen

In de voorafgaande paragrafen hebben we gezien welke mogelijkheden er zijn om van een object model naar tabellen te gaan. Het is nu tijd om de tabellen te creëren. Om straks een makkelijke mapping mogelijk te maken, vullen we elke tabel in in Figuur 9.11.

Equipment Table

Attribuut Naam	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
equipment-type	N	equip-type

Candidate key:(equipment-ID)(equipment-name)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)(equipment-name)

Pump table

Attribuut Naam	Nulls?	Domain
equipment-ID	N	ID
suction-pressure	Y	pressure
discharge-pressure	Y	pressure

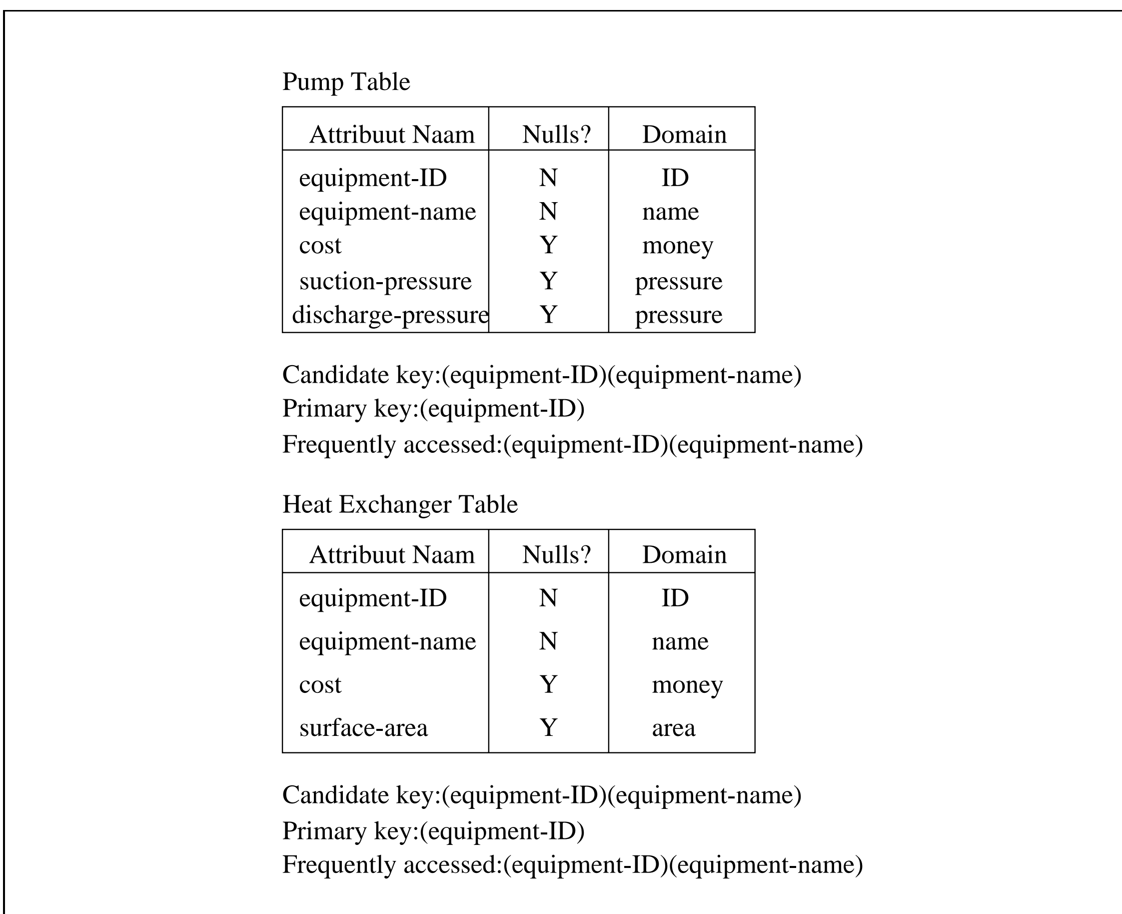
Candidate key:(equipment-ID)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)

Heat Exchanger Table

Attribuut Naam	Nulls?	Domain
equipment-ID	N	ID
surface-area	Y	area

Candidate key:(equipment-ID)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)

Figuur 9.8: Normale mapping van superklassen en subklassen elk in een aparte tabel.



Pump Table

Attribuut Naam	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
suction-pressure	Y	pressure
discharge-pressure	Y	pressure

Candidate key:(equipment-ID)(equipment-name)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)(equipment-name)

Heat Exchanger Table

Attribuut Naam	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
surface-area	Y	area

Candidate key:(equipment-ID)(equipment-name)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)(equipment-name)

Figuur 9.9: Many subclass approach.

Equipment Table

Attribute name	Nulls?	Domain
equipment-ID	N	ID
equipment-name	N	name
cost	Y	money
equipment-type	N	equip-type
suction-pressure	Y	pressure
discharge-pressure	Y	pressure
surface-area	Y	area

Candidate key:(equipment-ID)(equipment-name)
Primary key:(equipment-ID)
Frequently accessed:(equipment-ID)(equipment-name)

Figuur 9.10: One superclass table approach.

Veld	Veld Naam	Type	Grootte	Omschrijving

Figuur 9.11: Tabel om de nieuwe gegevensstructuren in weer te geven.

Hoofdstuk 10

Stap 4: Maak Nieuwe en Virtuele Oude Database

Het is nu noodzakelijk dat alle gegevens hergebruikt gaan worden. Dit gebeurt door het maken van een *mapping* tussen de oude tabelstructuren en de nieuwe tabelstructuren, die in Stap 1 en 3 verkregen zijn.

10.1 Een Inventarisatie

Om de mapping te maken moet er eerst een inventarisatie gemaakt worden van wat er tot nu toe is.

10.1.1 De Oude Tabellen

In *Stap 1: Analyseer Huidige Situatie* zijn de oude tabelstructuren in kaart gebracht.

We hebben een database DB, bestaande uit een aantal relationele schema's:

$DB = \{< tb_1, ..., tb_l | tb_i \in TB\}$

Er is een verzameling D van domeinen en een verzameling A van attributen. De functie Dom associeert aan elk attribuut een domein:

$Dom : A \rightarrow D$

10.1.2 De Nieuwe Tabellen

In *Stap 3: Kies Gewenst DataBase Management Systeem* zijn de nieuwe tabelstructuren gemaakt.

Er is een database DB' bestaande uit een aantal relationele schema's:

$DB' = \{< tb'_1, ..., tb'_k | tb'_i \in TB\}$

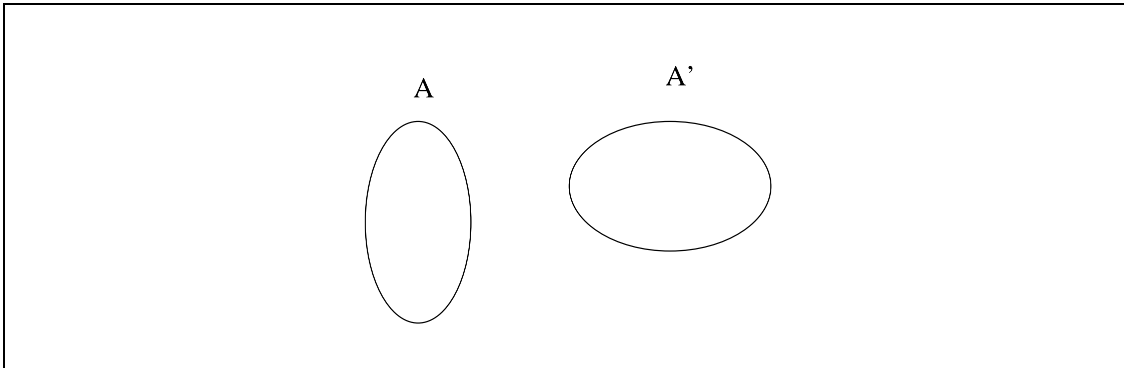
Er is een verzameling D' van domeinen en een verzameling A' van attributen. De functie Dom' associeert aan elk attribuut een domein:

$Dom' : A' \rightarrow D'$

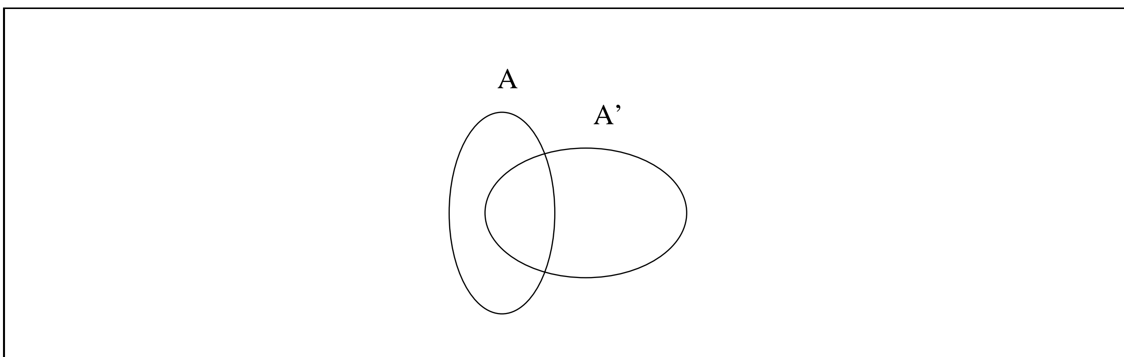
10.2 De Mapping

10.2.1 Attribuut Mapping

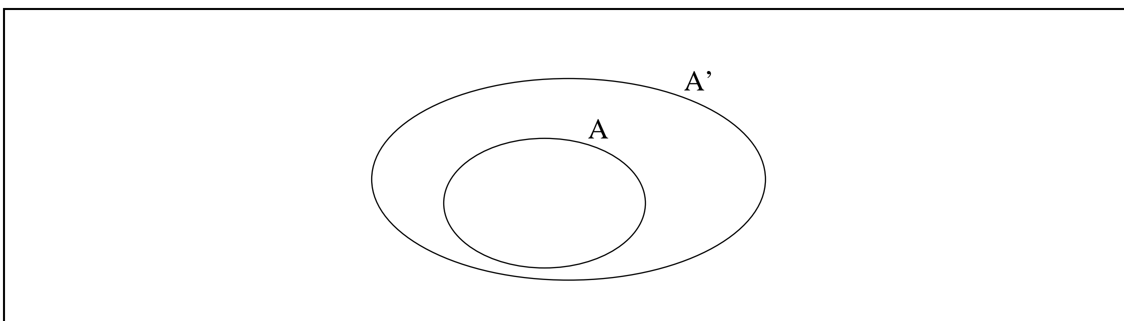
Als eerste moet er gekeken worden naar welke attributen uit de oude situatie ook in de nieuwe situatie voorkomen. We vergelijken de verzameling A met A'. In de figuren 10.1, 10.2, 10.3, 10.4 en 10.5 zijn de vijf mogelijke situaties weergegeven.



Figuur 10.1: A en A' zijn twee disjuncte verzamelingen.



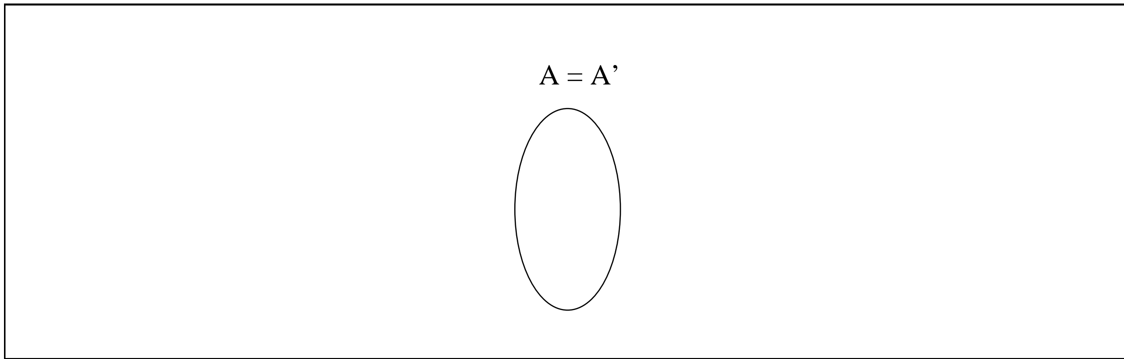
Figuur 10.2: A en A' hebben overeenkomstige elementen.



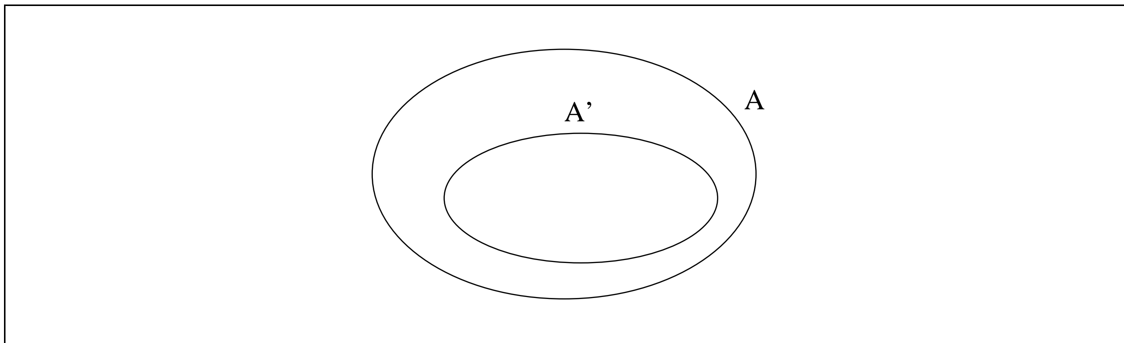
Figuur 10.3: A is een deelverzameling van A'.

De situatie in Figuur 10.1 zal zich in de praktijk niet voordoen. Het zou betekenen dat er niets van de oude data hergebruikt gaat worden. Omdat we in deze scriptie over *legacy systemen* praten, gaan we er vanuit dat er ook echt iets geërfd wordt.

De situaties in Figuur 10.2 en Figuur 10.3 geven de meest realistische situaties weer. In Figuur 10.2 wordt een deel van de oude data hergebruikt en wordt er nieuwe data toegevoegd. Een deel



Figuur 10.4: A en A' hebben overeenkomstige elementen.



Figuur 10.5: A' is een deelverzameling van A.

van de oude data wordt niet meer gebruikt. Figuur 10.3 geeft een uitbreiding van de oude situatie weer, alle oude data blijft behouden.

De situatie in Figuur 10.4 doet zich voor als men een oud systeem migreert naar een nieuwe omgeving, zonder dat men nieuwe data toevoegt.

De situatie in Figuur 10.5 doet zich voor als men een deel van een oud systeem migreert naar een nieuwe omgeving, zonder dat men nieuwe data toevoegt.

We pakken de tabellen die we uit Stap 1 en Stap 3 gekregen hebben en stellen de verzameling A en A' op. We doen dit door eenvoudigweg alle attributen van de tabellen uit Stap 1 in A en alle attributen uit Stap 3 in A' te stoppen. Nu moeten we gaan onderzoeken welke attributen van A ook in A' voorkomen. We maken dus een mapping tussen A en A' (zie Figuur 10.6).

We moeten nu nagaan welke attributen uit A overeenkomen met attributen uit A'. Als we naar een attribuut $a \in A$ kijken kunnen er zich twee situaties voordoen:

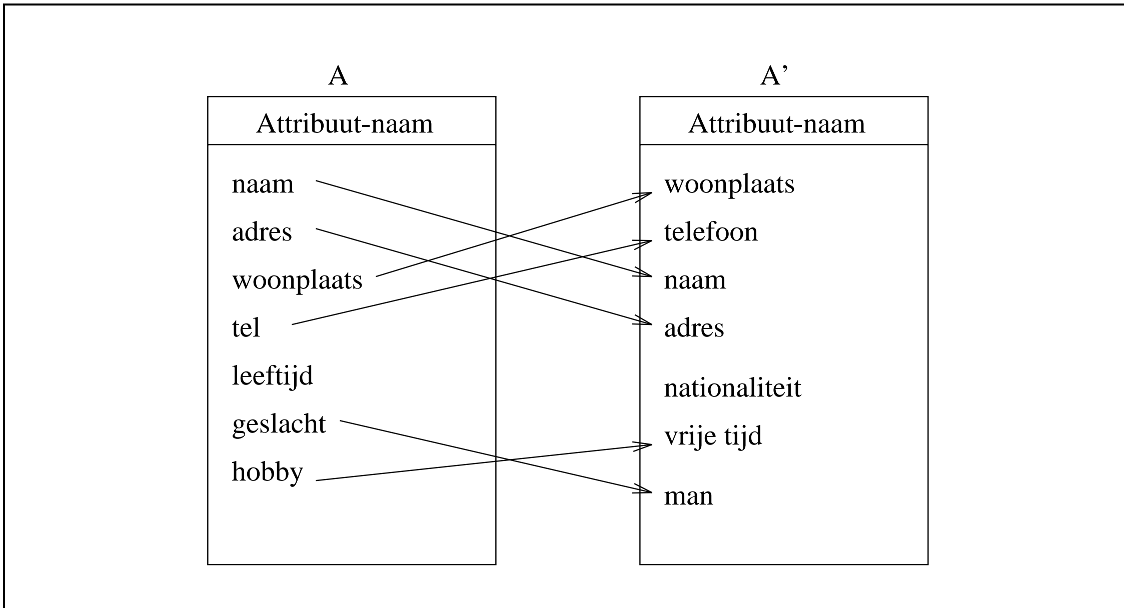
1. Er is een $a' \in A'$ met: $naam(a) = naam(a')$.
2. Er is geen $a' \in A'$ met: $naam(a) = naam(a')$.

In geval één hebben we waarschijnlijk een mapping voor a te pakken. Door de omschrijving van a met de omschrijving van a' te vergelijken, kunnen we beslissen of a gelijk is aan a'.

In geval twee moeten we gaan kijken of er misschien een $a' \in A'$ is waarvoor geldt dat de omschrijvingen gelijk zijn, wat betekent dat a gelijk is aan a'.

Ten slotte kunnen we nog een restverzameling van A en van A' overhouden. De restverzameling van A duidt op informatie die in de toekomst niet meer nodig is. De restverzameling van A' duidt op nieuwe informatie die nog nooit is vastgelegd.

Het beslissen of a en a' gelijk zijn, kan ondersteund worden door met de gebruikers te praten. Wat verstaan zij onder a?



Figuur 10.6: We maken een mapping tussen A en A'.

10.3 De Migratie

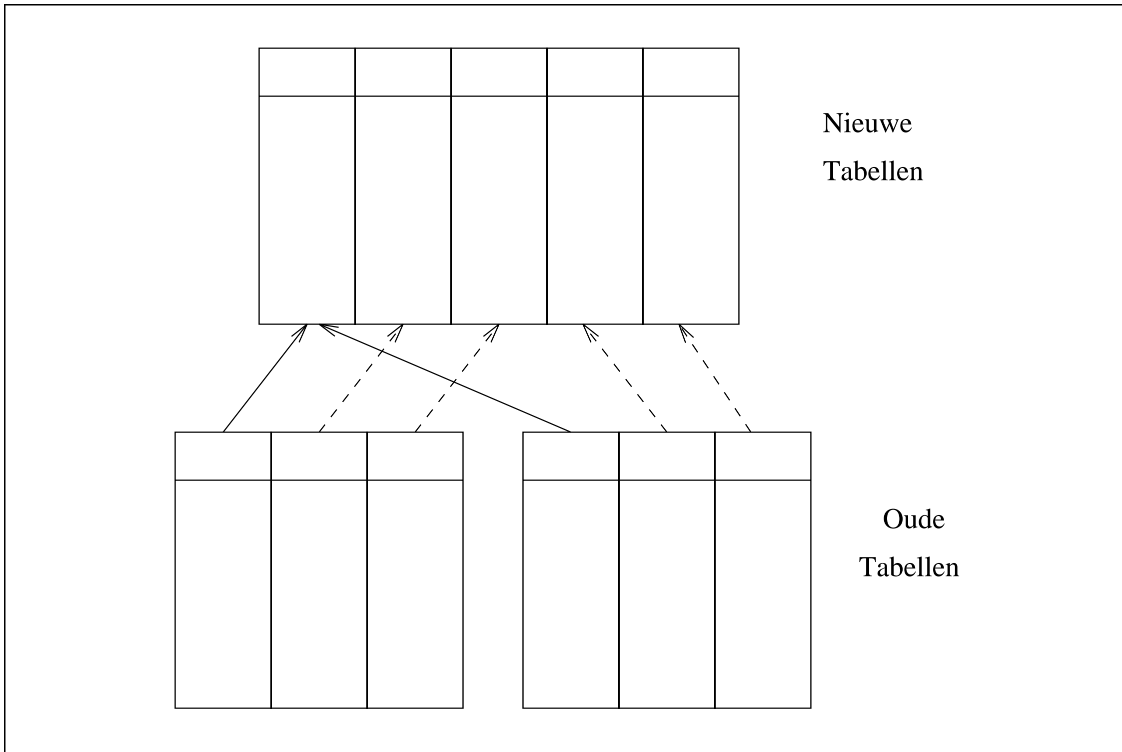
We weten nu van de tabellen uit de oude en nieuwe situatie welke attributen gelijk zijn. Met deze informatie kunnen we aan de slag gaan en de oude tabelstructuren stap voor stap omvormen tot de nieuwe tabelstructuren. Vervolgens moeten we de oude situatie *virtueel* herstellen zodat de oude programma's gebruikt kunnen blijven worden.

10.3.1 Maak Nieuwe Tabellen uit de Oude Tabellen

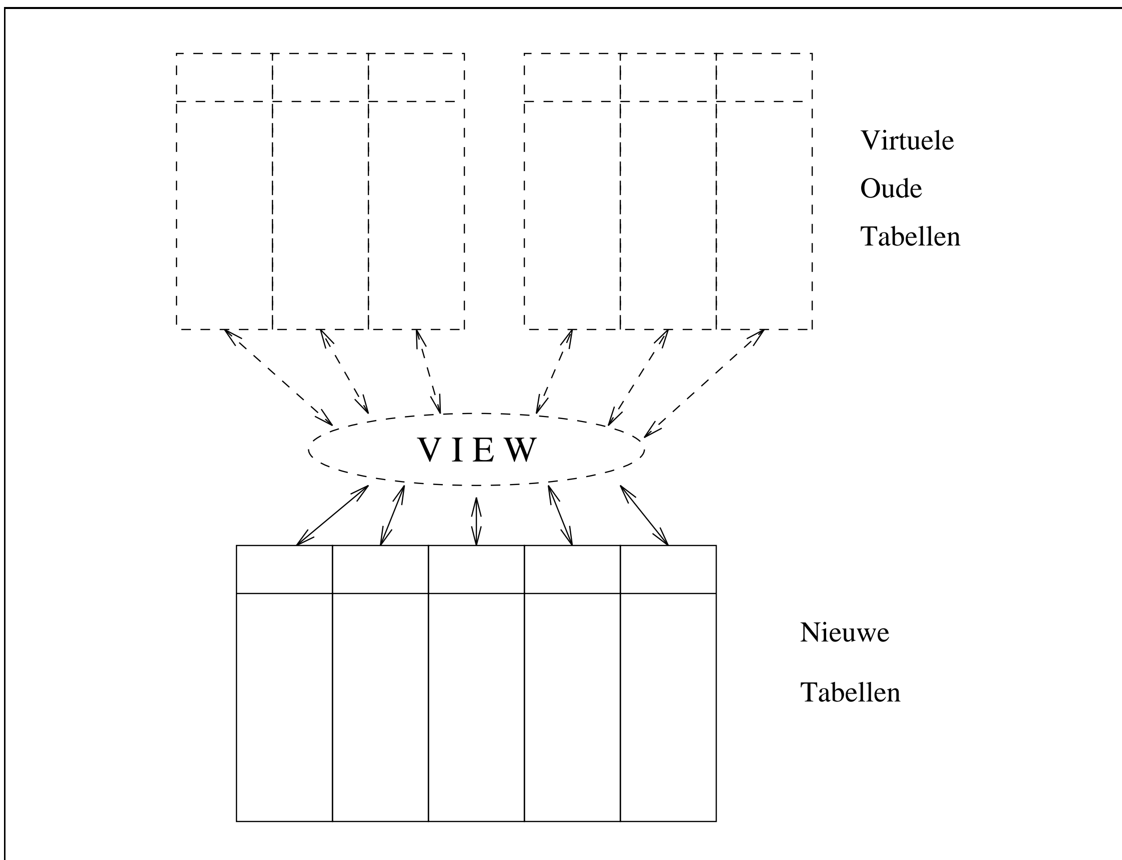
Als eerste worden de nieuwe tabelstructuren gecreëerd. Deze worden vervolgens gevuld. Het creëren gaat met behulp van het SQL commando *CREATE TABLE*. Het vullen van de nieuwe tabellen met de informatie uit de oude tabellen gaat met behulp van het SQL commando *INSERT*. Figuur 10.7 geeft dit alles schematisch weer.

10.3.2 Maak Virtuele Oude Tabellen uit de Nieuwe Tabellen

Omdat ook de oude programma's moeten blijven draaien, moeten de oude tabelstructuren hersteld worden. Dit herstellen gebeurt door gebruik te maken van het *view* mechanisme dat SQL kent. Hierdoor wordt de oude situatie *virtueel* hersteld. Update van de nieuwe tabellen gebeurt automatisch door het *view* mechanisme. Figuur 10.8 geeft dit alles schematisch weer.



Figuur 10.7: Het maken van nieuwe tabellen uit oude tabellen.



Figuur 10.8: Het maken van virtuele oude tabellen uit nieuwe tabellen.

Hoofdstuk 11

Stap 5: Migreer Oude Software

Op dit moment hebben we een architectuur gerealiseerd waarmee de 'legacy' systemen en de nieuwe object georiënteerde systemen naast elkaar kunnen draaien. Het zou kunnen zijn dat we bepaalde functionaliteiten van de oude systemen willen migreren naar de nieuwe omgeving. Kan ik de oude functionaliteit echter zomaar migreren? Dit hoofdstuk geeft een analyse methode om de 'legacy' systemen te analyseren op stukken software die onafhankelijk gemigreerd kunnen worden.

11.1 Migratie van Programma's P_i

In het geval dat we een volledig programma P_i willen migreren naar een object georiënteerde omgeving, zijn er weinig moeilijkheden met de afhankelijkheden tussen de oude programma's. Bij de analyse van de programma's P_i is er van uit gegaan dat deze programma's onafhankelijk van elkaar draaien. We kunnen dus stellen dat: Elke P_i zonder afhankelijkheidsproblemen te migreren is naar een object georiënteerde omgeving, omdat elke P_i onafhankelijk is van P_j.

11.2 Migratie van een Deel van P_i

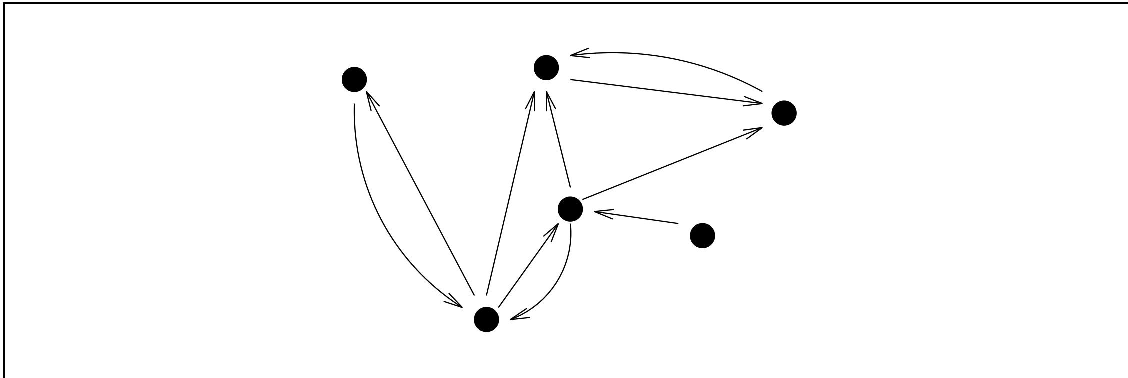
Als we niet de hele P_i maar een deel van P_i willen migreren, doordat bijvoorbeeld een deel efficiënter moet worden, moeten de delen binnen P_i een lage *koppelingsgraad* hebben, omdat delen die sterk van elkaar afhankelijk zijn samen gemigreerd moeten worden. Sommerville[Som89] definieert het begrip koppeling als volgt:

> Coupling is an indication of the strength of interconnection between program units. Highly coupled systems have strong interconnections, with program units which are dependant on each other, whereas loosely coupled systems are made up of units which are independent or almost independent. As a general rule, modules are tightly coupled if they make use of shared, global variables or if they interchange control information. Constantine and Yourdon call this common coupling and control coupling. Loose coupling is achieved by ensuring that, wherever possible, representation information is held within a component and that its data interface with other units is via its parameter list.

Het vaststellen van de koppelingsgraad binnen een programma P_i is mogelijk door een *gerichte graph* op te stellen van de functies/procedures binnen P_i. In Jeurissen[Jeu91] wordt een graph gedefinieerd:

> Een gerichte graph is een paar $< Q, L >$ waarin Q een eindige niet lege verzameling is en L een verzameling van geordende tweetallen elementen van Q, met $Q \cap L = \phi$. De elementen zijn dus van de vorm (x, y) met $x, y \in Q, x \neq y$; L is deelverzameling van $Q \times Q$.

In de gerichte graph komen dus de functies F_k van P_i te staan. Om misverstanden te voorkomen, wordt afgesproken dat punt k in de gerichte graph functie F_k voorstelt. Figuur 11.1 geeft een voorbeeld van zo'n gerichte graph.



Figuur 11.1: Gerichte Graph.

Een functie F_k is migreerbaar als geldt: onafhankelijk(F_k).

Onafhankelijk(F_k) geldt als voor alle $j \in P \backslash \{k\}$ geldt: onafhankelijk(k,j).

Onafhankelijk(k,j) geldt als $< k, j > \notin L$ en $< j, k > \notin L$.

Het is ook mogelijk dat een aantal functies F' samen een onafhankelijke eenheid vormen. Deze functies moeten dan samen gemigreerd worden. $F' \subset F$ is migreerbaar als geldt onafhankelijk(F').

onafhankelijk(F') geldt als voor alle $j \in F \backslash \{F'\}$ en $i \in F'$ geldt: onafhankelijk(i,j).

Als een functie F_i gemigreerd moet worden, moet er eerst gekeken worden of geldt: onafhankelijk(F_i). Als dat niet het geval is dan moet er gekeken worden of er een $F' \subset F$ is waarvoor onafhankelijk(F') geldt. We kunnen eenvoudig afleiden dat elke F_i migreerbaar is omdat $F \subset F$ en $F_i \in F$ en onafhankelijk(F) geldt.

Hoofdstuk 12

Tips voor Management van de Methode

In dit hoofdstuk komen een aantal tips voor het management van de methode aan bod. Als eerste zal gekeken worden voor welke soort systemen de methode toepasbaar is. Vervolgens worden een aantal factoren belicht die het slagen van de methode bepalen. Als laatste wordt ingegaan op de project scheduling.

12.1 Doelgebied

Voordat men deze methode gaat gebruiken, is het belangrijk dat men vooraf inschat of de methode een *werkbare* methode is voor de specifieke situatie.

Het *Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd* beschreven in deze scriptie, is toepasbaar in het *RDBMS naar RDBMS* geval. Dat betekent dat vooraf goed gekeken moet worden of het specifieke probleem zich in deze categorie laat plaatsen. Dat wil zeggen dat de huidige situatie een relationeel databasemanagement systeem met SQL omvat en dat het toekomstige object model relationeel opgeslagen gaat worden. Valt het specifieke probleem niet in de categorie *RDBMS naar RDBMS* dan kan eventueel de *Eerste aanzet tot een Model voor de Evolutie van Legacy Systemen* gebruikt worden. Deze is toepasbaar voor een breed scala van specifieke probleemsituaties. Het nadeel van deze eerste aanzet is echter het ontwikkelen van een *Forward Gateway*. Het ontwikkelen van een Forward Gateway is een grote technische uitdaging en kan veel tijd en geld kosten. Van te voren moet dus goed de haalbaarheid van een Forward Gateway bekeken worden. Het is onmogelijk om algemeen aan te geven wanneer een Forward Gateway haalbaar is of niet, omdat de slagingskans sterk afhankelijk is van de manier waarop de oude software gebouwd is. Bij elke situatie zal steeds opnieuw weer ingeschat moeten worden of zo'n Forward Gateway haalbaar is of niet.

Bij veel bedrijven praten verschillende afdelingen over dezelfde entiteiten uit de reële wereld met een verschillende betekenis. Afdeling A praat bijvoorbeeld over een entiteit **persoon** met een attribuut **naam**. Zo'n naam bij afdeling A ziet er bijvoorbeeld als volgt uit: **Koning, J.A.**. Een afdeling B praat over een zelfde entiteit **persoon** met attributen **achternaam** en **voorletters**. Een moeilijkere situatie doet zich voor als verschillende afdelingen over de entiteit klant praten met elk een andere betekenis in de reële wereld (UoD). Zo praat afdeling A bijvoorbeeld over een klant in de zin van een bedrijf, een afdeling B over een klant in de zin van een afdeling en een afdeling C weer over een klant in de zin van een betaler. In de praktijk komt het voor dat de situatie niet een 1 op 1 relatie kent met de werkelijkheid. Het integreren van zulke databases met verschillende definities kan een probleem opleveren. De methode beschreven in deze scriptie, gaat niet op deze problemen in en het is dus belangrijk dat van te voren deze problemen opgelost worden.

De methode in deze scriptie gaat uit van een object model. Het goed slagen van de methode valt of staat met hoe goed het ontworpen object model is. Vandaar dat het heel belangrijk is dat een

passende object georiënteerde methode gekozen wordt bij het specifieke probleem. In de praktijk is het helaas zo dat veel object georiënteerde methoden een ontwerp afhankelijk van de implementatie taal maken. Rumbaugh, Blaha, Premerlani, Eddy en Lorenson[RBP+91] en Kristen[Kri93] beweren een onafhankelijk ontwerp te produceren, terwijl Meyer[Mey88] ervan uitgaat dat Eiffel de uiteindelijke implementatietaal wordt. Een ander feit is dat sommige methoden bij een bepaald toepassingsgebied een grotere slagingskans hebben dan bij een ander toepassingsgebied. In een onderzoek dat Howard en Potter[HP94] onlangs hebben uitgevoerd blijkt dat sommige methoden in bepaalde toepassingsgebieden uitstekend werken en in andere situaties volkomen te falen.

Als we de bovengenoemde punten nog eens op een rijtje zetten, krijgen we het volgende beslissings-schema:

1. *RDBMS naar RDBMS* Geval?
   - Ja, ga naar 2.
   - Nee, is een Forward Gateway haalbaar?
     - Ja, ga naar 2.
     - Nee, slagingskans is gering met de methode in deze scriptie.
2. Zijn de definitie problemen op te lossen?
   - Ja, ga naar 3.
   - Nee, slagingskans is gering met de methode in deze scriptie.
3. Is een acceptabel object model te maken?
   - Ja, slagingskans is groot met de methode in deze scriptie.
   - Nee, slagingskans is gering met de methode in deze scriptie.

12.2 Risico Factoren

In de vorige paragraaf is een aantal mogelijke knelpunten gesignaleerd en weergegeven in een beslissings-schema. In deze paragraaf wordt een aantal risico factoren gisignaleerd die in de gaten gehouden moeten worden om het *Model voor de Evolutie van Legacy Systemen naar Object Geriënteerd* goed te laten slagen.

Het overschakelen naar het object georiënteerd paradigma kost veel tijd. Uit gesprekken met mensen die zich het object georiënteerd paradigma eigen gemaakt hebben, blijkt dat men na ongeveer anderhalf jaar werken met object georiënteerd genoeg kennis en praktische ervaring in huis heeft van het object georiënteerd paradigma. Het is belangrijk dat de mensen die het evolutie-traject gaan uitvoeren voldoende kennis en ervaring in huis hebben om het traject te laten slagen.

Het is belangrijk dat het systeem-domein voor iedereen duidelijk en hetzelfde is. Als het systeem-domein onduidelijk of voor een aantal personen verschillend is, verschuift de aandacht in het traject van de werkelijke architecturele problematiek naar het systeem-domein en is er op het einde te weinig tijd over om aan de echte problematiek toe te komen.

Voor de systeem-eisen geldt hetzelfde als voor het systeem-domein. De systeem-eisen moeten duidelijk, consistent en stabiel zijn opgesteld. Bij problemen verschuift de aandacht vaak van de werkelijke problematiek naar de systeem-eisen problematiek en is er op het einde te weinig tijd over voor de werkelijk problematiek.

Het is belangrijk dat elke stap goed doordacht gemaakt wordt. Overslaan van een stap of het niet goed uitvoeren van een stap is van directe invloed op het uiteindelijk resultaat. Vandaar dat het belangrijk is dat er voldoende tijd is voor het gehele traject.

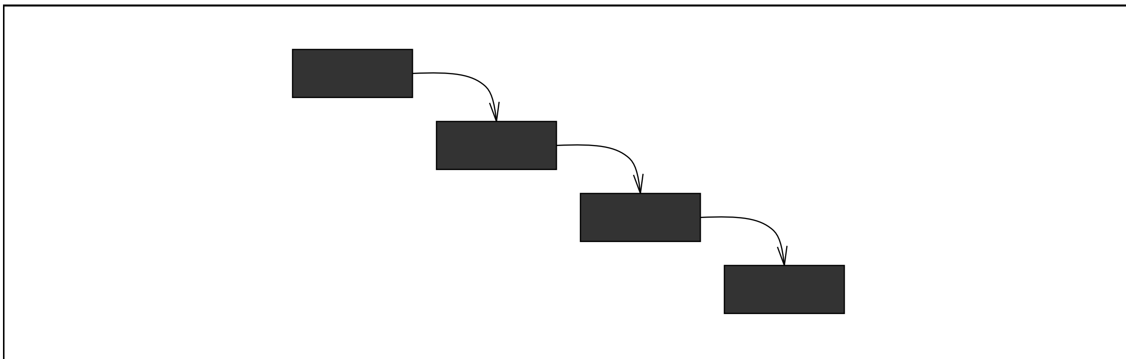
Vaak vindt een overschakeling naar object georiënteerd plaats in één of meerdere *pilot projecten*. Om de mensen te overtuigen dat een overschakeling echt mogelijk is met hergebruik van oude systemen, is het belangrijk dat de mensen die overtuigd moeten worden vantevoren realistische

verwachtingen koesteren. Bij te hoge verwachtingen wordt zo'n pilot project vaak als *niet rendabel* van de tafel geschoven.

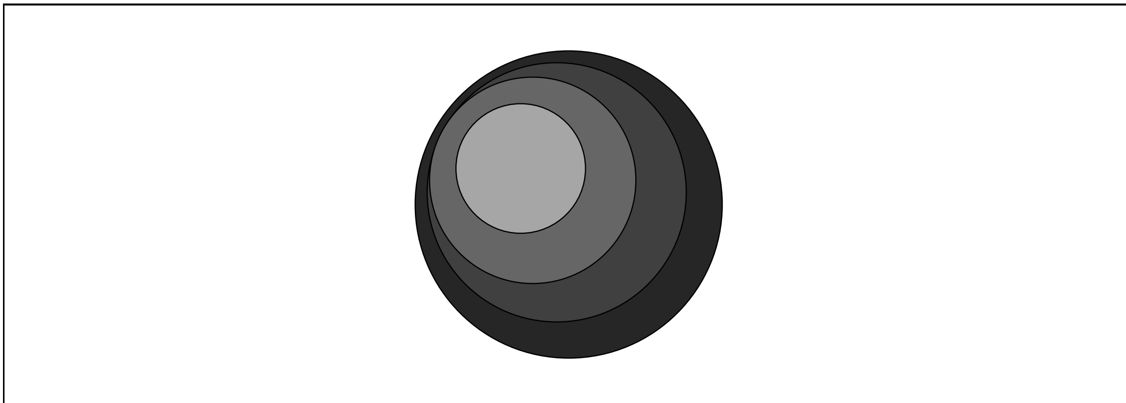
Het is belangrijk dat iedereen overtuigd is van het nut van een overschakeling naar object georiënteerd. Afdelingen die niet overtuigd worden van het nut en niet in het proces betrokken worden, kunnen zich in hun voortbestaan bedreigd voelen. Hun kennis raakt verouderd en is in de toekomst misschien niet meer nodig. Deze afdelingen zouden zich kunnen gaan verzetten tegen een mogelijke overschakeling. Door deze afdelingen het te doorlopen traject te laten zien en ze te overtuigen dat de nieuwe architectuur het mogelijk maakt dat zowel de oude als de nieuwe technologieën naast elkaar bestaan, kunnen veel politieke spelletjes vermeden worden.

Het is belangrijk om met een klein, te overzien stukje te beginnen. Hierdoor ligt de nadruk op de te realiseren architectuur en niet op de complexiteit van het probleem. Met de opgedane kennis kan een eventueel groter vervolgproject met een grotere slagingskans uitgevoerd worden. Het principe van verdeel en heers gaat in deze situatie op.

In het bedrijfsleven komt het soms voor dat de klant een deel van het ontwerp zelf aanlevert. In zo'n geval is het heel belangrijk dat van het begin af aan de opdrachtgever en de uitvoerder samen aan tafel gaan zitten en afspreken wat er en hoe het opgeleverd moet worden.



Figuur 12.1: Waterval Methode.



Figuur 12.2: Cyclische Ontwerpmethoden.

De kans is groot dat de klant de spullen volgens conventionele methoden (zie Figuur 12.1) oplevert, terwijl het object georiënteerd ontwerpen een meer cyclische methode is (zie Figuur 12.2). Als men niet in een vroeg stadium met de klant aan tafel gaat zitten, bestaat het gevaar dat de klant om de zoveel tijd een pak onbruikbare ontwerp-documenten over de muur gooit waar het migratie-team niets of nauwelijks iets mee kan doen. Dan moet er veel tijd gestoken worden in discussie.

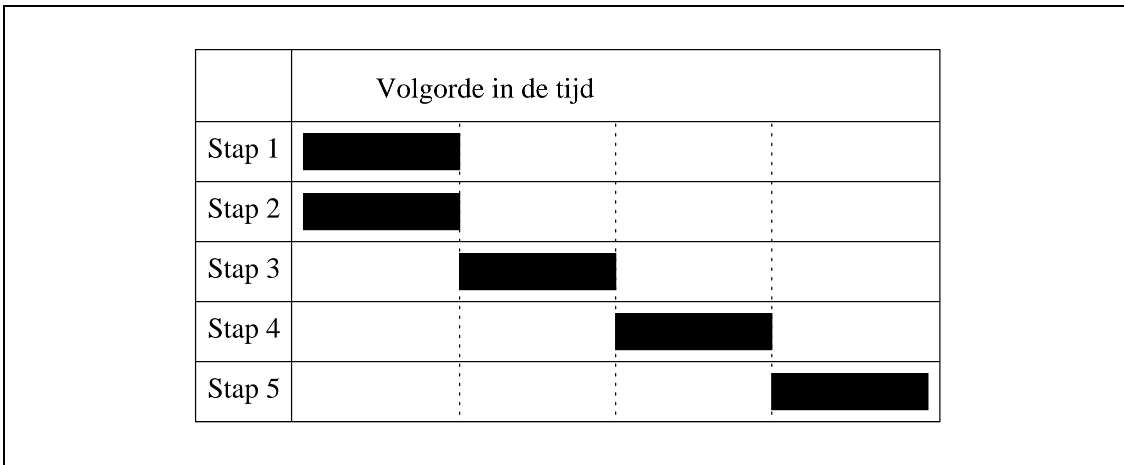
De beschreven risico's in dit hoofdstuk monden uit in een check-list:

- Is er genoeg kennis van object georiënteerd paradigma aanwezig?
- Is het systeem-domein duidelijk?
- Zijn de systeem-eisen duidelijk, consistent en stabiel opgesteld?
- Is er voldoende tijd?
- Zijn de verwachtingen realistisch?
- Zijn er politieke belangen die omgebogen moeten worden?
- Is het project klein genoeg?
- Sluiten de ontwerpmethoden op elkaar aan?

12.3 Project Scheduling

Eén van de moeilijkste taken bij het management van een project is de project scheduling. In deze paragraaf zal een globale scheduling gegeven worden, die als basis kan dienen voor de scheduling van een werkelijk migratie-plan.

Als we naar de methode beschreven in deze scriptie kijken, zien we dat *Stap 1: Analyseer Huidige Situatie* en *Stap 2: Maak Object Model Toekomstige Situatie* onafhankelijk van elkaar uitgevoerd kunnen worden, omdat Stap 1 de huidige situatie en Stap 2 de toekomstige situatie bekijkt. *Stap 3: Omzetten Object Model naar Relationeel Model* heeft het object model van Stap 2 als basis en moet dus na Stap 2 plaatsvinden. *Stap 4: Maak Nieuwe en Virtuele Oude DataBase* heeft de huidige tabellen van Stap 1 en de toekomstige tabellen van Stap 3 als basis. Stap 4 moet dan ook na Stap 1 en Stap 3 uitgevoerd worden. *Stap 5: Migreer Oude Software* moet na Stap 4 uitgevoerd worden omdat, voordat er een migratie plaats kan vinden, de totale architectuur gerealiseerd moet zijn. Als we dit alles in een plaatje vastleggen krijgen we Figuur 12.3.



Figuur 12.3: Scheduling van het Interface Approach Stappenplan.

In een aantal ontwerpmethoden wordt vaak eerst de huidige situatie geanalyseerd. Vervolgens wordt er gekeken naar wat er toen mis ging en waarom dat mis ging. Uiteindelijk wordt dan bekeken hoe dit opgelost kan worden en aan welke eisen het nieuwe systeem dus moet voldoen. In zo'n geval is het duidelijk dat *Stap 1: Analyseer Huidige Situatie* en *Stap 2: Maak Object Model Toekomstige Situatie* niet volledig parallel uitgevoerd kunnen worden.

Hoofdstuk 13

Verder Onderzoek

Dit hoofdstuk geeft een aantal punten voor verder onderzoek aan. Het stipt nieuwe technologieën en tools aan die het migratie proces gigantisch zouden vergemakkelijken.

13.1 Mapping Tools

Onderzoek naar automatische mapping van de oude data-structuren naar de nieuwe data-structuren zou de *Interface Approach Method*, met betrekking tot *Stap 4: Maak Nieuwe en Virtuele Oude DataBase*, aanzienlijk kunnen vergemakkelijken.

13.2 Forward Gateway

De Forward Gateway is het kritieke deel van de *Interface Approach Method.* Ze zijn een grote technische uitdaging en kunnen het meeste tijd en geld kosten. Onderzoek naar de Forward Gateway maakt het idee van deze scriptie algemener toepasbaar.

13.3 Specificatie Extractor

Bij legacy systemen zijn specificaties soms niet aanwezig, de code zelf is dan de enige documentatie. Een programma dat uit de code een specificatie extraheert, kan het proces van het herschrijven van programma's aanzienlijk vergemakkelijken. Dit zou een ondersteuning bieden voor *Stap 6: Migreer oude Software.*

Er zijn al enige vorderingen op dit gebied. Zo gebruikt de afdeling I&AT van PTT Telecom B.V. al een aantal tools die code als input krijgen en eenvoudige specificaties als output geven.

13.4 Automatische Afhankelijkheids Analysator

Belangrijk bij het vaststellen welke delen van de oude systemen in één keer gemigreerd moeten worden, is de *koppelingsgraad* . Een tool die een programma analyseert op afhankelijkheden en deze in een afhankelijkheidsgraph weergeeft, zou een goede ondersteuning bieden van *Stap 6: Migreer oude Software.*

13.5 SQL-3 en SQL-4

Er wordt momenteel hard gewerkt aan de opvolgers van SQL-2 , de standaard gebruikt in deze scriptie. De opvolgers, SQL-3 en SQL-4, zouden het object georiënteerd paradigma gaan ondersteunen. Onderzocht zou moeten worden in hoeverre SQL-3 en SQL-4 dat werkelijk gaan doen.

Als SQL-3 en SQL-4 een werkelijke integratie van de object wereld en de relationele wereld bewerkstelligen zou dit een groot voordeel opleveren voor de in deze scriptie beschreven methode. Het wordt dan mogelijk om binnen hetzelfde databasemanagement systeem[1] de data te migreren van een relationele view naar een object georiënteerde view.

13.6 Organisatorische Eisen voor Slagen van Migratie

Door mijn stage bij het Tripoli-1 project heb ik een aantal factoren kunnen opsporen die van invloed zijn op het wel of niet slagen van een migratie-plan. Omdat één project waarschijnlijk niet voldoende is om alle factoren op te sporen, zou het wenselijk zijn om de ervaring van meerdere projecten samen te voegen in een algemene factoren-lijst die van belang is bij een migratie.

13.7 Performance van View Mechanisme

Daar de methode beschreven in deze scriptie voor een groot deel steunt op het *view* mechanisme van SQL, is het belangrijk dat ook naar de performance van het *view* mechanisme onderzoek gedaan wordt.

13.8 Integratie Verschillende DBMS-en

De architectuur die als uitgangspunt dient voor deze scriptie, gaat er van uit dat er slechts één DBMS aanwezig is in een situatie. Het komt in de praktijk echter vaak voor dat er verschillende DBMS-en zijn die geïntegreerd moeten worden indien men wil migreren naar een nieuwe 'betere' situatie. Belangrijke vragen zijn dan: 'Hoe maak ik een mapping tussen de verschillende DBMS-en?', 'Kan ik de verschillende DBMS-en wel integreren?' of 'Kan ik de gegevens hergebruiken of moet ik alle gegevens opnieuw intypen?'. Onderzoek naar de integratie van verschillende DBMS-en zou de methode beschreven in deze scriptie voor meer problemen inzetbaar maken.

13.9 Definitie Problemen bij Integratie van Verschillende Databases

Een veel voorkomend probleem is dat twee verschillende afdelingen praten over één en dezelfde entiteit uit de reële wereld die voor de beide afdelingen een verschillende betekenis hebben. Het integreren van databases met zulke problemen is zeker niet triviaal. Onderzoek naar deze problemen zou de methode beschreven in deze scriptie bruikbaar maken voor meerdere situaties.

[1] We spreken dan niet meer van een *RDBMS* of een *ODBMS* maar van een DBMS dat én de relationele én de object georiënteerde wereld implementeert.

Hoofdstuk 14

Conclusies

In dit hoofdstuk zal een aantal conclusies getrokken worden uit mijn onderzoeksresultaten.

Aan het begin van mijn onderzoek dacht ik een algemene oplossing te kunnen presenteren voor het 'legacy' probleem. Dat mijn idee van het 'legacy' probleem veel te nauw was, bleek al heel snel tijdens mijn stage bij het Tripoli-1 project bij de afdeling I&AT van PTT Telecom B.V. Het bleek dat ik vanuit de theorie niet voldoende inzicht had gekregen in de werkelijke 'legacy' problematiek, in werkelijkheid was deze veel complexer dan dat ik mij voorstelde. Het gebruiken van standaarden binnen bedrijven vond vroeger alleen binnen één applicatie plaats, omdat deze stand-alone draaide. Tegenwoordig eist de situatie dat systemen met elkaar kunnen communiceren en dat delen hergebruikt kunnen worden. Hierdoor is het noodzakelijk dat er standaarden gemaakt worden die over de applicaties heen gelden. De 'legacy'-systemen die vele jaren ouder zijn, zijn niet volgens deze standaarden gemaakt. Documentatie is vaak niet gemaakt. Er worden vele verschillende DBMS-en gebruikt, als deze überhaupt al gebruikt worden. Binnen een bedrijf praten verschillende afdelingen over dezelfde entiteiten uit de reële wereld met verschillende definities.

Wat ook snel duidelijk werd, is dat niet alleen technische factoren van invloed zijn op het slagen van een migratie-plan. Organisatorische en politieke obstakels moeten vaak eerst overwonnen worden om aan een succesvolle migratie te kunnen beginnen.

De door mij beschreven methode is niet alleen een model voor de evolutie van legacy-systemen naar OO, maar integreert ook twee verschillende werelden met elkaar. De conventionele wereld en de OO wereld kunnen met de beschreven architectuur naast elkaar blijven bestaan. Dit is een belangrijk argument om afdelingen, die niet in de evolutie naar OO worden meegenomen, toch achter de plannen te laten staan. Het meekrijgen van alle afdelingen is van essentieel belang voor het slagen van een migratie-plan.

Het *Model voor de Evolutie van Legacy Systemen naar Object Georiënteerd* maakt de migratie van legacy systemen naar OO mogelijk voor het *RDBMS naar RDBMS* geval. Voor de andere gevallen is migratie eveneens mogelijk als een Forward Gateway haalbaar is.

In het onderstaande overzicht worden alle conclusies nog eens kort opgesomd:

- Voor *RDBMS naar RDBMS* geval is migratie van legacy systemen mogelijk.
- Migratie van legacy systemen is mogelijk voor alle andere gevallen waar een Forward Gateway realiseerbaar is, waar er geen definitie problemen zijn, waar er geen verschillende databases zijn, waar er een object model te maken is, waar er voldoende geld en tijd is en waar de politieke en organisatorische problemen onder controle te houden zijn.
- Integratie van de conventionele wereld en de object georiënteerde wereld is mogelijk.
- Niet alleen technische factoren zijn van invloed op het wel of niet slagen van een migratie-plan. Organisatorische en politieke factoren hebben vaak een nog grotere invloed op het wel of niet slagen van een migratie-plan.

- Het 'legacy'-probleem is eigenlijk een verzamelnaam voor een heleboel specifieke problemen. Deze scriptie lost slechts één zo'n specifiek probleem op.

Bibliografie

[AG89] A.W. Abcouwer and L.H. Geesink. *Leerboek Relationele Databases met SQL en QBE.* Uitgeverij Tutein Nolthenius, The Netherlands, 1989.

[Boo91] G Booch. *Object-Oriented Design with Applications.* Benjamin Cunnings, 1991.

[BS93] M.L. Brodie and M. Stonebraker. Darwin: On the incremental migration of legacy information systems. *College of Engineering - University of California - Berkeley,* 1993.

[Cod90] E.F. Codd. *The Relational Model for Database Management.* Addison-Wesley Company, 1990.

[DH89] L. Dusink and P. Hall. *Software Re-use, Utrecht 1989.* Springer-Verlag, Germany, 1989.

[HP94] P. Howard and C. Potter. *Case and Methods Based Development Tools - an evaluation and a comparison.* Milton Keyns - U.K., 1994.

[JCJvO92] I. Jacobson, M. Christerson, M. Jonsson, and P. van Overgaard. *OO Software Engineering, A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jeu91] R.H. Jeurissen. Graph algoritmen voor informatica studenten. 1991.

[KC86] S. Khoshafian and G. Copeland. Object identity. *SIGPLAN Notices vol.21*, 1986.

[Kri93] G.J.H.M. Kristen. *KISS-methode voor Object Orientatie.* Academic Service, 1993.

[Mcl76] M.D. Mcllroy. Mass-produces software components. *Software Engineering Concepts and Techniques (1968 NATA Conf. on Software Engineering)*, pages 88–98, 1976.

[Mey88] B. Meyer. *Object-oriented software construction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[NEK94] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

[PB94] W. Premerlani and M. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Som89] I. Sommerville. *Software Engineering.* Addison-Wesley, Reading, Massachusetts, 1989.

[vdL89] R.F. van der Lans. *The SQL Standard, a complete reference.* Prentice Hall International (UK), Great Britain, 1989.

[Wei88] Th.P. van der Weide. Datastructuren en Informatiesystemen. Lecture Notes (In Dutch), University of Nijmegen, 1988.

Auteurs Index

A.W. Abcouwer 24, 65

M. Blaha 1, 7, 9, 13, 37–39, 58, 65
G Booch 7, 9, 13, 65
M.L. Brodie 16, 65

M. Christerson 7–8, 13, 37, 65
E.F. Codd 23, 29, 65
G. Copeland 7, 65

L. Dusink 11–12, 19, 65

F. Eddy 1, 7, 9, 13, 37, 39, 58, 65
A. Engberts 19, 65

L.H. Geesink 24, 65

P. Hall 11–12, 19, 65
P. Howard 58, 65

I. Jacobson 7–8, 13, 37, 65
R.H. Jeurissen 55, 65
M. Jonsson 7–8, 13, 37, 65

S. Khoshafian 7, 65
W. Kozaczynski 19, 65
G.J.H.M. Kristen 37, 58, 65

R.F. van der Lans 24, 65
W. Lorenson 1, 7, 9, 13, 37, 39, 58, 65

M.D. Mcllroy 12, 65
B. Meyer 7–9, 11–12, 37, 58, 65

J.Q. Ning 19, 65

P. van Overgaard 7–8, 13, 37, 65

C. Potter 58, 65
W. Premerlani 1, 7, 9, 13, 37–39, 58, 65

J. Rumbaugh 1, 7, 9, 13, 37, 39, 58, 65

I. Sommerville 11, 55, 65
M. Stonebraker 16, 65

Th.P. van der Weide 23, 65

Index

afhankelijkheid 55–56
analyse 37
associatie 7–8, 37–40
attribuut 7–8, 13, 23, 37, 39, 49
automatische afhankelijkheids analysator 61

Backus Naur Form 24
binaire associatie 40
black-box 12
BNF 24

candidate key 38
compatibility 11
component 11–12
compositionele benadering 12
conceptueel schema 24
conventionele wereld 15
cyclische methode. 59

data 23
Data Definition Language 24
Data Manipulation Language 24
data model 7
data structuur 35
DataBase 15
datadictionary 37
definitie problemen 62
definitieprobleem 15, 18
disjoint multiple inheritance 43
DLL 24
DML 24
doelgebied 57
domei 23
domein 49

eigenschap 7–8
eigenschappen 8
embedded SQL 29
encapsulatie 40
encapsulating the whole system 19
entiteit 57
entiteit integriteit 29
entiteiten 7
evolutie 15
extendibility 11, 40, 42

file 15
flexibel systeem 14
foreign key 29, 38, 41

forward gateway 16, 18, 21, 33–34, 57, 61
functie 8, 11

gedeelde identifier 42
gedrag 8
generalisatie 8–9, 38–39, 42
generalistie 41
gerichte graph 55–56
glass-box 12

herbruikbaar 11
herbruikbaarheid 11
hergebruik 11–12, 14
horizontale partitionering 40

identiteit 7, 42
impedantie probleem 21
informatie systeem 13
inheritance 9
integratie 62
integriteit 29
integriteitsregel 23
Interface Approach Method 18

klasse 7–8, 13, 37–40
kolom 24
koppeling 55
koppelingsgraad 55, 61

legacy probleem 63
legacy systeem 50, 55
link 8, 37
link attribuut 41

maintainability 11
many subclass approach 42
many-to-many associatie 8
mapping 16, 32, 49
mapping table 18, 33
mapping tool 61
meta-data 39
migratie 33, 52
migreerbaar 56
module 37
Module Language 24
more table approach 39, 43
multiple inheritance 9, 43

object 7–8, 37

object georiënteerde wereld 15
object model 7, 15, 18, 33, 37–39, 57
object moodel 43
Object-oriented Modeling and design Technique 7
objecten 7
onafhankelijkeheid 56
onderhoudbaar 11
one superclass table approach 43
one table approach 39, 43
one-to-many associatie 8, 41
operatie 7–9, 11, 13
operaties 7
operator 23
organisatorische eisen 62
overerving 9
overlapping multiple inheritance 43

pilot project 58
politieke factoren 59
primary key 29, 41
programmeertaal-onafhankelijk 7
project scheduling 60

qualified associatie 8, 41
qualifier 41
query-taal 16

RDBMS 23, 39
RDBMS naar RDBMS geval 31
re-use 11
referentiële integriteit 29
relatie 7, 9
relationeel data model 23
relationeel gegevens model 24
relationeel model 33, 43
relationeel schema 24, 49
relationele tabel 32
reusable code recovery 19
reusebility 11
reverse engineering 35
risico factoren 58

software crisis 11–12
specificatie extractor 61
SQL 24
SQL - Close 27
SQL - Commit 27
SQL - Create schema 25
SQL - Create table 25, 52
SQL - Create view 25–26
SQL - Declare 27
SQL - Delete 27
SQL - Fetch 27
SQL - Grant 25
SQL - Insert 27–28, 52
SQL - Module 28
SQL - Not null 26
SQL - Null 26, 41
SQL - Open 27
SQL - Procedure 28
SQL - Rollback 27
SQL - Select 26–27
SQL - Unique 26
SQL - Update 27
SQL - With check option 27
SQL Interface Approach Method 33
SQL-2 61
SQL-3 61
SQL-4 61
star systeem 14
subklasse 9, 42
superklasse 9, 42
superklasse naar subklasse navigatie 42
systeem-domein 58, 60
systeem-eisen 58, 60

tabel 23
ternaire associatie 8–9, 41
tijd 58, 60
tolk 16
transformatie 8, 38
transformationele benadering 11
tripoli-1 project 18
tripoli-1 project 18, 63
tupel 24
tuple 15–16

uitbreidbaar 11
uitbreidbaarheid 40, 42
Universe of Discourse 7, 24, 57
UoD 7, 24, 57
usebility 11

veiligheid 19
verdeel en heers 59
verticale partitionering 40
verwachtingen 60
view mechanisme 21, 32–33, 52, 62
virtuele tabel 26
volledige analyse 37

waterval methode 59
white-box 12