Faculteit Toegepaste Wetenschappen

Vakgroep Informatietechnologie

Voorzitter : Prof. Dr. Ir. P. Lagasse

Een generieke object-georiënteerde bibliotheek voor databanktransacties

Andy Verkeyn

Promotor : Prof. Dr. Ir. H. Tromp

Thesisbegeleiders :

Ir. G. Premereur

Lic. K. Carron

Afstudeerwerk ingediend tot het behalen van de academische graad van

licentiaat toegepaste informatica

Academiejaar 1996 - 1997

Voorwoord

Een thesis schrijven is een opdracht die zeker niet gemakkelijk is. Voor het tot een goed einde brengen van deze zware taak was de deskundige begeleiding van Geert Premereur en Kris Carron dan ook van onschatbare waarde.

Het schrijven van een afstudeerwerk is typisch de laatste fase van de studie. Gedurende deze studie en in het bijzonder tijdens het werken aan deze thesis is de technische en morele steun van talrijke collega's in spe onmisbaar gebleken. Speciale vermelding verdienen Frederik De Vusser en Benny Verplancke.

De vele tik-en taalfouten die tijdens het ijverig werken in dit afstudeerwerk geslopen waren, werden er even ijverig uitgehaald door mijn vriend Bart Lemarcq, waarvoor dank. Ook mijn ouders wil ik hierbij bedanken voor het nalezen van de kladversie.

Verder wens ik al mijn vrienden en vriendinnen te bedanken voor de morele steun die ze me de afgelopen maanden en jaren gegeven hebben alsook voor het delen van de vele plezierige momenten gedurende deze periode. In het bijzonder kunnen worden vermeld: Ingrid Maeyens, Geert Zegers, Wim Vanhulle, Ulrike Vanhessche, Ringo Vanhooren en Hilde Vansteelant.

Deze thesis werd gemaakt met het tekstverwerkingspakket MS-Word 7.0 (Windows 95).

Toelating tot bruikleen

De auteur geeft de toelating dit afstudeerwerk voor consultatie beschikbaar te stellen en delen van het afstudeerwerk te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit dit afstudeerwerk.

5 september 1999

Handtekening

Overzicht

Universiteit Gent
Faculteit Toegepaste Wetenschappen
Vakgroep Informatietechnologie
Voorzitter : Prof. Dr. Ir. P. Lagasse

Een generieke object-georiënteerde bibliotheek voor databanktransacties.

Andy Verkeyn

Afstudeerwerk ingediend tot het behalen van de academische graad van licentiaat toegepaste informatica.

Promotor : Prof. Dr. Ir. H. Tromp
Thesisbegeleiders : Ir. G. Premereur, Lic. K. Carron
Academiejaar 1996 - 1997

Samenvatting

In het inleidend hoofdstuk worden enkele definities van begrippen aangehaald die relevant zijn voor het vervolg van de thesis. Belangrijk hierin is het feit dat we aantonen dat er momenteel twee verschillende modellen veelvuldig in gebruik zijn, namelijk het relationele datamodel en het objectmodel.

Hoofdstuk 2 schetst de problemen die voortvloeien uit het combineren van deze twee modellen. Deze moeilijkheden worden klassiek aangeduid met de term impedance mismatch. Tevens worden de huidige oplossingen daarvoor kort toegelicht. Daarna formuleren we voorstellen om de impedance mismatch op een betere manier aan te pakken. Deze vormen dan ook de doelstellingen van de bibliotheek die in deze thesis ontwikkeld zal worden.

Het ontwerp van deze bibliotheek wordt besproken in hoofdstuk 3. Eerst komt het systeemontwerp en daarna het objectmodel aan bod. Tot slot omschrijven we beknopt de afzonderlijke verantwoordelijkheden van elke klasse.

De belangrijkste implementatie aspecten kan men terugvinden in hoofdstuk 4. Hiertoe behoren onder andere de manier waarop het databankschema gedefinieerd wordt, het geheugenbeheer en de verwerking van conditionele opvragingen.

In hoofdstuk 5 wordt uiteengezet hoe de bibliotheek door de programmeur gebruikt kan worden en wat de mogelijkheden ervan zijn. Deze worden ook geïllustreerd met een paar voorbeelden. Bij wijze van besluit vergelijken we de realisatie met de geformuleerde doelstellingen en maken we ook enkele kritische bedenkingen.

Trefwoorden :

object-oriëntatie, relationele databanken, software-ontwikkeling, persistentie

Acroniemen

ADT : Abstract Data Type

ANSI : American National Standards Institute

API : Application Programming Interface

DBMS : DataBase Management System

ISO : International Standards Organization

ODBC : Open DataBase Connectivity

ODMG : Object Database Management Group

OMT : Object Modeling Technique

OO : Object-Oriented *of* Object-Orientation

OOA : Object-Oriented Analysis

OOD : Object-Oriented Design

OODBMS : Object-Oriented DataBase Management System

OOP : Object-Oriented Programming

OT : Object Technology

RDBMS : Relational DataBase Management System

RTTI : Run-Time Type Information

SQL : Structured Query Language

STL : Standard Template Library

Inhoudstafel

Hoofdstuk 1 : Inleiding

The palest ink is better than the best memory.
Chinese proverb

1. Relationele databanken

1.1. Het relationele datamodel

In wat volgt worden enkele belangrijke databankbegrippen die relevant zijn voor deze thesis beknopt besproken. Voor meer informatie wordt verwezen naar de uitgebreide literatuur die over dit onderwerp beschikbaar is, onder andere [1].
Een RDBMS (Relational DataBase Management System) is gebaseerd op het relationeel datamodel. Dit datamodel werd rond 1970 geïntroduceerd door E.F. Codd en is nog steeds het dominerende model in de databankwereld.
Een datamodel is een manier van kijken naar gegevens. In het geval van het relationeel datamodel worden de gegevens georganiseerd als een verzameling van tabellen die uit rijen en kolommen bestaan. In plaats van tabel, rij, kolom,... gebruikt men soms ook andere termen naargelang de context. Een overzicht van de verschillende termen wordt gegeven in tabel a. In deze thesis zal hoofdzakelijk de informeel relationele terminologie gebruikt worden.

DBMS	Informeel RDBMS	Formeel RDBMS
bestand (file)	tabel (table)	relatie (relation)
record (record)	rij (row)	tuppel (tuple)
veld (field)	kolom (column)	attribuut (attribute)

Tabel A : De verschillende terminologie

Een relationele tabel voldoet aan de volgende eigenschappen :

- meerdere rijen met dezelfde inhoud zijn niet toegelaten

- rijen zijn niet geordend (bijvoorbeeld van boven naar beneden)
- kolommen zijn niet geordend (bijvoorbeeld van links naar rechts)
- alle waarden zijn atomisch

Met atomische waarden wordt bedoeld dat de waarden scalair zijn. Repeterende groepen (dit zijn meerdere waarden per kolom) zijn niet toegelaten.

De kolommen zijn gedefinieerd over een domein waaruit de actuele waarden genomen worden. Dit wordt de attribuut-integriteitsregel genoemd.

Wanneer een kolom niet kan ingevuld worden omdat er informatie ontbreekt of omdat deze kolom voor de rij in kwestie geen betekenis heeft, wordt er een NULL markering aangebracht. Deze NULL is geen echte waarde.

Aan de hand van de kolommen definiëert men kandidaat-sleutels (candidate key) die gebruikt worden voor de identificatie van de rijen. De definitie is als volgt :

Een kandidaat-sleutel voor een relatie R is een deelverzameling van de verzameling van de attributen van R, stel K, zodat altijd het volgende geldt :

- *Uniciteit : geen twee verschillende tuppels van R dezelfde K waarden hebben.*
- *Onreduceerbaar : geen echte deelverzameling van K de uniciteitseigenschap heeft.*

Uit de verzameling van de kandidaat-sleutels wordt er één sleutel uitgekozen als primaire sleutel (primary key). De componenten van de primaire sleutel mogen niet NULL kunnen zijn, dit is de zogenaamde entiteit-integriteitsregel.

Het uitdrukken van relaties tussen tabellen gebeurt met behulp van vreemde sleutels (foreign key). Die zijn als volgt gedefinieerd :

Een vreemde sleutel in een basisrelatie R2 is een deelverzameling van de verzameling van de attributen van R2, stel FK, zodat :

- *er een basisrelatie R1 bestaat (R1 en R2 niet noodzakelijk verschillend) met een kandidaat-sleutel CK.*
- *altijd, elke waarde van FK in R2 ofwel NULL is ofwel gelijk is aan de waarde van CK in een tuppel van R1.*

Elke component van een vreemde sleutel moet gedefinieerd zijn over hetzelfde domein als de kandidaat-sleutel waarnaar verwezen wordt. Verder mag een databank geen

vreemde sleutels bevatten die nergens naar verwijzen. Dit is de referentiële-integriteitsregel.

1.2. SQL

SQL (Structured Query Language) is de standaardtaal om met relationele databanken te werken. Standaard moet hierbij wel enigzins tussen haakjes geplaatst worden want bijna elke databank heeft zijn eigen SQL-dialect. Hoe dan ook, SQL is gestandaardiseerd door ANSI/ISO en de huidige versie is SQL-92 (ook wel SQL-2 genoemd).
In de taal werd het relationele datamodel echter niet strikt gevolgd. Zo kunnen gelijke rijen in resulterende tabellen bijvoorbeeld wel voorkomen. Ook de ondersteuning van domeinen is over het algemeen nogal beperkt.
Wanneer SQL gebruikt wordt vanuit een andere programmeertaal, zoals bijvoorbeeld C, spreekt men van embedded SQL. Embedded SQL wordt meestal geïmplementeerd met behulp van een preprocessor die de SQL-statements omzet in functie-aanroepen naar een meegeleverde bibliotheek.

1.3. ODBC

Om de vele SQL dialecten toch op een uniforme manier te kunnen benaderen heeft Microsoft ODBC (Open DataBase Connectivity) ontwikkeld. Deze C API (Application Programming Interface) is een softwarelaag die een applicatie afschermt van een specifieke databank.
De architectuur van ODBC bestaat uit een driver manager en verschillende drivers die databankafhankelijk zijn. Het zijn deze drivers die de SQL-statements van ODBC vertalen naar de native SQL van de databank. In de driver manager worden data resources geregistreerd waarbij een verbinding gelegd wordt met een bepaalde driver die gebruikt moet worden. Op die manier kan men van databank veranderen zonder ook maar iets aan het applicatieprogramma te veranderen. Het enige dat men moet doen, is in de uitwendige driver manager, de data resource naam aan een andere driver koppelen.
Er is zo ongeveer voor elke mogelijke databank een ODBC driver verkrijgbaar, onder andere dBase, IBM DB2, Informix, Ingres, Oracle, Progress, MS-SQL Server,... om er

maar enkele te noemen. Het door elkaar gebruiken van verschillende databanken en het platform-onafhankelijk zijn van applicaties is dus gemakkelijk te bereiken met ODBC.
De API van ODBC is ingedeeld in 3 levels :

- de kern (core level)
- uitbreiding niveau 1 (extension level 1)
- uitbreiding niveau 2 (extension level 2).

Elke driver moet vermelden tot op welk niveau ODBC ondersteund is. De meeste ondersteunen buiten de kern ook de functies van level 1.
De kernfuncties bevatten alle nodige faciliteiten om :

- een connectie te maken met een data resource (gekoppeld aan een databank). Vooraleer een connectie handle kan aangevraagd worden moet men een environment handle aanvragen. Binnen één environment handle zijn meerdere connecties mogelijk.
- gegevens op te vragen.
- arbitraire SQL-statements uit te voeren. De syntax van deze statements moet voldoen aan de SQL-92 standaard.

Voor het uitvoeren van SQL-statements moet men eveneens een statement handle aanvragen. Het is ook weer mogelijk om binnen één connectie meerdere statement handles te bekomen.

2. Object-oriëntatie

2.1. Het objectmodel

Het meest in het oog springende informaticabegrip van de laatste jaren is ongetwijfeld OT (Object Technology) en alles wat daarrond hangt. Nochtans zijn de ideeën achter OO (Object-Oriëntatie) al relatief oud en eigenlijk enkel logische uitbreidingen op het voorgaande structureel paradigma. Een paradigma is een verzameling van samenhangende concepten. De eerste programmeertaal die de term object introduceerde was Simula en deze dateert van 1967. Het is dan ook op het gebied van programmeren dat OO ten volle tot ontwikkeling gekomen is, hoewel het paradigma veel meer is dan enkel en alleen een manier van programmeren : OO is een manier van denken. Recent beginnen inderdaad ook andere domeinen binnen de informatica het

OO paradigma te gebruiken, zoals bijvoorbeeld software-analyse en -ontwerp maar ook databanken.

De basisgedachte van OO is eigenlijk erg eenvoudig : het verhogen van het abstractieniveau. Toch bestaat er in de literatuur veel onenigheid over de concepten die al dan niet aan de basis van OO liggen (onder andere [2] en [3]). Gebaseerd op [4] kan er toch min of meer een consensus bereikt worden door concepten in drie samenhangende groepen in te delen.

- Data-abstractie, classificatie, ADT (Abstract Data Type),...

  Men houdt zich enkel bezig met het abstractieniveau, de details waarin men geïnteresseerd is.

- Overerving (inheritance), polymorfisme, dynamische binding

  Hiermee kan een object-hiërarchie opgebouwd worden zoals die in de werkelijkheid voorkomt. Dit maakt het modelleren van de reële wereld gemakkelijker.

- Objectidentiteit, modulariteit, encapsulatie, information hiding

  Men werkt met identificeerbare zaken die tot één geheel gegroepeerd worden.

Als voordelen van object-georiënteerde software-ontwikkeling kan men onder andere vermelden :

- betere aansluiting bij de reële wereld (door een hoger abstractieniveau)
- grote herbruikbaarheid van code
- robuustere software
- betere onderhoudbaarheid (door de effecten van wijzigingen te beperken)

2.2. C++

Tussen de vele object-georiënteerde talen, zoals bijvoorbeeld Simula, Smalltalk, Eiffel en recent ook Java, neemt C++ nog altijd een dominante plaats in. Deze uitbreiding van C werd in 1985 ontwikkeld door Bjarne Stroustrup bij AT&T. Voor een volledige bespreking van de taal verwijs ik naar [5].

De taal is ook nog erg levendig en werd onlangs gestandaardiseerd (de ANSI/ISO C++ standaard). Enkele belangrijke, relatief recente, toevoegingen zijn bijvoorbeeld :

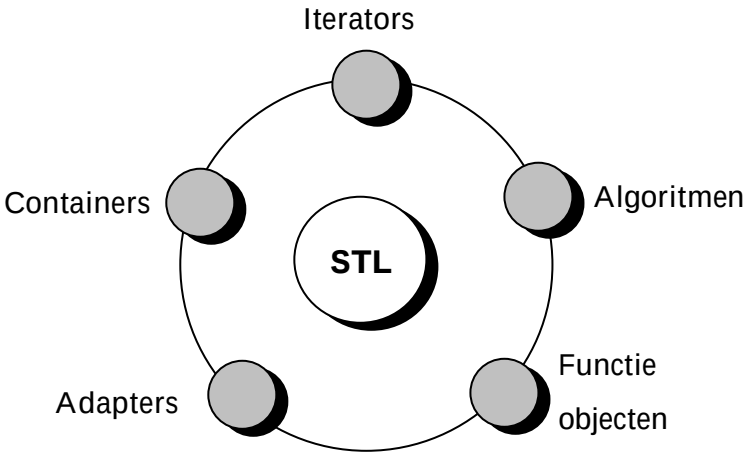
- het datatype bool (voor logische waarden)
- template functies en klassen

- exception handling (via een throw-and-catch mechanisme)
- STL (Standard Template Library)
- RTTI (Run-Time Type Information)

2.3. STL

2.3.1. Overzicht

Ongetwijfeld één van de belangrijkste toevoegingen aan de C++ standaard is STL (opgenomen in de ANSI/ISO C++ standaard op 14 juli 1994). C++ had immers nog geen standaard klassenbibliotheek, met bijvoorbeeld collectieklassen (ook wel containerklassen genoemd), zoals andere object-georiënteerde talen die wel hebben.
STL is een op templates gebaseerde bibliotheek van generieke C++ datastructuren en algoritmen die ontwikkeld werd door Alexander Stepanov en Meng Lee in de HP laboratoria. Stepanov had voordien al een dergelijke generieke bibliotheek geschreven voor ADA. Een eerdere poging om dit ook in C++ te doen mislukte omdat er in 1987 nog geen templates aanwezig waren.
De structuur van STL wordt geïllustreerd in figuur a.



Figuur A : Overzicht van STL

Verder worden deze onderdelen kort besproken. Voor een uitvoeriger behandeling van STL kan [6] geraadpleegd worden.

2.3.2. Container klassen

In de STL specificatie van 7 juli 1995 wordt een container als volgt gedefinieerd :

Containers are objects that store other objects.

Hoewel het gebruik van templates voor het implementeren van containerklassen niet echt noodzakelijk is, biedt deze methode toch wel belangrijke voordelen ten opzichte van de andere technieken. Dit wordt duidelijk wanneer de verschillende methoden met elkaar worden vergeleken :

- collectieklassen met void pointers (in C) :
  * alle bewaarde variabelen moeten gealloceerd worden op de heap : inefficiënt.
  * pointer casting : onelegant, inefficiënt en verlies van typecontrole
- collectieklassen gebaseerd op afleiding :
  * voor elk datatype dat moet bewaard worden, moet een klasse afgeleid worden van de basisklasse.
  * virtuele functies : inefficiënt (vanwege de dynamische typecontrole)
- collectieklassen gebaseerd op templates :
  * geen bijkomende klassen nodig
  * ingebouwde datatypes kunnen ook gebruikt worden
  * volledig statische typecontrole

STL bevat twee aparte families van containers :

- Sequentie containers : vector<T>, deque<T>, list<T>
  → Dit zijn de klassieke datastructuren.
- Associatieve containers : set<Key>, multiset<Key>, map<Key, T>, multimap<Key, T>
  → Bewaren de objecten op basis van een sleutelwaarde.

De STL specificatie geeft ook aanwijzingen voor het gebruik van de sequentie containers. Deze richtlijnen worden samengevat in tabel b.

vector	Moet als default gebruikt worden
deque	Moet gebruikt worden als de meeste objecten in het begin of op het einde toegevoegd of gewist worden.
list	Moet gebruikt worden als er vaak objecten in het midden toegevoegd of gewist worden.

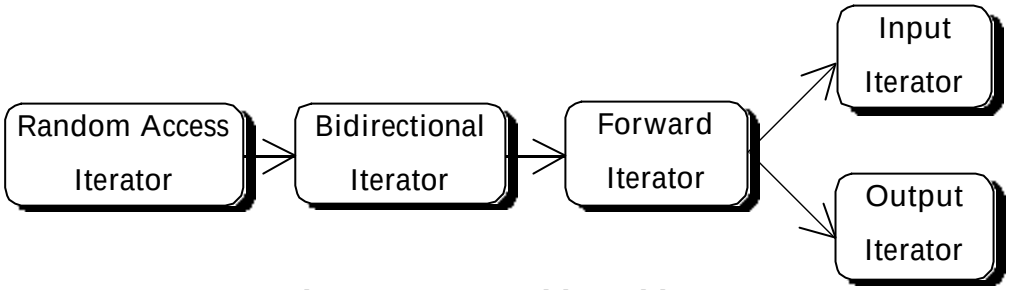
Tabel B : Gebruik van de sequentie containers

Typisch voor alle STL-containers is de zo uniform mogelijke interface. Zo beschikt elke container bijvoorbeeld over een functie insert en erase voor het invoegen en verwijderen van objecten.

Het gebruik van deze containers stelt echter wel een aantal eisen aan de objecten die erin bewaard moeten worden. Minimale vereisten zijn :

- publieke default constructor
- publieke kopie constructor
- publieke assignatie operator
- publieke destructor

2.3.3. Iteratoren



Figuur B : Iterator-hiërarchie

Bij elke container hoort er een iteratorklasse. Een iterator kan gezien worden als een veralgemening van een pointer en wordt gebruikt voor het navigeren door de containers. Het is zelfs mogelijk om gewone pointers te gebruiken waar een iterator vereist is (deze doorlopen dan de array container). Aangezien niet elke container even flexibel te doorlopen valt, zijn niet alle iteratoren gelijkwaardig op gebied van functionaliteit, met andere woorden er is een iterator-hiërarchie nodig. Deze wordt voorgesteld in figuur b. De linkse iterator voldoet daarbij steeds aan de eisen van de rechtse iterator.

Elke STL-container beschikt over de functies begin() en end() die geschikte iteratoren retourneren waarmee de volledige container kan doorlopen worden.

2.3.4. Algoritmen

De algoritmen van STL zijn in feite gewone template functies en hebben geen weet van containers. De toegang tot een bepaalde container wordt hen via iteratoren verschaft. Op die manier worden de datastructuren (containers) en algoritmen volledig van elkaar

losgekoppeld, wat betekent dat de algoritmen niet voor elke container apart moeten geprogrammeerd worden. Sommige algoritmen kunnen echter wel een bepaalde iterator categorie vereisen, bijvoorbeeld het sorteeralgoritme vereist een random-access iterator.

2.3.5. Functie-objecten

Een functie-object is een veralgemening van een functie pointer en bevat enkel en alleen de operator()(). Functie-objecten worden bijvoorbeeld gebruikt om operaties uit te voeren op container objecten. STL heeft standaard een aantal rekenkundige, logische en vergelijkingsfunctie-objecten ingebouwd.

2.3.6. Adapters

Er zijn twee soorten adapters :

- container adapters : definiëren een andere interface op een bestaande container, bijvoorbeeld stack<Container>, queue<Container> en priority_queue<Container>.
- function adapters : creëren nieuwe functie-objecten van bestaande functies, bijvoorbeeld negators (inverteren van een functie-object), binders (binden een argument van een functie-object aan een vaste waarde).

2.4. RTTI

Met RTTI is het mogelijk om tijdens de uitvoering van het programma de klasse waartoe een object behoort te bepalen. Dit kan handig zijn om bijvoorbeeld een gespecialiseerd algoritme te gebruiken in het geval het object tot een welbepaalde klasse behoort en een algemener, minder efficiënt algoritme in de andere gevallen.
Stroustrup ([5]) schrijft over RTTI :

> *Run-time type information has many uses including support for object I/O, persistent objects and object-oriented databases, and debugging. However it has also an enormous potential for misuse.*

Men zou inderdaad RTTI kunnen gebruiken op plaatsen waar men eigenlijk virtuele functies zou moeten gebruiken, waardoor alle voordelen van OO verloren gaan.

3. Object-georiënteerde databanken

Aangezien entiteiten uit de reële wereld niet altijd even gemakkelijk kunnen gemodelleerd worden in een relationele tabel (bijvoorbeeld beelden, spraak,...) is het evident dat men ook in databanken OT wil gebruiken, zodat de gegevens een complexere structuur kunnen hebben. Voor het incorporeren van OO capaciteiten in databanken zijn er drie methoden mogelijk :

- een compleet nieuw systeem ontwerpen.

  Deze aanpak is gestandaardiseerd als de Object Database Standard ODMG-93 (Object Database Management Group). Hierbij wordt eveneens een binding met C++ en Smalltalk gedefinieerd.

- een RDMBS uitbreiden met OO mogelijkheden :

  Op dit gebied is men aan het werken aan de nieuwe SQL3 standaard die inderdaad over OO constructies zal beschikken. Het is trouwens zo dat er nog maar weinig echt puur relationele databanken op de markt zijn.

- een OO programmeertaal uitbreiden met databank mogelijkheden :

  Dit gebeurt dan met bibliotheken die de mogelijkheid bieden om objecten persistent te maken, d.w.z. dat de levensduur langer is dan de uitvoeringstijd van het programma.

Huidige OODBMS kunnen echter vanwege de slechte performantie de RDBMS nog niet volledig vervangen en worden daarom uitsluitend gebruikt waar een relationele databank niet voor geschikt is, zoals bijvoorbeeld CAD/CAM en multimedia-toepassingen.

Hoofdstuk 2 : Doelstelling

No one knows what power lies yet undevelopped in that wiry system of mine.
Ada Lovelace

1. Relationele databanken versus OO

Uit de inleiding blijkt dat er momenteel verschillende modellen in gebruik zijn :

- In de databankwereld : het relationeel model
- In de software-ontwikkeling : het objectmodel

Wanneer een object-georiënteerde applicatie gebruik wil maken van een relationele databank moeten beide modellen dus gecombineerd worden. Dit levert echter een aantal problemen wat men traditioneel de impedance mismatch noemt. Men schrijft in [7] :

Unless you have the right tools, mixing objects and tables is like mixing oil and water.

De impedance mismatch wordt veroorzaakt door de architecturale verschillen tussen de beide modellen. Deze verschillen zijn samengevat in tabel c.

	Relationeel model	Object-georiënteerd model
datastructuur	twee-dimensioneel	complexe groepering
applicatie logica	losgekoppeld van de data	gekoppeld met de data
verwerkingsniveau	per verzameling rijen	per object

Tabel C : Architecturale verschillen

Er is dus duidelijk een semantische kloof die moet overbrugd worden.

2. Traditionele aanpak

Een eerste mogelijke aanpak van de impedance mismatch is het gebruik van traditionele, procedurale API's, althans voor wat de databank-toegang betreft. Hierbij

worden echter veel voordelen van OO teniet gedaan, onder andere data-abstractie en encapsulatie. Dit betekent immers dat wijzigingen aan het databank model ook wijzigingen aan de applicatiecode veroorzaken, wat uiteraard niet gewenst is.
Er kan onderscheid gemaakt worden tussen :

- Embedded SQL

  Hierbij worden SQL-statements hard gecodeerd in de applicatiecode. Dit is uiteraard niet flexibel en brengt ook de typische nadelen van embedded SQL met zich mee zoals bijvoorbeeld :

  * niet gebruiksvriendelijk (bijvoorbeeld het overbrengen van gegevens naar de applicatie)
  * preprocessor : hierdoor gaat alle typecontrole verloren
  * gebonden aan één databankleverancier

- C API's, zoals bijvoorbeeld ODBC

  Ook deze methode biedt een te laag abstractieniveau en is niet volkomen type-veilig, onder andere door het gebruik van void pointers. ODBC is echter wel al databank-onafhankelijk.

Kortom, het is niet verstandig om de impedance mismatch op te lossen door een stap achteruit te zetten en terug proceduraal te gaan programmeren.

3. Bestaande OO bibliotheken

Een andere mogelijke oplossing is ervoor te zorgen dat het relationele model zich gedraagt of tenminste benaderd kan worden alsof het een objectmodel zou zijn. Dit kan worden bereikt door bovenop de RDBMS een laag te voorzien die een mapping uitvoert van objecten naar tabellen (en omgekeerd). Op die manier kan men met de relationele databank communiceren alsof het virtueel persistente objecten zijn. Zo'n mappinglaag wordt soms een impedance mismatch resolver genoemd, omdat de impedance mismatch ermee opgelost wordt.
Voor een uitwerking van zo'n bibliotheek kan verwezen worden naar [8].
Een nadeel aan deze methode is het - nog altijd - vrij lage abstractieniveau. Men blijft immers geconfronteerd worden met klassieke relationele termen zoals kolommen, rijen,... met dit verschil dat deze zich nu voordoen als objecten.

Bovendien maken dergelijke bibliotheken veelvuldig gebruik van overerving wat uiteraard niet zo goed is voor de efficiëntie. Het is zelfs zo dat een object moet afgeleid zijn van een bepaalde klasse, vooraleer het kan bewaard worden in een databank. Op die manier ontstaat echter een onrealistische modellering van de werkelijkheid, want er is in wezen geen verschil tussen bijvoorbeeld een Persoon object dat persistent is en een Persoon object dat dit niet is. Het al dan niet persistent zijn is een vorm van gebruik van een object en geen essentieel kenmerk van het object zelf.
Het spreekt ook vanzelf dat deze extra mappinglaag tijd opeist en dus de performantie van de relationele databank nadeling beïnvloedt.
Toch bestaan er talrijke commerciële bibliotheken die dit principe toepassen, zoals bijvoorbeeld :

- Persistence (Persistence Software Inc.)
- Enterprise Object Framework (NeXT Software Inc.)
- DBTools.h++ (Rogue Wave Software Inc.)
- Zinc DataConnect (Zinc Software Inc.)

4. Een nieuwe aanpak

In het kader van deze thesis is het de bedoeling een OO bibliotheek te ontwikkelen die het abstractieniveau wel voldoende verhoogt zodat de gebruiker zich niet meer moet bezighouden met kolommen en rijen. De gebruiker mag enkel zijn businessobjecten zien, wat impliceert dat het expliciet gebruik van eerder onnatuurlijke primaire sleutels vermeden wordt. Deze onnatuurlijke kolommen (gewoonlijk een soort rangnummer) worden vaak in een tabel toegevoegd om entiteiten te kunnen identificeren, hoewel ze in feite geen semantische betekenis hebben. In OO zijn dergelijke attributen ongewenst omdat we de werkelijkheid zo getrouw mogelijk willen modelleren, en onnodig omdat elk object immers door zijn bestaan al een unieke identiteit heeft.
Het zou bovendien moeten zo zijn dat er niets maar dan ook absoluut niets aan de businessobjecten gewijzigd moet worden vooraleer ze persistent gemaakt kunnen worden. Om dit te bereiken dringt zich het gebruik van templates op. Templates bieden tevens het voordeel van statische typecontrole wat de efficiëntie ten goede komt. Bovendien ondersteunen templates ook de ingebouwde datatypes zodat er geen klassen

nodig zijn die enkel deze datatypes als het ware inpakken. Dit is vaak het geval bij de bestaande OO bibliotheken.

Gezien STL, sinds kort, een officieel onderdeel is van de ANSI/ISO C++ standaard zal van bibliotheken ook meer en meer geëist worden dat ze er compatibel mee zijn en er samen mee gebruikt kunnen worden. Hierbij stelt zich het idee om de databank op te vatten als een persistente STL-container. Dit betekent dat er naar een uniforme interface en analoge gebruiksmogelijkheden gestreefd zal worden, bijvoorbeeld door te voorzien in een iteratorklasse. Hierdoor zal het ook mogelijk zijn om de STL-algoritmen te gebruiken.

Tevens biedt het ontwikkelen van deze bibliotheek de mogelijkheid om de bruikbaarheid van STL en de andere recente toevoegingen aan de C++ standaard na te gaan.

Samengevat kunnen de doelstellingen als volgt geformuleerd worden :

- verhogen van het abstractieniveau bij het werken met relationele databanken. De gebruiker wil enkel met businessobjecten werken.
- vermijden van het gebruik van onnatuurlijke primaire sleutels.
- mogelijkheid om businessobjecten ongewijzigd persistent te maken.
- compatibel zijn met STL
- de bruikbaarheid van STL nagaan.

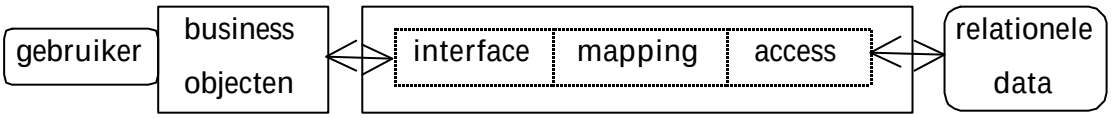
Als de technologie van object-georiënteerde databanken verbetert en ze erin slagen om de RDBMS overal te verdringen, dan zullen de functies van deze bibliotheek zeker in de één of andere vorm aanwezig moeten zijn in het OODBMS zelf.

Hoofdstuk 3 : Ontwerp

First think, then program.
Dijkstra

1. Systeemontwerp

De begrippen en notaties van de figuren die in dit hoofdstuk gehanteerd worden zijn afkomstig van OMT (Object Modeling Technique), zie [9].
Een eerste taak van het systeemontwerp is het identificeren van de verschillende subsystemen. Deze worden getoond in figuur c.

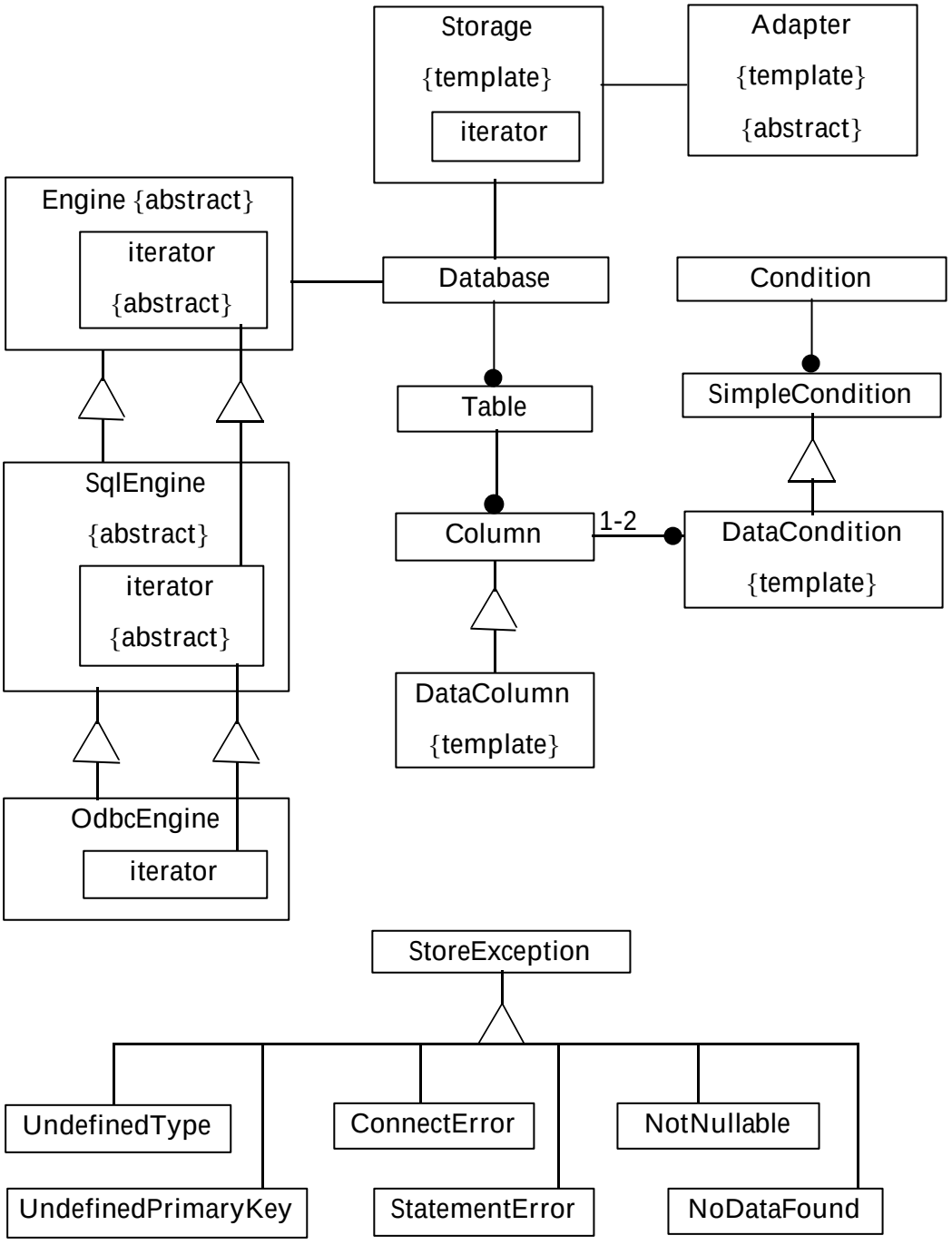


Figuur C : Subsystemen

Het is hierbij belangrijk op te merken dat de interface rechtstreeks gebruik maakt van de businessobjecten. Deze ondergaan daarna een mapping naar het relationele datamodel zodat ze zonder problemen kunnen bewaard worden in een RDBMS.
Als toegangsmethode tot de relationele databank wordt in deze thesis gekozen voor ODBC. Op die manier worden immers meteen een uitgebreid aantal, verschillende databanken ondersteund. Het spreekt voor zich dat deze procedurale API enkel intern zal gebruikt worden. Het versienummer van de ODBC driver manager die gehanteerd wordt is 2.10.
De implementatie van de bibliotheek zal gebeuren in C++, meerbepaald met de Borland C++ 5.0 compiler onder het Windows-95 besturingssysteem. Als hardware wordt een Pentium PC met 32 MB RAM gebruikt.

2. Objectmodel

Het volledige objectmodel van de bibliotheek wordt getoond figuur d.



Figuur D : Objectmodel

De subsystemen die in het systeemontwerp onderscheiden werden corresponderen met de volgende klassen :

- user-interface : `Storage, Condition`
- mapping :

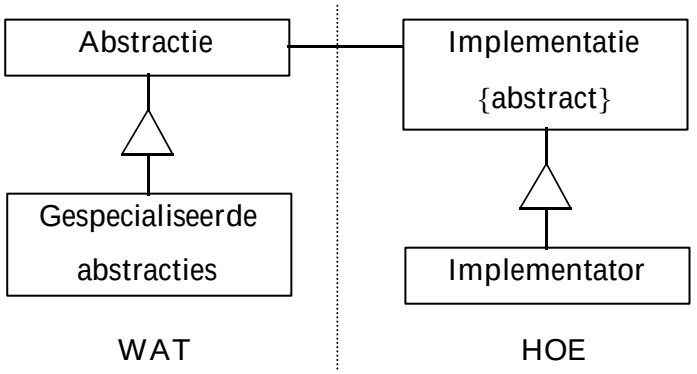
  * databank schema : `Database, Table, Column, DataColumn`

  * object-tabel : `Adapter`

  * voorwaarden : `SimpleCondition, DataCondition`
- access : `Engine, SqlEngine, OdbcEngine`

3. Platform-onafhankelijkheid

De `Engine` hiërarchie is nodig voor de platform-onafhankelijkheid van de bibliotheek. Dit wordt bereikt met het zogenaamde bridge design pattern (zie figuur e en [10]).



Figuur E : Bridge design pattern

Dit ontwerppatroon is gebaseerd op overerving en laat toe een abstractie op verschillende manieren te implementeren. De gekozen implementatie wordt dan geassocieerd met de abstractie en aangezien de implementaties afgeleid zijn van een abstracte klasse die de interface vastlegt, kan gemakkelijk van implementatie veranderd worden.

De abstractie (`Database`) is hier niet verder gespecialiseerd. `Engine` vervult de rol van abstracte interface klasse. Als voorbeeld van een driver zal in deze thesis de `OdbcEngine` geïmplementeerd worden. Indien men een databank in zijn native taal wenst aan te spreken (bijvoorbeeld uit efficiëntie-overwegingen), hoeft men enkel een

klasse, afgeleid van `Engine`, te programmeren die deze interface op een andere manier invult.

4. Tabel versus object

In de literatuur bestaat er veel discussie omtrent het feit of een klasse nu de tegenhanger is van een tabel of een domein. Date schrijft in [1] :

> *In the previous section, we equated object classes and domains. Many products and prototypes, however, are equating object classes and relations instead. In this section we argue that this latter equation is a serious mistake.*

De kern van de discussie wordt hier kort geïllustreerd. Wanneer we in een RDBMS een persoon wensen te bewaren dan kan deze tabel bijvoorbeeld bestaan uit vier kolommen : Voornaam, Naam, StraatNaam en StraatNr. Dit schema wordt getoond in tabel d. Een OODBMS zal echter enkel objecten van de klasse Persoon bevatten. Het feit dat een dergelijk schema niet mogelijk is in een RDBMS is een gevolg van de gebrekkige ondersteuning van domeinen door de producten die momenteel op de markt zijn.

Voornaam	Naam	StraatNaam	StraatNr
Jan	Jansens	Woningstraat	3
...	...	...	...

Tabel D : Personen in een RDBMS

De hele discussie draait er nu om of tabel d op zich kan beschouwd worden als een klasse, met als rijen de objecten die tot die klasse behoren, of als de domeinen moeten gegroepeerd worden tot een nieuw Persoon domein (= klasse), zoals in tabel e.

Persoon
jjansens
...

Tabel E : Het Persoon domein in een RDBMS

De meeste commerciële, object-georiënteerde databanken volgen de eerste redenering. Aangezien het momenteel in de praktijk dus onmogelijk is om in een RDBMS zelf complexe domeinen, zoals bijvoorbeeld een Persoon, te definiëren, zal er hiermee ook

geen rekening gehouden worden. Van domeinen wordt verondersteld dat ze eenvoudige, ingebouwde datatypes zijn.
De bibliotheek die hier ontwikkeld wordt, laat verschillende mogelijkheden toe voor het mappen van objecten in tabellen :

- 1 object in 1 tabel (of een deel ervan)
- 1 object in meerdere tabellen
- meerdere objecten in 1 tabel
- meerdere objecten in meerdere tabellen

5. Klasse verantwoordelijkheden

5.1. Database

De `Database` klasse vervult eigenlijk twee functies :

- Enerzijds bevat deze klasse de tabellen die moeten gebruikt worden bij het bewaren van een businessobject. Zoals eerder vermeld kunnen dat er meer dan één zijn.
- Anderzijds is er hiermee een `Engine` object geassocieerd zodat de `Database` weet wie met de relationele tabellen kan communiceren.

Per klasse waarvan er objecten persistent moeten kunnen gemaakt worden, moet men een `Database` object maken dat enkel het deel van het databankschema bevat dat relevant is voor die klasse. Er zijn dus meerdere `Database` objecten per fysische databank mogelijk.

5.2. Table

Een `Table` object correspondeert met precies één relationele tabel en is eigenlijk niets meer dan een verzameling van kolommen met een naam.

5.3. Column

De klasse `Column` is nodig om het mogelijk te maken dat de klasse `Table` een lijst bevat van kolommen over verschillende domeinen (zie `DataColumn`). Hierin worden gegevens die voor elke kolom, ongeacht het domein, belangrijk zijn bijgehouden :

- de naam van de kolom
- de mogelijkheid om deze kolom NULL te maken en of dit momenteel zo is.

- het al dan niet tot de primaire sleutel behoren van deze kolom
- de gerefereerde kolom indien het een vreemde sleutel is

5.4. DataColumn

Een `DataColumn` is een concrete instantie van een `Column` en maakt in feite de mapping tussen een databank domein en een C++ datatype mogelijk. Dit datatype moet aan deze klasse meegegeven worden via een templateargument zodat statische typecontrole gegarandeerd wordt. `DataColumn` bevat dan ook een member variabele van dit type die fungeert als een soort buffer tussen het businessobject en een kolom. Wanneer de databank of een businessobject gelezen wordt, dan worden de `DataColumn` objecten met de gelezen waarden opgevuld.

Om deze buffervariabelen te kunnen instellen of opvragen, stelt zich echter het probleem dat deze `Column` objecten (die zich in `Table` bevinden) eerst geconverteerd moeten worden naar `DataColumn` objecten. Dit komt omdat deze functies niet aanwezig zijn in de `Column` klasse. Ze bevatten immers een parameter (of een retourneerwaarde) van het templateargument type dat enkel gekend is in de klasse `DataColumn`.

De sleutel tot de oplossing van dit probleem is RTTI. Dit betekent echter dat de mogelijke datatypes die als templateargument gebruikt kunnen worden op voorhand bekend moeten zijn. De C++ datatypes die men zeker moet ondersteunen zijn deze die corresponderen met de mogelijke databankdomeinen.

In deze bibliotheek worden, door de `Engine` klassen, de volgende datatypes ondersteund : bool, short, integer, long, float, double en string. Het spreekt voor zich dat een commerciële versie deze lijst zou moeten uitbreiden met bijvoorbeeld telkens de unsigned versies, datum en tijd types,... Om deze redenen zal er bij de implementatie voor gezorgd worden dat het gebruik van RTTI tot het strikte minimum beperkt blijft zodat uitbreiding geen groot probleem is. Verder zal er ook voor gezorgd worden dat deze uitbreiding kan gebeuren door het, in een afgeleide klasse (zie de `Engine` hiërarchie), opnieuw implementeren van deze functies.

5.5. Adapter

Wanneer een businessobject persistent moet gemaakt worden, moeten de `DataColumn` objecten uit de `Database` dus opgevuld worden met de corresponderende member variabelen van het object.
Deze mapping (alsook de omgekeerde) gebeurt in klassen die afgeleid zijn van de abstracte template klasse `Adapter`, die de interface voor het lezen en schrijven van de businessobjecten vastlegt. Het zijn dus eigenlijk deze afgeleide klassen die de impedance mismatch oplossen (de feitelijke impedance mismatch resolvers), door per klasse aan te geven welke member variabele in welke kolom moet komen. Deze werkwijze laat eveneens de mogelijkheid om samengestelde C++ datastructuren over meerdere kolommen in de databank te splitsen, het hoeft dus zeker geen één-op-één mapping te zijn. De `Adapter` kan bijvoorbeeld één C++ Datum object in drie aparte kolommen (dag, maand en jaar) bewaren, indien de databank geen datum domein kent.
Het templateargument van de `Adapter` klasse is de klasse van de persistente objecten.

5.6. Engine

De `Engine` klasse is eigenlijk de echte toegangspoort tot één welbepaalde relationele databank. Dit betekent dat als men van twee klassen objecten wil bewaren in eenzelfde databank, men daarvoor hetzelfde `Engine` object kan gebruiken. Merk op dat indien deze klassen een verschillend deel van het volledige databankschema gebruiken, men wel twee verschillende `Database` objecten zal nodig hebben.
Door klassen van `Engine` af te leiden die de volledige abstracte interface implementeren kan men aparte databank drivers (= engines) schrijven die de databank in zijn eigen taal, zo efficiënt mogelijk aanspreken.
De geneste, eveneens abstracte, iteratorklasse is de tegenhanger van het SQL-cursor concept.
Het is in deze `Engine` hiërarchie dat de functie die van RTTI gebruik zullen maken, zoals dit eerder beschreven werd, zich zullen bevinden.

5.7. SqlEngine

Aangezien de verschillende databanken vooral verschillend zijn op gebied van initialisatie (het maken van een connectie), ophalen van gegevens uit de databank,

opkuisen (het verbreken van een connectie),... maar toch min of meer dezelfde basis SQL-statements ondersteunen (vooral deze die in deze bibliotheek gebruikt worden), leek het verstandig om de opbouw van deze statements in een aparte klasse onder te brengen.
Indien men een driver wil schrijven voor een bepaalde databank die deze basis SQL-statements begrijpt, dan kan deze nieuwe engine klasse onmiddellijk van `SqlEngine` afgeleid worden en de aanwezige functies herbruiken.

5.8. OdbcEngine

Een voorbeeld van een driver die van `SqlEngine` gebruik maakt is `OdbcEngine`. Enkel in deze klasse worden de ODBC-functies gebruikt, nergens anders. Het is dan ook in deze klasse dat de verschillende ODBC-handles bijgehouden worden.
De geneste iteratorklasse staat in voor het beheer van de SQL-cursors (die uitgevoerd worden door ODBC).

5.9. SimpleCondition

Deze klasse is voor `DataCondition` wat `Column` is voor `DataColumn`. Op die manier is het mogelijk om een lijst bij te houden van condities over verschillende domeinen (de klasse `Condition`).
In `SimpleCondition` worden domeinonafhankelijke variabelen bijgehouden :

- de vergelijkingsoperator
- of het een vergelijking van een kolom met NULL betreft

In tegenstelling tot de `Column` klasse heeft een object van de klasse `SimpleCondition` op zich ook een betekenis. Zo'n object zal immers gebruikt worden om verschillende condities met elkaar te verbinden. De vergelijkingsoperator wordt dan gezien als een logische operator (AND, OR of NOT) .

5.10. DataCondition

Deze templateklasse, die afgeleid is van `SimpleCondition`, dient net zoals `DataColumn` voor het mappen van het domein op een C++ datatype. Ook hier wordt een variabele van dit type bijgehouden. Het is immers niet mogelijk hiervoor de `DataColumn` te gebruiken, omdat een kolom meerdere keren gebruikt kan worden in

één conditie en er dus meerdere waarden per kolom bijgehouden moeten kunnen worden.
Bovenstaande redenering heeft als gevolg dat voor het lezen van een `DataCondition` object opnieuw RTTI zal moeten toegepast worden.
Verder bevat deze klasse nog twee associaties naar `DataColumn` objecten. Men kan immers niet alleen een kolom vergelijken met een waarde maar ook met een andere kolom.
De `DataCondition` objecten zullen worden aangemaakt door de vergelijkingsoperatoren (==, !=, <, >,...) die inwerken op de `DataColumn` objecten. Op deze manier kan ervoor gezorgd worden dat enkel zinvolle vergelijkingen mogelijk zijn. Hiermee wordt bedoeld dat enkel vergelijkingen over hetzelfde datatype (en dus hetzelfde domein) geformuleerd kunnen worden, zoals dit trouwens ook in relationele databanken het geval is. Dit geldt zowel voor de vergelijkingen tussen een kolom en een waarde als voor een vergelijking tussen twee kolommen. Bovendien gebeurt deze typecontrole volledig statisch.

5.11. Condition

In een `Condition` object worden de `DataCondition` objecten bijgehouden die samen één grote conditie vormen. Deze worden door gewone `SimpleCondition` objecten, die enkel een logische operator bevatten, aaneengeschakeld. Deze lijst wordt in postfix-notatie bewaard zodat er geen behoefte is om haakjes te gebruiken.
Aangezien elk `DataCondition` object onmiddellijk geconverteerd wordt naar een `Condition` object (bestaande uit één enkele voorwaarde) is dit de enige conditie klasse waarmee de gebruiker in contact komt. Hij kan zo'n objecten aanmaken, kopiëren en verbinden met de gebruikelijke logische operatoren van C++ (&&, | | en !).

5.12. Storage

Deze klasse vormt het hart van de bibliotheek. Het is aan `Storage` dat de gebruiker de opdrachten geeft om met de objecten transacties te verrichten. De klasse van deze objecten moet als templateargument opgegeven worden. Volgende functies moeten zeker ondersteund worden :

- op basis van de primaire sleutel (hierbij gaat het dus altijd om één enkel object) :
  * toevoegen van een businessobject

* wijzigen
* wissen
* opvragen

- op basis van een willekeurige conditie : opvragen van (meerdere) objecten

Teneinde deze functionaliteiten te kunnen aanbieden bevat deze klasse :

- De `Adapter` die de mapping uitvoert en over hetzelfde templateargument gedefinieerd is.
- Een `Database` object met het (deel van het) databankschema dat nodig is voor het werken met deze businessobjecten en de toegang tot de databank.

Om de objecten die aan een bepaalde voorwaarde voldoen te kunnen opvragen, wordt ook hier een geneste iteratorklasse voorzien. De `Storage` iterator zal tot de STL input iterator-categorie behoren. Dit is de iteratorsoort die bijvoorbeeld gebruikt wordt bij het lezen van een tekstbestand. Dit betekent dat volgende functionaliteiten zullen moeten voorzien worden :

- het aanmaken van een iterator

  Dit zal gebeuren door een conditionele zoekfunctie in de klasse `Storage` zelf. Aangezien er naar uniformiteit met STL gestreefd wordt, zal `Storage` ook de functies `begin()` en `end()` moeten bezitten die eveneens (speciale) iteratoren aanmaken.
- kopie constructor
- assignatie operator
- derefentie operator

  Merk op dat deze operator geen lvalue hoeft te retourneren. Dit betekent dat het niet mogelijk is om gegevens in de databank te schrijven door gebruik te maken van een iterator. Dit heeft zin gezien de vergelijking tussen een databank en een tekstbestand waaruit gelezen wordt (normaal kan er dan ook niet terug naar geschreven worden).
- increment operator
- vergelijkingsoperatoren (== en !=)

Terzijde kan men opmerken dat het uit bovenstaande opsomming duidelijk is dat de vergelijking tussen iteratoren en gewone C++ pointers zeker niet volledig opgaat voor dergelijke input iteratoren. Het zijn in feite enkel de random-access iteratoren die de

vergelijking wel volledig doorstaan (en bijvoorbeeld ook pointer rekenkunde ondersteunen). Telkens ligt echter wel hetzelfde principe aan de basis : de objecten in de container één voor één doorlopen.

5.13. StoreException

Dit is de basisklasse van de gehele exceptie-hiërarchie. Op die manier zal het mogelijk zijn om met één catch-instructie alle fouten die in de `Storage` bibliotheek opgeworpen worden op te vangen.
Er werd een onderscheid gemaakt tussen volgende excepties :

- `UndefinedType`

  Deze fout wordt opgeworpen wanneer een domein gebruikt wordt die niet door de bibliotheek ondersteund is. Zoals eerder aangehaald zal RTTI nodig zijn om de waarde uit `DataColumn` en `DataCondition` objecten te halen. Dit is de fout die men krijgt als RTTI het type niet kon terugvinden.

- `UndefinedPrimaryKey`

  Aangezien het relationele datamodel sleutelwaarden gebruikt om onderscheid te maken tussen verschillende rijen (en dus een identiteit aan een waarde koppelt), moet een object dat gemapt wordt in een tabel ook een voor dit object uniek attribuut bevatten. Zoals in de doelstellingen die we vooropgesteld hebben, zullen deze onnatuurlijke sleutelwaarden volledig verborgen kunnen worden.

- `ConnectError`

  Deze exceptie verkrijgt men als de driver er niet in slaagde een connectie te leggen met de databank. In het geval van de ODBC driver kan dit bijvoorbeeld zijn omdat de data resource name niet gevonden wordt in de ODBC driver manager.

- `StatementError`

  Dit betekent dat er een fout gebeurde tijdens het uitvoeren van een SQL-statement. Merk op dat een integriteitswaarschuwing (bijvoorbeeld de rij die in een tabel ingevoerd wordt bestaat al en kan dus niet meer toegevoegd worden, tenminste als dubbele rijen niet toegelaten zijn) geen aanleiding geeft tot deze exceptie. Wanneer excepties in C++ niet opgevangen worden, stopt de uitvoering. Dit zou dus betekenen dat er telkens gecontroleerd zou moeten worden of deze waarschuwing al dan niet optrad. Dit is uiteraard onaanvaardbaar en daarom zal

deze integriteitswaarschuwing aangegeven worden met een bepaalde returnwaarde, zoals dit gebruikelijk is in gewone C. Deze situatie is een gevolg van de inflexibele exception handling van C++.

- `NotNullable`

  Wanneer er geprobeerd wordt een kolom die als not-nullable gedefinieerd werd, toch op NULL te zetten, wordt deze exceptie opgeworpen.

- `NoDataFound`

  Deze exceptie treedt op wanneer een zoekoperatie (of een iterator) geen gegevens kon vinden.

Hoofdstuk 4 : Implementatie

See simplicity in the complicated.
Lao Tzu

1. Naam conventies

Bij het programmeren in C++ werden volgende conventies gevolgd :

- alle eigen klassen beginnen met een hoofdletter
- woorden binnen een naam beginnen ook met een hoofdletter
- alle member variabelen (of instantie variabelen) beginnen met i
- alle globale veranderlijken beginnen met een g
- alle klasse variabelen (variabelen die static zijn) beginnen met een c

2. Het databankschema

2.1. Definitie

Een belangrijk deel van de initialisatie is het definiëren van het databankschema dat door een bepaalde klasse gebruikt wordt. Hierbij moeten volgende constructors, in deze volgorde, gebruikt worden :

- de gebruikte `Engine` constructor
- de `Database` constructor :

  ```
  Database (string dbName, Engine *engine, bool autoCommitMode
  = true);
  ```

- de `Table` constructor : `Table (Database& db, string tableName);`
- de `DataColumn` constructor :

  ```
  DataColumn<T> (Table& table, string colName, bool primaryKey
  = false, Column *foreignKey = 0, bool nullable = false);
  ```

waarbij T het C++ datatype is dat overeenkomt met het domein van de kolom.

Bovenstaande constructors zorgen ervoor dat de lijsten (in `Database` en `Table`) correct opgebouwd worden.

De objecten die het databankschema definiëren worden globaal verondersteld omdat deze ook gebruikt kunnen worden voor het opleggen van condities.

Net zoals in een RDBMS kan een kolom die als primaire sleutel (of toch een deel daarvan) opgegeven wordt, niet nullable zijn. De bibliotheek zal dit niet zelf corrigeren maar zal de gebruiker daarop wijzen door de `NotNullable` exceptie op te werpen. Dit kan immers een indicatie zijn dat er nog andere fouten in het databankschema zitten, fouten die misschien niet zo gemakkelijk automatisch gedetecteerd kunnen worden.

Het is duidelijk dat bovenstaande informatie allemaal opgeslagen is in de systeemtabellen van de relationele databank zodat deze definities grotendeels automatisch gegenereerd kunnen worden met behulp van een script.

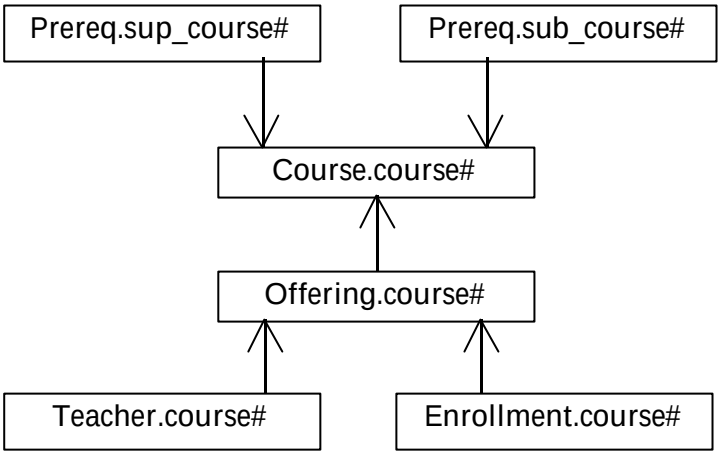
Voor de volledigheid moet hier ook nog vermeld worden dat het niet altijd strikt noodzakelijk is om een primaire sleutel uit de databank ook zo te definiëren in het C++ `DataColumn` object. De praktische gevolgen van deze vereenvoudiging en tevens ook de beperkingen zullen later met een voorbeeld geïllustreerd worden.

2.2. Vreemde sleutels

Zoals uiteengezet kunnen vreemde sleutels meegegeven worden in de constructor van een kolom. Om echter altijd alle kolommen per tabel na elkaar te kunnen plaatsen (voor de overzichtelijkheid) en toch alle mogelijkheden voor het leggen van referenties (in het bijzonder bij zelf-refererende tabellen) te bewaren, werd ook een functie voorzien waarmee vreemde sleutels kunnen gedefinieerd worden. Dit kan dan bijvoorbeeld gebeuren in de constructor van de afgeleide `Adapter` klasse.

Aan de hand van de vreemde sleutels kan er een hele cyclus van referenties ontstaan. Het spreekt vanzelf dat bij het lezen (en schrijven) van objecten deze hele cyclus telkens dezelfde waarde zal hebben. Om de programmeur niet te verplichten om telkens al deze kolommen in te stellen (en fouten die hierbij kunnen ontstaan te vermijden) krijgen deze kolommen automatisch dezelfde waarde. Om dit te kunnen bereiken moet niet alleen bijgehouden worden of een kolom een vreemde sleutel is, maar ook van welke kolommen een bepaalde kolom een vreemde sleutel is (dit kunnen er meerdere zijn en

vormen een lijst van refererende kolommen). Het algoritme maakt gebruik van twee functies en kan het best uitgelegd worden met een voorbeeld (zie [1]).
Veronderstel een referentiecyclus zoals in figuur f, waarbij een pijltje van de vreemde sleutel kolom naar de gerefereerde kolom wijst. Hierin is Offering.course# dus een vreemde sleutel naar Course.course# en heeft Course.course# drie refererende kolommen, namelijk Offering.course#, Prereq.sup_course# en Prereq.sub_course#.



Figuur F : Vreemde sleutel cyclus

Wanneer de waarde van een kolom met de functie `setData(const T&)` opgevuld wordt, wordt er gecontroleerd of deze kolom een vreemde sleutel is. Indien dit zo is wordt dezelfde functie van deze kolom opgeroepen. In het andere geval wordt de waarde van de kolom ingesteld en voor elke kolom in de lijst van refererende kolommen de functie `setKeys(const T&)` opgeroepen. Deze functie stelt de kolomwaarde in en roept zichzelf recursief op voor elke refererende kolom. Dit algoritme werkt omdat een kolom nooit een vreemde sleutel kan zijn naar twee verschillende kolommen.
Wanneer één van de kolommen uit het voorbeeld een waarde krijgt, wordt de ketting telkens afgelopen tot bij Course.course#. Aangezien dit zelf geen vreemde sleutel is, krijgt deze kolom zijn waarde en wordt hier de functie `setKeys` gebruikt : de ketting wordt zo recursief terug opgeklommen.
Dit mechanisme werd niet geïmplementeerd bij het op NULL stellen van een vreemde sleutel omdat de primaire sleutels uit de cyclus toch nooit NULL kunnen zijn.

3. Storage details

3.1. Objecten retourneren

Om het kopiëren van objecten (met een door de gebruiker gekozen complexiteit) zoveel mogelijk te vermijden, wordt (een pointer naar) het object dat het laatst uit de databank gelezen werd intern bijgehouden. Op die manier kan een referentie naar dit object geretourneerd worden. Indien de applicatie dit object verder niet meer nodig heeft moet het dus nergens gekopiëerd worden. Het komt bijvoorbeeld vaak voor dat de databank volledig doorlopen moet worden en alle voorkomende namen gewoon in een Windows scrollbox gezet moeten worden, zonder dat ze later nog nodig zijn.

Om dit te bereiken wordt de zogenaamde placementsyntax van de new operator gebruikt (zie [5]). Normaal gezien doet de new operator twee zaken :

- geheugen alloceren voor het object (op de heap)
- het object initialiseren door de constructor op te roepen

Met de placementsyntax kunnen deze twee taken apart gebeuren. Er wordt éénmaal geheugen gealloceerd met behulp van de default constructor, en daarna wordt deze geheugenplaats telkens opnieuw gebruikt om het object met een andere constructor te initialiseren. Bij dit laatste wordt aan new de (eerder gereserveerde) geheugenplaats meegegeven, de plaats waar het object moet gemaakt worden.

Dit heeft tot gevolg dat de destructor niet automatisch zal opgeroepen worden, zoals normaal is bij het gebruik van new, maar dat ook de delete operator niet kan gebruikt worden, omdat het geheugen dan vrijgegeven wordt. De oplossing is een expliciete destructor aanroep. Het spreekt vanzelf dat er voorzichtig omgesprongen moet worden met zo'n expliciete destructoroproepen, maar in combinatie met de new-placementsyntax is dit essentiëel.

Door het gebruik van deze techniek moet de geheugenruimte voor het object slechts éénmaal gealloceerd worden, wat efficiënt is. Het alloceren en terug vrijgeven van geheugen vergt immers relatief veel tijd. Deze werkwijze eist echter wel dat het object beschikt over een default constructor. Dit is eveneens nodig voor het werken met de STL-containers en vormt dus geen grote beperking.

3.2. Sorteren van de tabellen

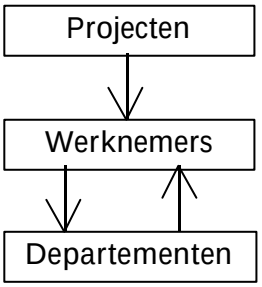
3.2.1. Algoritme

Wanneer een object fysisch in meerdere tabellen bewaard wordt, is de volgorde waarin de tabellen aan bod komen bij het bewaren van een object heel belangrijk. Dit om problemen met de referentiële integriteit te voorkomen. Eén en ander wordt met een voorbeeldje duidelijk gemaakt.

We beschouwen een databank met drie tabellen (hierbij werden de primaire sleutels dubbel onderlijnd en de vreemde sleutels enkel onderlijnd) :

- Werknemers : emp#, e_name, salaris, dept#
- Departementen : dept#, d_name, budget, head_emp#
- Projecten : proj#, p_name, head_emp#

De referentiële cyclus van de tabellen wordt getoond in figuur g. Stel dat deze informatie allemaal in één object zit. Wanneer zo'n object in de databank toegevoegd wordt en de tabel met de projecten wordt eerst genomen, kan de integriteit geschonden worden omdat de leidinggevende werknemer mogelijk nog niet in de Werknemers tabel aanwezig is. De volgorde waarin de tabellen Departementen en Werknemers bewaard worden speelt echter geen rol. Belangrijk is alleen dat deze voor het project toegevoegd worden. Hoe kan de ideale volgorde nu algoritmisch bepaald worden ?



Figuur G : Referentie cyclus van tabellen

Het is duidelijk dat een tabel die geen vreemde sleutels bevat (en er dus geen enkele pijl van vertrekt) zeker eerst aan de beurt mag komen. De tabellen kunnen dus gesorteerd worden op basis van het aantal verschillende tabellen waarnaar ze vreemde sleutels bevatten.

In het voorbeeld kan men vaststellen dat alle drie de tabellen naar één tabel verwijzen en er dus op basis van enkel deze redenering niet gesorteerd kan worden.
Een tweede belangrijk aspect is het aantal refererende tabellen (het aantal pijlen die toekomen). Hoe meer tabellen er naar een bepaalde tabel verwijzen, hoe sneller deze tabel aan de beurt moet komen.
In het voorbeeld bekomt men op deze manier de volgorde Werknemers (2), Departementen (1) en Projecten (0), wat een goede oplossing is.
Door eerst het aantal gerefereerde tabellen in stijgende orde te sorteren, en bij gelijkheid het aantal refererende tabellen in dalende orde te sorteren, bekomt men dus altijd de best mogelijke volgorde van de tabellen. Belangrijk is hierbij dat men sorteert op het aantal verschillende tabellen dat gerefereerde of refererende sleutels bevat en niet op het aantal sleutels zelf.
Merk ook op dat bij het wissen van een object deze volgorde omgekeerd moet worden.

3.2.2. Implementatie

Dit algoritme wordt geïmplementeerd door eerst het aantal verschillende tabellen te tellen waarnaar de tabel vreemde sleutels bevat. Hiervoor worden alle kolommen uit een tabel overlopen en indien een vreemde sleutel gevonden wordt, wordt de tabelnaam die de gerefereerde kolom bevat in een verzameling bewaard. Als datastructuur wordt de STL set container gebruikt. Hierna worden de elementen uit deze opgebouwde verzameling geteld en in de tabel bewaard. Voor het tellen van het aantal verschillende refererende tabellen wordt een analoge techniek toegepast waarbij ook weer de lijst van refererende kolommen noodzakelijk is.
Voor het sorteren zelf wordt de STL-functie sort gebruikt waarbij het volgende functie-object meegegeven wordt :

```
struct tableComp {
   bool operator() (const Table *t1, const Table *t2) {
      if (t1 -> getFK() < t2 -> getFK())
         return true;
      else if (t1 -> getFK() > t2 -> getFK())
         return false;
      else
         return t1 -> getRK() > t2 -> getRK();
   }
};
```

De functies `getFK()` en `getRK()` halen respectievelijk het aantal verschillende vreemde sleutel tabellen en het aantal verschillende refererende tabellen op.

Deze sortering werd in `Storage` constructor geïmplementeerd omdat het volledige databankschema dan zeker gedefinieerd is en er voordien zeker geen databanktransacties kunnen gebeuren.

3.3. Functies

Bij de naamgeving van de operaties op basis van de primaire sleutel werd zoveel mogelijk de STL-container terminologie overgenomen, zoals bijvoorbeeld `insert`, `erase` en `find`. Verder is ook nog de functie `update` aanwezig.
De zoek-functie zal een SQL SELECT query opbouwen waarbij de primaire sleutelkolommen uit het object gelijk moeten zijn aan hun opgegeven waarde. Bovendien wordt automatisch een join uitgevoerd (op basis van de gelijkheid van de vreemde sleutels).
De andere functies overlopen de tabellen (in de welbepaalde volgorde) en sturen de respectieve functie door naar elke tabel. Als een bepaalde rij in één der tabellen al voorkomt, dan wordt de integriteitswaarschuwing door middel van een returnwaarde doorgegeven.
Er is ook een functie `getError()` aanwezig waarmee foutboodschappen van de Engine opgehaald kunnen worden.

4. De iteratorklassen

Een `Storage` iterator wordt (onder andere) aangemaakt door de conditionele zoekfunctie (find) van `Storage` en bevat ook een pointer naar dit `Storage` object. Dit is nodig voor de toegang tot de `Adapter` en het databankschema. Bij het aanmaken van een `Storage` iterator wordt eveneens een `Engine` iterator aangevraagd aan de `Engine`. Door het polymorfismeconcept is verzekerd dat een iterator uit de goede driver teruggegeven wordt. Bij het gebruik van de `OdbcEngine` wordt dus effectief een `OdbcEngine` iterator verkregen.
Ook hier weer wordt een pointer naar het laatst gelezen object bijgehouden en de placementsyntax van new gebruikt, om dezelfde redenen waarom dit in `Storage` zo gebeurt (namelijk het nutteloos kopiëren van objecten vermijden).

Zoals gebruikelijk in STL, beschikt de `Storage` container ook over de functies `begin()` en `end()`, die beide iterator objecten retourneren en het mogelijk moeten maken de container volledig te doorlopen.
In STL is de conventie dat een einditerator voorbij het laatste element wijst. Om dit te implementeren wordt in STL vaak een booleanvariabele gebruikt. Deze methode is hier overgenomen. De beginiterator wordt bekomen door alle objecten te zoeken zonder voorwaarden op te leggen.
Twee einditeratoren zijn altijd aan elkaar gelijk (zelfs als ze ontstaan op het einde van twee verschillende `Storage` containers) en kunnen nooit gelijk zijn aan een niet einditerator.

5. Adapter details

In deze klasse moeten er zeker twee functies aanwezig zijn :

- `void readObject (const T&);`

  Dit is de functie die gebruikt wordt om een businessobject te lezen en dus de `DataColumn` objecten op te vullen met gegevens uit het object. De opgevulde kolommen kunnen gebruikt worden voor het toevoegen, wijzigen of wissen van een bepaalde rij uit de databank.

- `void writeObjec(T&);`

  Hierin gebeurt het omgekeerde, de gegevens uit de kolommen worden weggeschreven in het object. Deze functie wordt uitsluitend gebruikt wanneer gegevens uit de databank opgevraagd worden.

  Het is belangrijk dat dezelfde geheugenplaats van het gekregen argument gebruikt wordt, bijvoorbeeld met de placementsyntax van new : `new (t) BusinessObject(...)`. Zoals eerder geargumenteerd moet men hiervoor eerst zelf het object verwijderen met een expliciete destructoroproep. Dit zorgt ervoor dat de eventuele resources die door het object gebruikt worden, terug vrijgegeven worden.

  Het gebruik van de new operator laat het oproepen van de businessobjectconstructor toe. Dit kan nodig zijn om bepaalde variabelen in te stellen waarvoor er geen gewone accessorfuncties bestaan.

Zoals gezegd kunnen er in de `Adapter` constructor nog vreemde sleutels gedefinieerd worden (zodat de kolommen die bij een bepaalde tabel horen mooi in volgorde kunnen staan, zonder zich te bekommeren om afhankelijkheden).
Indien dit zou gewenst zijn, kan men in een `Adapter` klasse eveneens een andere `Storage` container bijhouden en gebruiken om bepaalde integriteitsregels te verzekeren.
Tot slot kan men nog opmerken dat, net zoals het databankschema, ook een groot gedeelte van deze adapter, in het bijzonder het triviale deel, automatisch gegenereerd kan worden met een script. Enige discipline in verband met de namen van de functies en de attributen zal dan wel vereist zijn.

6. OdbcEngine details

6.1. Kopie semantiek

In ODBC is het niet mogelijk de connectie en statement handles te kopiëren. Dit heeft als evident gevolg dat een `Storage` container, in feite een connectie met een databank, en een `Storage` iterator, in feite een geconditioneerd SQL-statement, niet kan gekopiëerd worden op een manier waarbij beide objecten onafhankelijk zijn van elkaar. Uiteraard kan er wel een zogenaamde shallowkopie gemaakt worden, waardoor er dan twee referenties naar dezelfde databank connectie of SQL-statement bestaan. Men moet er dan ook voor zorgen dat de betreffende handle niet gesloten wordt als er ergens nog een object bestaat dat van deze handle gebruik maakt. Om die reden is het nodig bij te houden hoeveel shallowkopieën er van elke container en elke iterator gemaakt werden.
Het aantal referenties naar een bepaalde databank connectie (het aantal kopieën van een `Storage` container dat dezelfde databankconnectie gebruikt) wordt met een teller in `OdbcEngine` geregistreerd. Enkel wanneer deze teller terug op nul komt, mag de connectie verbroken worden.
Bij het verwijderen van een `Storage` iterator moet niet enkel de bijhorende statement handle gesloten worden, maar moet ook de `Engine` iterator uit het geheugen verwijderd worden. Dit is de reden waarom deze teller in de `Storage` iteratorklasse opgevraagd moet kunnen worden en waarom er hiervoor dus functies voorzien werden in `Engine` iterator zelf (ook de teller wordt daar bijgehouden).

6.2. Automatische join

De `Engine` zorgt ook voor het automatisch uitvoeren van een join wanneer een object, dat zich in meerdere tabellen bevindt, opgevraagd wordt. Indien dit niet zou gebeuren, zou men per tabel een aparte SQL SELECT query moeten genereren, wat uiteraard minder efficiënt zou zijn vanwege het grotere aantal databanktoegangen. Bovendien zou het dan onmogelijk zijn om gegevens uit meer dan één tabel op te vragen indien deze niet allemaal dezelfde primaire sleutel zouden hebben.
De join gebeurt aan de hand van de vreemde sleutels die gedefinieerd werden.

6.3. Gebruik van ODBC

6.3.1. Environment handle

Om geen geheugen te verkwisten werd de environment handle van ODBC statisch gemaakt. Dit betekent dat elke databankconnectie die gebruik maakt van de `OdbcEngine` binnen hetzelfde environment loopt.
Dit komt bovendien ook de snelheid ten goede aangezien er nu slechts één keer een environment gealloceerd en terug vrijgegeven moet worden.

6.3.2. Ophalen van gegevens

De ODBC kern voorziet in functies voor het beheer van environments en connecties, evenals voor het ophalen van gegevens uit de databank. Hierbij moeten variabelen aan de kolommen gebonden worden, die dan fungeren als buffer. Deze werkwijze is echter niet erg praktisch.
Daarom werd gekozen voor het gebruik van de functie `getData()` waarbij het niet meer nodig is elke kolom eerst aan een variabele te binden. Deze functie behoort niet tot de kern, maar tot het ODBC extension level 1. Aangezien de meeste relevante ODBC drivers dit niveau ondersteunen is dit geen doorslaggevende belemmering.
In de `OdbcEngine` werden twee statische functies voorzien voor het converteren tussen de STL string, die in de rest van de bibliotheek gebruikt wordt, en de UCHAR* (unsigned char pointer), waarmee ODBC werkt.

6.3.3. Transactieondersteuning

ODBC heeft een functie waarmee databanktransacties effectief bevestigd of ongedaan gemaakt kunnen worden. Dit is eveneens een functie van extension level 1. Een complicatie hierbij is het feit dat niet elke ODBC driver deze transactieverwerking ondersteunt. Daarom wordt in `OdbcEngine` eerst nagegaan of de gebruikte driver deze capaciteiten al dan niet heeft. Het resultaat hiervan wordt bijgehouden in een logische variabele.

`Storage` voorziet in de functies `commit()` en `rollback()` die rechtstreeks doorgegeven worden aan de `Engine`, die deze functies enkel zal uitvoeren indien ze effectief door de driver ondersteund worden (de logische veranderlijke staat dan op waar). Dit om te vermijden dat ODBC een foutmelding genereert, alles gebeurt voor de gebruiker volledig transparant. Hiermee wordt voorkomen dat de gebruiker zijn programma moet veranderen als hij overstapt van een ODBC driver die transacties wel ondersteunt naar een driver die dit niet doet.

Standaard staat ODBC in autocommit mode, wat betekent dat elke instructie na het beëindigen automatisch uitgevoerd wordt. Door dit op te geven aan de `Database` constructor kan dit veranderd worden. Merk op dat de `OdbcEngine` dit verzoek enkel zal inwilligen indien manuele transactieverwerking door de driver ondersteund wordt.

6.3.4. Vereiste versie

Elke ODBC-functie die in deze bibliotheek gebruikt werd, behoort tot ODBC versie 1.0.

6.4. Gebruik van RTTI

Zoals in het ontwerp besproken, werd hier gekozen voor het gebruik van RTTI om de mapping tussen een C++ datatype en een relationeel domein (dat hier eenvoudig verondersteld werd) te bewerkstelligen. Een andere mogelijkheid zou er kunnen in bestaan om de nodige informatie, die hier met behulp van templates bekomen wordt, aan de gebruiker te vragen. Op die manier zou men geen RTTI moeten gebruiken, maar er zou dan ook geen enkele typecontrole mogelijk zijn zoals dit hier wel het geval is.

Een nadeel van RTTI is wel dat het aantal ondersteunde datatypes op voorhand moet vastliggen. Zonder de gebruiksmogelijkheden te beperken, volstaat het hierbij echter te voorzien in de equivalenten van de domeinen die in de databank aanwezig zijn.

Zoals eerder vermeld worden hier de volgende dataypes ondersteund : bool, short, integer, long, float, double en string. Uitbreiding naar andere types (bijvoorbeeld unsigned versies) is triviaal, tenminste, indien het gebruik van RTTI zoveel mogelijk gegroepeerd wordt in een zo beperkt mogelijk aantal functies. Hierbij kunnen volgende bedenkingen gemaakt worden :

- De conversie van een `Column` pointer naar een template `DataColumn` pointer, is noodzakelijk voor het opvragen van de kolomwaarde bij het wegschrijven van een object of het formuleren van een query (bijvoorbeeld om de waarden van de primaire sleutels te kunnen meegeven). Om de waarde te kennen is immers het type van de variabele nodig.
- Een analoge redenering kan gevolgd worden voor het instellen van een kolomwaarde bij het lezen van een object uit de databank.
- Bij het formuleren van een willekeurige query zal een `SimpleCondition` pointer geconverteerd moeten worden naar een `DataCondition` pointer vooraleer de waarde opgevraagd kan worden. Deze waarde moet echter, na de aanmaak zelf, nooit meer veranderd worden.

Hieruit kan geconcludeerd worden dat RTTI kan beperkt worden tot slechts 3 functies. In de bibliotheek zijn dit :

- `string getColumnValue(Column *);`

  Deze functie van `Engine` leest de waarde van een `DataColumn` en converteert die naar een string.
- `void setColumnValue(Column *, HSTMT, int);`

  Het instellen van een `DataColumn` is enkel nodig wanneer een object gelezen wordt en mag dus in `OdbcEngine` geïmplementeerd worden. HSTMT is de ODBC statement handle van de SQL SELECT query, de integer is het nummer van de databankkolom dat moet gelezen worden. Deze functie wordt eveneens gebruikt door de `OdbcEngine` iteratorklasse.
- `string getColumnCondition(SimpleCondition *);`

  De `SqlEngine` iterator functie leest niet alleen de waarde van de conditie maar bouwt ook de volledige conditie in SQL syntax op. Dit houdt onder andere in dat een vergelijking met NULL niet als operator (=) maar als woord (IS) toegevoegd wordt.

Deze drie functies werden virtueel gemaakt zodat ze gemakkelijk in een afgeleide klasse opnieuw geïmplementeerd kunnen worden, indien men het aantal ondersteunde datatypes wenst uit te breiden.

7. Condition details

7.1. Condities formuleren

Zoals besproken in het ontwerp kunnen enkel zinvolle condities geformuleerd worden, omdat de domeinen (in feite het templateargument type) van de kolom en de opgegeven waarde (of een andere kolom) moeten overeenstemmen. Deze typecontrole gebeurt volledig statisch.

De `DataCondition` objecten worden aangemaakt door de operator functies die in de `DataColumn` klasse gedefinieerd zijn. Volgende binaire, relationele C++ operatoren werden voorzien : ==, !=, <, >, <= en >=. Bovendien werden ook de unaire not operator ! en de conversie operator ingebouwd om vergelijkingen met NULL uit te drukken, zoals dit gebruikelijk is met C++ pointers. Volgende voorbeelden illustreren dit :

```
gWeight > 100   // onderdelen met een gewicht groter dan 100
gEmpCity != gDeptCity // werken in een andere gemeente
!gProjectNr     // werknemers zonder project (== NULL)
gFaxNr          // werknemers die een fax bezitten (!= NULL)
```

In de operatorstring uit de `DataColumn` klasse (in feite de `SimpleCondition` klasse) wordt onmiddellijk de SQL versie van de operator gezet, == wordt bijvoorbeeld vervangen door =. Indien men een driver schrijft voor een databank die de SQL operatoren niet begrijpt, dan moeten deze in de driver omgezet worden. In de `SqlEngine` gebeurt iets dergelijks wanneer het een vergelijking met NULL betreft.

Wanneer een `DataColumn` zo'n `DataCondition` object aangemaakt heeft, wordt dit geretourneerd als een volwaardig `Condition` object (met slechts één enkele voorwaarde) omdat dit ook al kan gebruikt worden als voorwaarde voor een zoekoperatie. Deze condities kunnen dan gecombineerd worden met de logische C++ operatoren `&&`, `||` en !. Hierbij mogen haakjes gebruikt worden om de normale precedentieregels te wijzigen.

7.2. Condities interpreteren

De lijst van voorwaarden, die opgebouwd wordt in postfix-notatie om het intern gebruik van haakjes te vermijden, wordt dan met behulp van een stapel terug omgezet naar infix notatie voor het opbouwen van de SQL query. Hierbij moet de string met de logische operatoren (&&, || en !) geconverteerd worden naar AND, OR en NOT. Dit is geen grote moeite omdat het aantal beperkt is tot amper drie en omdat er toch moet gecontroleerd worden of het een binaire of een unaire operator is (om het juiste aantal objecten van de stapel te kunnen halen).
Als datastructuur wordt hierbij de STL stack container adapter gebruikt.
Naast de opgelegde voorwaarden wordt de join van de verschillende tabellen op basis van de vreemde sleutels automatisch uitgevoerd zoals dit in de bibliotheek altijd gedaan wordt.

7.3. Kopiëren van condities

Het zal waarschijnlijk vaak voorkomen dat voorwaarden meerdere keren gebruikt worden of na eenmalig gebruik nog uitgebreid worden met bijkomende beperkingen. Er is dus duidelijk behoefte aan een mogelijkheid om conditieobjecten te kopiëren.
Er is eigenlijk nood aan een soort virtuele kopie constructor omdat de objecten in de lijst van de klasse `Condition` ofwel `SimpleCondition`'s ofwel verschillende template `DataCondition`'s kunnen zijn. Een virtuele kopie constructor is niet mogelijk in C++, maar hetzelfde effect kan wel gemakkelijk bekomen worden, bijvoorbeeld door gebruik te maken van een virtuele clone functie (zie [5]).
De methode is vrij eenvoudig : in de klasse `SimpleCondition` wordt een virtuele functie voorzien die dit object kopiëert en een `SimpleCondition` pointer naar de gemaakte kopie teruggeeft. Deze functie wordt in de afgeleide template klasse `DataCondition` opnieuw geïmplementeerd zodat er van het object in kwestie een juiste kopie gemaakt kan worden. Hier wordt eveneens een `SimpleCondition` pointer naar deze kopie geretourneerd. De kopie constructor van de klasse `Condition` hoeft nu enkel zijn lijst te overlopen, telkens de `clone()` functie op te roepen en de gekregen pointer in de lijst van de kopie te stockeren.

8. Gebruik van STL

In de bibliotheek wordt voor elke container de STL deque container gebruikt. Meestal is de voornaamste bewerking immers het vooraan en/of achteraan toevoegen van objecten en zoals opgeven in de STL specificatie (zie tabel b), is de deque container dan de beste keuze.

Zeker in het geval van de stapel container adapter die gebruikt wordt bij het opbouwen van de SQL conditie is de deque als basiscontainer ongetwijfeld de meest geschikte datastructuur. In een stapel kan er immers enkel achteraan toegevoegd en gewist worden.

Voor het tellen van het aantal verschillende tabellen op basis van de vreemde sleutels (zie `Storage`), met het oog op het sorteren ervan wordt een STL set datastructuur gehanteerd. Ook voor het sorteren zelf wordt een STL-algoritme gebruikt.

De containers die gebruikt worden voor het bijhouden van de lijst met tabellen (in `Database`), kolommen (in `Table`) en voorwaarden (in `Condition`) zijn als aparte types gedefinieerd in het bestand `global.h` en kunnen dus heel eenvoudig gewijzigd worden. Dit kan bijvoorbeeld nuttig zijn als volgende versies van STL nieuwe containers bevatten die sneller en beter zijn. Wanneer men een ander type van container wil gebruiken, moet men echter met een aantal eisen rekening houden. Deze eisen hebben betrekking op de iterator types die de containers moeten ondersteunen, en worden samengevat in tabel f.

Container	Vereiste iterator types
ColumnContainer	Forward iterator
TableContainer	* Random acces iterator * Reverse iterator
ConditionContainer	Forward iterator

Tabel F : Container eisen

Merk op dat de TableContainer strengere eisen stelt omdat het deze datastructuur is die gesorteerd wordt en ook in omgekeerde volgorde doorlopen moet worden (wanneer een object gewist wordt).

Hoofdstuk 5 : Besluit

Library design is language design. And vica versa.
Bell labs proverb & A. Koenig

1. Gebruik van de bibliotheek

Veronderstel dat men een applicatieprogramma heeft waarbij een lijst van objecten in het geheugen bijgehouden wordt. Om de één of andere reden kan men wensen dat deze transiënte objecten (die verdwijnen als de applicatie stopt) persistent gemaakt worden (en dus een levensduur krijgen die deze van een uitvoering van het programma overstijgt). Deze situatie kan bijvoorbeeld in de praktijk optreden wanneer men van een prototype, waarbij nog geen databank gebruikt wordt, een volledig programma wil maken, waarbij wel een databank moet gebruikt worden. Wat moet er dan concreet gebeuren indien men ervoor kiest om deze bibliotheek als `Storage` laag te gebruiken :

- Een eerste stap is de keuze van een RDBMS en het aanmaken van de databank, tabellen en kolommen die voor de opslag van het businessobject uit de applicatie nodig zijn. Merk op dat dit niet eigen is aan deze bibliotheek maar noodzakelijk is voor elke bibliotheek die men wil gebruiken.
- Verder moet er voor een `Engine` klasse gezorgd worden. Indien er voor de gekozen RDBMS een ODBC driver beschikbaar is, kan de hier ontwikkelde `OdbcEngine` gebruikt worden. Gezien de ruime verspreiding van ODBC drivers zal dit wel meestal het geval zijn. Er kan echter om performantieredenen gekozen worden om een op-maat-geschreven, native databank `Engine` te ontwikkelen, zodat men de extra ODBC laag niet nodig heeft. Wanneer de databank SQL-statements begrijpt, kan men dan al direct gebruik maken van de `SqlEngine`, zodat enkel de databankspecifieke zaken (zoals bijvoorbeeld het leggen van connecties) geïmplementeerd moeten worden.

- Wanneer men beschikt over een `Engine`, kan men de objecten aanmaken die het databankschema in C++ vastleggen (`Database`, `Table`, `DataColumn`). Deze fase is enorm belangrijk omdat op basis hiervan bijvoorbeeld de joins uitgevoerd worden, fouten opgeworpen worden als men een not-nullable kolom toch NULL wil maken, waarden voor de primaire sleutels meegegeven worden,... Deze taak kan vergemakkelijkt worden door het maken van een script dat deze object definities automatisch genereert.
- Nog belangrijker is echter het schrijven van de adapter klasse die de mapping tussen de objecten en de databank verzorgt. Deze klasse moet afgeleid worden van `Adapter`. Aandachtspunten hierbij zijn de expliciete destructoroproep en de placementsyntax van new. Het triviale gedeelte van deze klasse kan eveneens automatisch gegenereerd worden door gebruik te maken van een script.
- Tot slot, kan men de bovenstaande objecten gebruiken om de uiteindelijke `Storage` container aan te maken. Deze moet in het applicatieprogramma overal de plaats van de transiënte container in het prototype innemen. Hierbij dient opgemerkt te worden dat de `Storage` iterator van het (eerder zwakke) input iterator type is. Dit zal echter doorgaans geen probleem zijn, aangezien een relationele databank zelf al heel wat mogelijkheden biedt voor de visualisatie van de gegevens, bijvoorbeeld een welbepaalde ordening. De bibliotheek zal de objecten immers ophalen in de volgorde zoals ze in de databank gevisualiseerd zouden worden. Ingewikkelde algoritmen (waarbij bijvoorbeeld random-access noodzakelijk is) zullen dus niet vaak nodig zijn.

In de beschreven werkwijze werd er totaal niets, maar dan ook niets, gewijzigd aan de businessobject klasse. Geen bijkomende afleidingen, geen bijkomende functies, geen... De objecten van deze klassen hoeven niet te weten of ze transiënt of persistent bewaard worden.

2. Voorbeelden

2.1. Databankschema

De twee voorbeelden die uitgewerkt en kort besproken worden, maken gebruik van een gemeenschappelijke databankstructuur die er als volgt uitziet :

```
CREATE TABLE DEPART (
     DEPID NUMERIC(4) NOT NULL,
```

```
        NAME  CHAR(25),
        CITY  CHAR(10),
        PRIMARY KEY (DEPID)
    )
    CREATE TABLE ADMIN (
        EMPID  NUMERIC(4) NOT NULL,
        FNAME  CHAR(20),
        LNAME  CHAR(20),
        STREET CHAR(25),
        CITY   CHAR(10),
        DEPID  NUMERIC(4),
        PRIMARY KEY (EMPID)
        FOREIGN KEY (DEPID) REFERENCES DEPART
    )
    CREATE TABLE FINAN (
        EMPID  NUMERIC(4) NOT NULL,
        SALARY NUMERIC(6),
        PRIMARY KEY (EMPID),
        FOREIGN KEY (EMPID) REFERENCES ADMIN
    )
```

Deze tabellen werden aangemaakt in Borland dBase 5.0 for Windows. Hiervoor is een ODBC driver beschikbaar zodat ook de `OdbcEngine` in het programma zal gebruikt worden.

2.2. Voorbeeld : Financial

In dit eerste voorbeeld wordt er enkel gewerkt met de tabellen ADMIN en FINAN. De tabel DEPART en ook de kolom DEPID (in ADMIN) zal dan ook volledig buiten beschouwing gelaten worden.

Er wordt een bedrijf verondersteld waarbij de databank met de adressen van de werknemers (ADMIN) publiek toegankelijk is. Op de loondienst wordt deze tabel eveneens gebruikt, in combinatie met een tabel die het salaris (en andere relevante zaken) van elke werknemer bevat (FINAN). Uiteraard heeft enkel deze dienst toegang tot deze tabel.

Dergelijke situatie kan bijvoorbeeld ook optreden in een ziekenhuisomgeving, waar de persoonsgegevens en de medische gegevens van de patiënten apart opgeslagen worden.

De loondienst van dit bedrijf werkt met een werknemersklasse waarin zijn salaris als member variabele opgenomen werd (de klasse Financial). Dit is dus een voorbeeld van één object klasse waarvan de variabelen over twee verschillende tabellen verspreid worden.

Een eerste stap is dus het vastleggen van het databankschema in C++.

```
Engine *gOdbc = new OdbcEngine();
Database gFinancial ("financial", gOdbc, false);
Table gAdmin (gFinancial, "admin");
DataColumn<int>    gAdmId     (gAdmin, "EMPID", true);
DataColumn<string> gAdmFname  (gAdmin, "FNAME");
DataColumn<string> gAdmLname  (gAdmin, "LNAME");
DataColumn<string> gAdmStreet (gAdmin, "STREET");
DataColumn<string> gAdmCity   (gAdmin, "CITY");
Table gFinan (gFinancial, "finan");
DataColumn<int>    gFinId     (gFinan, "EMPID", true, &gAdmId);
DataColumn<long>   gFinSal    (gFinan, "SALARY");
```

Als data resource naam wordt hier dus financial gebruikt. Dit is dan ook de naam die in de ODBC driver manager geregistreerd moet zijn (verwijzend naar de dBase driver en het pad met de tabellen).

Vervolgens kan men de adapter klasse schrijven.

```
class FinancialAdapter : public Adapter<Financial> {
   public:
      void readObject (const Financial& f) {
         gAdmId.setData(f.getId());
         gAdmFname.setData(f.getFname());
         gAdmLname.setData(f.getLname());
         gAdmStreet.setData(f.getStreet());
         gAdmCity.setData(f.getCity());
         gFinSal.setData(f.getSalary());
      }
      void writeObject (Financial *f) {
         f -> Financial::~Financial();
         new (f) Financial(gAdmId.getData(), gAdmFname.getData(),
                           gAdmLname.getData(), gAdmStreet.getData(),
                           gAdmCity.getData(), gFinSal.getData());
      }
};
```

Bemerk hierbij volgende zaken :

- er zijn enkel twee functies aanwezig (deze die vereist zijn)
- de expliciete destructoroproep
- het gebruik van de placementsyntax van new

Nu is men klaar om het applicatieprogramma te schrijven. Dit programma moet volgende bewerkingen verrichten :

- de volledige lijst van de werknemers tonen
- het gemiddelde loon berekenen
- alle werknemers tonen die meer dan het gemiddelde verdienen

- een nieuwe werknemer toevoegen (die het gemiddelde loon zal verdienen)
- het loon van een bepaalde werknemer verhogen.
- nogmaals de volledige lijst van werknemers tonen

Bij het berekenen van het gemiddelde loon worden er twee STL-algoritmen gebruikt, enkel en alleen om te illustratreren dat dit mogelijk is. Het is niet de bedoeling deze berekening heel efficiënt te doen. De algoritmen die gebruikt worden zijn :

- `void for_each (InputIterator start, InputIterator stop, FunctionObject functie);`

  Dit algoritme past een bepaalde functie (al dan niet ingekapseld in een functie-object) toe op elk object in de container.
- `void count_if (InputIterator start, InputIterator stop, FunctionObject functie, int& teller);`

  Hiermee kan het aantal objecten die aan een bepaald predicaat voldoen geteld worden.

Hierna volgt de volledige listing van het programma.

```
// Functie-object voor het berekenen van het totale loon
class SalarySum {
   public:
      SalarySum (long& s) : iSum(s) {
         iSum = 0;
      }
      void operator() (const Financial& f) {
         iSum += f.getSalary();
      }
      long& iSum;
};

// Dit functie-object geeft altijd waar en wordt gebruikt
// om het aantal elementen in de container te tellen
class AlwaysTrue {
   public:
      bool operator() (const Financial& f) {
         return true;
      }
};

// Hoofprogramma
void main() {
   // variabelen
   FinancialAdapter fadap;
   Storage<Financial> fstore (gFinancial, &fadap);
   Storage<Financial>::iterator fit;
```

```
   // lijst met werknemers tonen
   cout << "Alle werknemers : " << endl;
   for (fit = fstore.begin(); fit != fstore.end(); fit++)
      cout << *fit << endl;
   cout << endl;

   // berekenen van het gemiddelde loon met STL-algoritmen
   long total;
   SalarySum sos(total);
   for_each (fstore.begin(), fstore.end(), sos);

   int number = 0;
   count_if (fstore.begin(), fstore.end(), AlwaysTrue(), number);

   long gem = total / number;
   cout << "Gemiddelde loon : " << gem << endl << endl;

   // lijst met werknemers boven het gemiddelde loon tonen
   cout << "Alle werknemers boven het gemiddelde loon : " << endl;
   Condition c = gFinSal > gem;
   for (fit = fstore.find(c); fit != fstore.end(); fit++)
      cout << *fit << endl;
   cout << endl;

   // nieuwe werknemer toevoegen
   Financial piet(999,"Piet","Pol","Pollekesstraat 10","Brugge",gem);
   cout << "Toevoegen werknemer : " << fstore.insert(piet) << endl;

   // loon van werknemer verhogen
   Condition d = gAdmFname == "Jan" && gAdmLname == "Jansens";
   fit = fstore.find(d);
   if (fit != fstore.end()) {
      (*fit).setSalary((*fit).getSalary() + 5000);
      fstore.update((*fit));
   }
   else
      cout << "Gezochte werknemer niet gevonden." << endl << endl;

   // lijst met werknemers tonen
   cout << "Alle werknemers : " << endl;
   for (fit = fstore.begin(); fit != fstore.end(); fit++)
      cout << *fit << endl;
   cout << endl;
}
```

Hiermee worden enkele belangrijke zaken gedemonstreerd, zoals bijvoorbeeld het formuleren van condities (kolom-waarde) en het gebruik van STL.

2.3. Voorbeeld : Employee

In tegenstelling tot het voorgaande voorbeeld wordt hier de tabel FINAN niet gebruikt, maar wel de andere twee. Er zijn twee belangrijke C++ klassen : Employee en Departement, waarbij een Employee object een pointer naar een Departement object bevat. De twee objecten worden elk apart in een tabel bewaard.
Het vastleggen van het databankschema gebeurt als volgt.

```
Engine *gOdbc = new OdbcEngine();
Database gEmployee ("administration", gOdbc);
Table gAdmin (gEmployee, "admin");
DataColumn<int>    gAdmId     (gAdmin, "EMPID", true);
DataColumn<string> gAdmFname  (gAdmin, "FNAME");
DataColumn<string> gAdmLname  (gAdmin, "LNAME");
DataColumn<string> gAdmStreet (gAdmin, "STREET");
DataColumn<string> gAdmCity   (gAdmin, "CITY");
DataColumn<int>    gAdmDeptId (gAdmin, "DEPID", false, 0, true);
Table gDepart (gEmployee, "depart");
DataColumn<int>    gDepId   (gDepart, "DEPID", false, 0, true);
DataColumn<string> gDepName (gDepart, "NAME", false, 0, true);
DataColumn<string> gDepCity (gDepart, "CITY", false, 0, true);
```

Hierbij kunnen we een heel belangrijke opmerking maken. De kolom DEPID in de tabel DEPART staat niet als primaire sleutel aangeduid, hoewel dit wel zo is in de databank. Vooraleer hierop dieper in te gaan, wordt nu eerst de adapter getoond.

```
class EmployeeAdapter : public Adapter<Employee> {
   public:
      EmployeeAdapter() {
         gAdmDeptId.setForeignKey(&gDepId);
         d = new Department();
      }
      void readObject(const Employee& e) {
         gAdmId.setData(e.getId());
         gAdmFname.setData(e.getFname());
         gAdmLname.setData(e.getLname());
         gAdmStreet.setData(e.getStreet());
         gAdmCity.setData(e.getCity());
         if (e.getDepartment()) {
            gDepId.setData(e.getDepartment() -> getId());
            gDepName.setData(e.getDepartment() -> getName());
            gDepCity.setData(e.getDepartment() -> getCity());
         }
         else {
            gAdmDeptId.setNull();
            gDepId.setNull();
```

```
            gDepName.setNull();
            gDepCity.setNull();
         }
      }
      void writeObject (Employee *e) {
         e -> Employee::~Employee();
         d -> Department::~Department();
         if (gDepId.isNull())
            d = 0;
         else
            new (d) Department(gDepId.getData(), gDepName.getData(),
                               gDepCity.getData());
         new (e) Employee(gAdmId.getData(), gAdmFname.getData(),
                          gAdmLname.getData(), gAdmStreet.getData(),
                          gAdmCity.getData(), d);
      }
      ~EmployeeAdapter() {
         delete d;
      }
   private:
      Department *d;
};
```

Deze adapter zorgt ervoor dat bij het lezen van een Employee object, ook het Departement object correct aangemaakt wordt. Hierbij werd rekening gehouden met het feit dat de Departement pointer de waarde NULL kan hebben. Indien men een Employee object wil bewaren waarvan het Departement object niet bestaat, worden de betreffende kolommen op NULL ingesteld. De bibliotheek zal deze schending van de integriteit, in de Departement tabel probeert men dan immers een rij met als primaire sleutel NULL toe te voegen, aangeven door een false te retourneren.

Merk ook op dat de vreemde sleutel deze keer in de adapter gedefinieerd wordt.

Nu kunnen we ook verklaren waarom de afwijking van het databankschema in C++ ten opzichte van het relationele databankschema met betrekking tot de primaire sleutel toegelaten wordt. Stel eerst dat dit niet zo zou zijn en de `DataColumn` gDepId dus wel degelijk ook als primaire sleutel gedefinieerd zou zijn.

Wanneer men een Employee object opvraagt uit de databank, is uiteraard enkel het employee nummer gekend (de primaire sleutel uit de ADMIN tabel). De primaire sleutel uit de DEPART tabel zal meestal niet op voorhand gekend zijn omdat deze afhangt van de werknemergegevens die net opgevraagd worden. Toch zal de bibliotheek in dit geval deze waarde willen gebruiken in de SQL SELECT-query omdat deze kolom nu eenmaal een primaire sleutel is, met alle onvoorspelbare fouten tot gevolg. Dit probleem kan

worden opgelost door een aparte `Storage` container te maken voor de Departement objecten. Men moet dan eerst de werknemergegevens opvragen (zonder daarbij de DEPART tabel te gebruiken) en daarna zelf de rest van de departementgegevens opvragen. Analoge voorzieningen voor het correct toevoegen, wijzigen en wissen van een werknemer moeten dan ook zelf geïmplementeerd worden. Hierdoor wordt de complexiteit van het programma uiteraard verhoogd.

Wanneer men de primaire sleutel uit de DEPART tabel niet als primaire sleutel opgeeft in C++, zal de bibliotheek de waarde in gDepId niet gebruiken bij het opbouwen van de query. Aangezien de join echter wel zal uitgevoerd worden (door de vreemde sleutel definitie), zullen de resultaten correct zijn.

De gevolgen van deze methode zijn de beperkte mogelijkheden om met Departement objecten te werken, zoals bijvoorbeeld het wissen of wijzigen ervan (hiervoor is het kennen van de primaire sleutel een noodzaak). In dergelijke gevallen zullen deze beperkingen echter meestal zelfs gewenst zijn. De applicatie voor het beheer van werknemergegevens zal immers niet verondersteld worden departement eigenschappen te wijzigen of zelfs volledige departementen te wissen.

Er dient nogmaals op gewezen te worden dat het gebruik van deze aangeboden faciliteit niet noodzakelijk is. Het kan, door gebruik te maken van de eerder beschreven methode (een aparte `Storage` container voor departementen), ook volledig anders gebeuren, mits wat meer werk.

In het onderstaande applicatieprogramma wordt een vergelijking gemaakt tussen twee verschillende kolommen, een werknemer opgezocht en verwijderd.

```
void main() {
   // variabelen
   EmployeeAdapter eadap;
   Storage<Employee> estore (gEmployee, &eadap);
   Storage<Employee>::iterator eip;

   // werknemers opvragen die in hun eigen gemeente werken
   Condition c = gAdmCity == gDepCity;
   cout << "Lijst van de werknemers die in hun eigen gemeente werken"
        << endl;
   for (eip = estore.find(c); eip != estore.end(); eip++)
      cout << *eip << endl;
   cout << endl;

   // werknemer 24 opvragen en wissen
   Employee weg(24);
```

```
   bool wissen = true;
   try {
      weg = estore.find(weg);
   }
   catch(NoDataFound) {
      cout << "Deze werknemer werd al verwijderd." << endl;
      wissen = false;
   }
   if (wissen) {
      cout << weg << endl;
      estore.erase(weg);
   }

   // volledige lijst tonen
   cout << endl << "Volledige lijst" << endl;
   for (eip = estore.begin(); eip != estore.end(); eip++)
      cout << *eip << endl;
   cout << endl;
}
```

2.4. Efficiëntie

Hoewel er geen expliciete tijdsmetingen uitgevoerd werden, kon er bij de uitvoering van deze twee voorbeelden duidelijk vastgesteld worden dat de effectieve databanktoegang de vertragende factor is.

3. Conclusies

Met betrekking tot de geformuleerde doelstellingen kunnen volgende conclusies getrokken worden :

- De gebruiker komt, na het éénmalig definiëren, niet meer in contact met tabellen, rijen en kolommen. Deze zitten volledig weggestopt in de adapter klasse (beheersen van de complexiteit door encapsulatie is het voornaamste doel van OO). Het programma werkt enkel in op businessobjecten, wat het abstractieniveau verhoogt. Om voldoende flexibiliteit te laten, kunnen de kolomobjecten echter toch nog gebruikt worden bij het formuleren van willekeurige queries.
- De eventuele, onnatuurlijke primaire sleutels die ingevoerd werden om gemakkelijker te kunnen werken met het relationele datamodel, kunnen eveneens volledig verborgen worden.
- Aan de businessobjecten moet er niet gewijzigd of toegevoegd worden vooraleer ze persistent gemaakt kunnen worden. Het al dan niet persistent maken van deze objecten is volledig in handen van het applicatieprogramma.

- De `Storage` container is volledig uniform met de interface van de STL-containers.
- Zoals in de voorbeelden geïllustreerd werd, is er samenwerking tussen de Storage container en de STL-algoritmen mogelijk. Het is dus in principe mogelijk om een dergelijke persistente container in volgende versies van STL standaard te voorzien.
- Tijdens het ontwikkelen van de bibliotheek werden vooral diverse STL-containers gebruikt, maar ook het sorteeralgoritme. Door deze containers te gebruiken, kon heel wat tijd bespaard worden. Bovendien is het zo dat, indien elke container zelf nog geïmplementeerd zou moeten worden, waarschijnlijk overal dezelfde container gebruikt zou zijn. Dit was misschien niet zo efficiënt geweest. Het feit dat weinig algoritmen aan bod kwamen is te verklaren omdat hier een bibliotheek en geen applicatieprogramma ontwikkeld werd. Het valt immers op dat in de gegeven voorbeelden wel gemakkelijk (nuttige) toepassingen van de beschikbare algoritmen gevonden werden. De bruikbaarheid van STL wordt dus goed bevonden.
- Over de andere uitbreiding van de C++ standaard kunnen volgende opmerkingen gemaakt worden :

  * RTTI :

    Hiermee werd statische typecontrole mogelijk bij de mapping tussen domein en datatype.

  * Exception handling :

    Dit is een flexibel systeem dat echter tot veel code kan leiden om alle nodige foutcontroles te doen. Bovendien is er geen mogelijkheid om enkel een waarschuwing te genereren, waarbij het programma niet noodzakelijk moet eindigen als deze niet opgevangen wordt. Hiervoor moet terug een beroep gedaan worden op de traditionele C stijl waarbij bepaalde codes geretourneerd worden.

4. Beperkingen

Tot slot kunnen ook nog enkele beperkingen vermeld worden :

- Beperkte ondersteuning van C++ datatypes door het gebruik van RTTI. In feite hoeft men echter enkel de datatypes die corresponderen met de databankdomeinen te ondersteunen. Bovendien kunnen deze gemakkelijk uitgebreid worden.
- Bij complexe klassen met pointers naar andere klassen moet men, om alle functionaliteiten te hebben, per klasse een `Storage` container bijhouden en zelf

zorgen dat de nodige onderlinge referenties juist zijn. Hierbij kan de referentiële integriteit die in de databank zelf gedefinieerd werd als ultiem vangnet dienen.

- De ontwikkelde iteratorklasse is van de input iteratorsoort en biedt niet zoveel mogelijkheden (bijvoorbeeld in vergelijking met pointer semantiek).

Appendix A : Header bestanden

Being a good computer scientist means having a high and low-level understanding simultaneously.
Donald Knuth

1. Global.h

```
#ifndef _GLOBAL_H
#define _GLOBAL_H

#include <deque>

using namespace std;

// moet forward iterator hebben
class Column;
typedef deque<Column*> ColumnContainer;
typedef deque<Column*>::iterator ColumnIterator;

// moet random access en reverse iterator hebben
// (onder andere sorteren in Storage)
class Table;
typedef deque<Table*> TableContainer;
typedef deque<Table*>::iterator TableIterator;
typedef deque<Table*>::reverse_iterator TableReverseIterator;

// moet forward iterator hebben
class SimpleCondition;
typedef deque<SimpleCondition*> CondContainer;
typedef deque<SimpleCondition*>::iterator CondIterator;

#endif
```

2. Exception.h

```
#ifndef _EXCEPTION_H
#define _EXCEPTION_H

#include <string>

using namespace std;

class StoreException {
   public:
      StoreException() { }
      StoreException(string s) : message(s) { }
      virtual string getMessage() const {
         if (message.length() == 0)
            return "";
         return message;
      }
      virtual ~StoreException() { }
   protected:
      string message;
};

// Opgeworpen door de routines die RTTI gebruiken, als het type niet kon
// geïdentificeerd worden
class UndefinedType : public StoreException { };

// Opgeworpen door Storage als er nergens een primary key aanwezig is
class UndefinedPrimaryKey : public StoreException { };

// Opgeworpen door OdbcEngine als er geen connectie kan gemaakt worden
class ConnectError : public StoreException { };

// Opgeworpen door OdbcEngine als er een fout gebeurde in de uitvoering
// van een statement
class StatementError : public StoreException { };

// Opgeworpen door OdbcEngine als een SELECT statement geen data vind
class NoDataFound : public StoreException { };

// Opgeworpen als een not-nullable DataColumn op NULL gezet wordt
class NotNullable : public StoreException { };

#endif
```

3. Adapter.h

```
#ifndef _ADAPTER_H
#define _ADAPTER_H

template<class T> class Adapter {
   // Abstracte interface klasse voor de mapping object-tabel
   public:
      virtual void readObject(const T&) = 0;
      virtual void writeObject(T*) = 0;
      virtual ~Adapter() { }
};

#endif
```

4. Engine.h

```
#ifndef _ENGINE_H
#define _ENGINE_H

#include <stdio.h>
#include <typeinfo.h>
#include <string>
#include "global.h"
#include "datacol.h"
#include "exception.h"

using namespace std;

class Column;
class Table;
class SimpleCondition;
class Engine {
   // Abstracte interface klasse voor databanktoegang
   public:
      // de boole variabele geeft de commit mode aan
      // true = automatisch, false = manueel
      virtual void open (string, bool) = 0;
      virtual void close () = 0;
      // false bij een integrity violation
      virtual bool insert (ColumnContainer&) = 0;
      virtual bool update (ColumnContainer&) = 0;
      virtual void remove (ColumnContainer&) = 0;
      virtual void retrieve (TableContainer&) = 0;
      virtual void commit () = 0;
      virtual void rollback () = 0;
      virtual string getError() = 0;
      virtual ~Engine() { }
      class iterator {
         public:
            iterator () : iCounter(0) { }
            virtual void advance (TableContainer&) = 0;
            void incCount();
            void decCount();
            bool isCount();
            virtual ~iterator() { }
         private:
            // bijhouden van het aantal shallow iterator kopies
            // belangrijk voor het opkuisen in Storage::iterator
            int iCounter;
      };
      virtual Engine::iterator* getIterator(TableContainer&,
                                            CondContainer&) = 0;
   protected:
      // gebruikt RTTI
      virtual string getColumnValue(Column *);
};
```

```
#endif
```

5. Sql_eng.h

```
#ifndef _SQL_ENGINE_H
#define _SQL_ENGINE_H

#include <string>
#include <stack>
#include <deque>

#include "global.h"
#include "exception.h"
#include "engine.h"
#include "column.h"
#include "scond.h"

using namespace std;

class SqlEngine : public Engine {
   // Opbouw van de SQL statements
   protected:
      string SqlInsert(ColumnContainer&);
      string SqlDelete(ColumnContainer&);
      string SqlUpdate(ColumnContainer&);
      string SqlSelect(TableContainer&);

      class iterator : public Engine::iterator {
         public:
            string SqlCondSelect(TableContainer&, CondContainer&);
         private:
            // gebruikt RTTI
            virtual string getColumnCondition (SimpleCondition*);
      };
};

#endif
```

6. Odbc_eng.h

```
#ifndef _ODBC_ENGINE_H
#define _ODBC_ENGINE_H

#include <string>

// ODBC includes
#include <windows.h>
#include <odbcinst.h>
#include <w16macro.h>
#include <sql.h>
#include <sqlext.h>

#include "global.h"
#include "engine.h"
#include "sql_eng.h"
#include "table.h"
#include "column.h"
#include "datacol.h"
#include "exception.h"

using namespace std;

// Gebruikt voor SQLGetError en SQLGetData<string>
const int gStringLen = 200;

class OdbcEngine : public SqlEngine {
   // ODBC implementatie voor databanktoegang
   public:
      OdbcEngine ();
      ~OdbcEngine ();
      void open(string, bool);
      void close();
      bool insert(ColumnContainer&);
      bool update(ColumnContainer&);
      void remove(ColumnContainer&);
      void retrieve(TableContainer&);
      void commit();
      void rollback();
      string getError();

      class iterator;
      friend class iterator;

      class iterator : public SqlEngine::iterator {
         public:
            friend class OdbcEngine;
            void advance (TableContainer&);
            ~iterator ();
         private:
            iterator (OdbcEngine *, HDBC, TableContainer&,
                      CondContainer&);
```

```
            HSTMT iHstmt;
            OdbcEngine *iEngine;
      };

      Engine::iterator *getIterator(TableContainer&, CondContainer&);

   private:
      // gebruikt RTTI
      virtual void setColumnValue (Column *, HSTMT, int);

      // conversie tussen UCHAR* en string
      static void strtochar(string, UCHAR*);
      static string chartostr (UCHAR *);

      // Uitvoeren van 1 ODBC statement (en terug sluiten van de handle)
      // false bij een integrity violation
      bool OdbcExecute (string);

      // gemeenschappelijk ODBC environment
      static HENV cHenv;
      // ODBC connectie
      HDBC iHdbc;
      // geeft aan of een commit/rollback mag doorgegeven worden aan ODBC
      bool iTransactionSupport;
      // bijhouden van het aantal Storage objecten die deze connectie
      // gebruiken belangrijk voor het éénmalig sluiten van de connectie.
      int iCounter;
      string iDataSourceName;
};

#endif
```

7. Database.h

```
#ifndef _DATABASE_H
#define _DATABASE_H

#include <string>

#include "global.h"

using namespace std;

class Engine;
class Table;
class Database {
   // Bevat de beschrijving van en toegang tot de databank
   public:
      // het bool argument zet de autocommit mode
      Database (string, Engine *, bool = true);
      void installTable (Table *);
      inline Engine* getEngine() const;
      inline string getName() const;
      inline TableContainer& getTables();
      inline bool getCommitMode();
      ~Database();

   private:
      string iDbName;
      Engine *iDbEngine;
      TableContainer iTableCont;
      bool iAutocommit;
};

#endif
```

8. Table.h

```
#ifndef _TABLE_H
#define _TABLE_H

#include <string>

#include "global.h"
#include "database.h"

using namespace std;

class Column;
class Table {
   // Bevat de beschrijving van een tabel
   public:
      Table (Database&, string);
      inline ColumnContainer& getColumns ();
      inline string getTableName () const;
      void installColumn (Column *);
      // instellen en opvragen van het aantal tabellen met refererende
      // sleutels naar deze tabel
      inline void setRK(int);
      inline int getRK () const;
      // instellen en opvragen van het aantal tabellen waarnaar deze
      // tabel refereert
      inline void setFK(int);
      inline int getFK () const;
      ~Table () { }

   protected:
      string iTableName;
      ColumnContainer iColumnCont;
      // het aantal vreemde sleutels (naar verschillende tabellen) en het
      // aantal refererende sleutels (van verschillende tabellen) wordt
      // gebruikt voor het sorteren van de tabellen
      int iFKNr;
      int iRKNr;
};

#endif
```

9. Column.h

```
#ifndef _COLUMN_H
#define _COLUMN_H

#include <string>

#include "global.h"
#include "exception.h"
#include "table.h"

using namespace std;

class Column {
   // Bepaalt de positie van een kolom in een databank
   public:
      Column (Table&, string, bool, Column*, bool);
      void addReferencedKey (Column *);
      inline ColumnContainer& getReferencedKeys ();
      inline void setForeignKey(Column*);
      inline Column *getForeignKey() const;
      inline void setNull();
      inline bool isNull() const;
      inline bool isNullable() const;
      inline string getColumnName() const;
      inline string getTableName() const;
      inline string getQualifiedName() const;
      inline bool isPrimaryKey() const;
      inline bool isForeignKey() const;
      // Column moet virtueel zijn voor het gebruik van RTTI
      virtual ~Column () { }

   protected:
      Table& iTable;
      string iColumnName;
      bool iPrimaryKey;
      // NULL indien geen foreign key
      Column *iForeignKey;
      // NULL indien nergens naar dit veld gerefereerd wordt
      ColumnContainer iRKCont;
      bool iNullable;
      bool iIsNull;
};

#endif
```

10. DataCol.h

```
#ifndef _DATACOLUMN_H
#define _DATACOLUMN_H

#include <string>

#include "global.h"
#include "table.h"
#include "column.h"
#include "scond.h"
#include "datacond.h"
#include "condition.h"

using namespace std;

template<class T> class DataColumn : public Column {
   // Doorgeefluik van informatie tussen de databank en deze bibliotheek
   public:
      DataColumn (Table&, string, bool = false, Column * = 0,
                  bool = false);
      // volledige reference cycle wijzigen
      void setData (const T&);
      void setKeys (const T&);
      T getData() const;
      ~DataColumn() { }

      Condition operator< (T);
      Condition operator== (T);
      Condition operator!= (T);
      Condition operator> (T);
      Condition operator<= (T);
      Condition operator>= (T);

      Condition operator< (DataColumn<T>&);
      Condition operator== (DataColumn<T>&);
      Condition operator!= (DataColumn<T>&);
      Condition operator> (DataColumn<T>&);
      Condition operator<= (DataColumn<T>&);
      Condition operator>= (DataColumn<T>&);

      Condition operator! ();
      operator Condition ();

   protected:
      T iData;
};

#include "datacol.cpp"

#endif
```

11. SCond.h

```
#ifndef _SIMPLECONDITION_H
#define _SIMPLECONDITION_H

#include <string>

using namespace std;

class SimpleCondition {
   // Basisklasse van DataCondition
   public:
      SimpleCondition (string);
      inline string getOperator() const;
      inline bool isCondition() const;
      inline bool isNull() const;
      // gebruikt als virtuele kopie constructor
      virtual SimpleCondition* clone();
      // SimpleCondition moet virtueel zijn voor het gebruik van RTTI
      virtual ~SimpleCondition() { }

   protected:
      SimpleCondition (string, bool, bool);
      bool iCondition;
      bool iIsNull;
      string iOperator;
};

#endif
```

12. DataCond.h

```
#ifndef _DATACONDITION_H
#define _DATACONDITION_H

#include <string>
#include "scond.h"

using namespace std;

template<class T> class DataCondition : public SimpleCondition {
   // Formuleren van voorwaarden voor kolommen
   public:
      DataCondition(Column *, string, T);
      DataCondition(Column *, string);
      DataCondition(Column *, string, Column *);
      inline T getData() const;
      inline bool isColumnCondition() const;
      inline string getFirstName() const;
      inline string getSecondName() const;
      // gebruikt als virtuele kopie constructor
      SimpleCondition *clone();
      ~DataCondition() { }

   private:
      DataCondition(string, bool, bool, Column *, Column *, T);
      Column *iFirstColumn, *iSecondColumn;
      T iData;
};

#include "datacond.cpp"

#endif
```

13. Condition.h

```
#ifndef _CONDITION_H
#define _CONDITION_H

#include <string>

#include "global.h"
#include "scond.h"

using namespace std;

class Condition {
   // Verzameling van kolomvoorwaarden
   public:
      Condition() { }
      Condition(SimpleCondition *);
      Condition (Condition&);
      Condition& operator= (Condition&);
      inline CondContainer& getConditions();
      Condition& operator&& (Condition&);
      Condition& operator! ();
      Condition& operator|| (Condition&);
      ~Condition();

   private:
      CondContainer iCondCont;
      // gebruikt voor het opkuisen (destructor, assignatie)
      void removeall();
      // gebruikt voor het kopieren (copy constructor, assignatie)
      void makecopy (Condition&);
};

#endif _CONDITION_H
```

14. Storage.h

```
#ifndef _STORAGE_H
#define _STORAGE_H

#include <set>
#include <algorithm>
#include <string>
#include <iterator>

#include "global.h"
#include "database.h"
#include "engine.h"
#include "adapter.h"
#include "table.h"
#include "column.h"
#include "condition.h"
#include "exception.h"

using namespace std;

template<class T> class Storage {
   // User-interface voor databank toegang
   public:
      Storage (Database&, Adapter<T> *);
      Storage (const Storage<T>&);
      Storage& operator= (const Storage<T>&);
      bool insert(const T&);
      bool update(const T&);
      void erase(const T&);
      T& find(const T&);
      // herophalen van het laatst opgehaalde object
      T& find();
      void commit();
      void rollback();
      string getError();
      ~Storage ();

      class iterator;
      friend class iterator;

      // afgeleid van de STL input-iterator categorie
      class iterator : input_iterator<T, ptrdiff_t> {
         friend class Storage<T>;
         public:
            iterator() : iEndMarker(true),iStor(0),iEngineIterator(0) {}
            iterator(const iterator&);
            iterator& operator= (const iterator&);
            T& operator*() const;
            iterator& operator++();
            iterator operator++(int);
            bool operator==(const iterator&) const;
            bool operator!=(const iterator&) const;
```

```
            ~iterator();

         private:
            // private constructor
            iterator(Storage<T> *, Engine::iterator *);
            // gebruikt voor het kopieren (copy constructor, assignatie)
            void makecopy (const iterator&);
            // gebruikt voor het opkuisen (destructor, assignatie)
            void remove ();
            // leest het volgende object uit de SQL-cursor
            void read ();

            Storage* iStor;
            T* iData;
            Engine::iterator *iEngineIterator;
            // true als er geen data beschikbaar is
            bool iEndMarker;
      };

      iterator find(Condition);
      iterator begin ();
      iterator end();

   private:
      // bepaalt de gewenste volgorde van de tabellen
      struct tableComp {
         bool operator() (const Table *t1, const Table *t2) {
            if (t1 -> getFK() < t2 -> getFK())
               return true;
            else if (t1 -> getFK() > t2 -> getFK())
               return false;
            else
               return t1 -> getRK() > t2 -> getRK();
         }
      };

      void sortTables();
      T* iData;
      Database &iDatabase;
      Adapter<T> *iAdapter;
};

#include "storage.cpp"

#endif
```

Bibliografie

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it.
Samuel Johnson

[1] C.J. DATE, *An introduction to Database Systems*, Sixth edition, Addison-Wesley, 1995

[2] G. BOOCH, *Object-oriented analysis and design with applications*, Second edition, Benjamin-Cummings, 1994

[3] S. KHOSHAFIAN & R. ABNOUS, *Object orientation*, Second edition, John Wiley & Sons, 1995

[4] B. HENDERSON-SELLERS, *A Book of Object-Oriented Knowledge*, Second edition, Prentice Hall, 1997

[5] B. STROUSTRUP, *The C++ Programming Language*, Second edition, Addison-Wesley, 1991

[6] M. NELSON, *C++ Programmer's Guide to the Standard Template Library*, IDG Books, 1995

[7] D.S. LINTHICUM, *Objects meet data*, DBMS, 9, 10, 1996, pag. 72-78

[8] G. PREMEREUR, *Een object-georiënteerde klassenboom voor transacties op relationele databanken in een client/server omgeving*, Ingenieursthesis, R.U.G., Gent, 1994

[9] J. WARMER & A. KLEPPE, *Praktisch OMT*, Addison-Wesley, 1996

[10] E. GAMMA, R. HELM, R. JOHNSON & J. VLISSIDE, *Design Patterns : Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995