Designing Business Frameworks Using DeKlarit

June, 2006

Information in this document, including URLs and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, persons, places or events depicted herein are fictitious, and no association with any real organization, product or person is intended or should be inferred.
DeKlarit is a trademark or registered trademark of ARTech. All other trademarks contained herein are property of their respective owners.
Copyright © 2002, 2006 ARTech. All rights reserved.

Table of Contents

Introduction........ 3
What is a Business Framework?........ 3
  More on Business Frameworks 3
  DeKlarit: a RAD tool for .NET Business Frameworks........ 4
  Business Components 4
  Data Providers 5
Designing a Business Framework: Online Broker 6
  Online Broker's Business Components 6
  Identification Attributes........ 8
  Relating Business Components 9
  Data Modeling 10
  Business Rules........ 11
  Vertical Formulas 12
  Multi-level Business Components........ 13
  Data Modeling – Business Component Levels........ 14
  Nested & Parallel Levels 15
  Online Broker's Data Providers 16
  Multi-level Data Providers........ 18
  Organizing the Business Framework........ 20
Generating and Using the Business Framework 21
  Impact Analysis & Database Creation 21
  Code Generation 23
  Using the DeKlarit Business Framework Prototyper........ 26
  DeKlarit Add-Ins 29
  Incremental Development 30
Rules........ 31
  More on Rules........ 31
  Error 32
  Assign 33
  Default 33
  Static Method Call 34
  Add and Subtract 34
  Serial 34
Extended Table 35
Extended Table and Data Providers 37
More on Formulas........ 41
  Horizontal Formulas........ 41
  Vertical Formulas 42
  Built-in Methods........ 43
Data Types........ 44
Subtypes........ 45
  Multiple Relationships between Business Components........ 45
  Self-References........ 46
  Inferred Attributes........ 48
  Avoiding Unnecessary Referential Integrity Constraints........ 48
  Inheritance with Subtypes 51
  Implementing Inheritance 53
Physical Database Design 55
  Table Name........ 55
  Indexes 56

© Copyright 2002-2006 ARTech www.deklarit.com

Introduction

This is an introduction to the design of Business Frameworks using Visual Studio .NET and DeKlarit. It is aimed at developers who are new to DeKlarit, to guide them on the best-known principles of Business Framework design.

To avoid an abstract and impractical document, DeKlarit's best practices will be introduced during the development of a simplified real-world example: a Business Framework for an online broker. This sample application is distributed with DeKlarit in C# and Visual Basic versions, and is located at Program Files\DeKlarit \Samples. You can find more information about this application in the MinBroker topic of the program's Help file.

This paper focuses on the design aspects of a DeKlarit project and not on how to use DeKlarit. For more on this, read "Getting Started with DeKlarit and Visual Studio .NET." If you are interested in DeKlarit's theoretical justification, the reading of "An Introduction to DeKlarit" white paper is highly recommended. Both documents are available online at http://www.deklarit.com

Those interested in learning how to use a Business Framework instead of how to design it should read "Working with DeKlarit Components" and "Using Component Services with DeKlarit Components." For more advanced uses of DeKlarit Business Frameworks see "Working with DeKlarit Metadata." These documents can be found in the DeKlarit Help documentation.

What is a Business Framework?

After a slow start, Object Oriented Frameworks are becoming very popular. The best example of this is the .NET Framework itself.

Our vision is that Business Frameworks are the next step of this trend. But, what is a Business Framework?

Basically, a Business Framework is an information business model that represents the behavior and structure of a business without focusing on User Interface, Platform, etc.

More on Business Frameworks

A Business Framework must be domain specific and, in most cases, company specific. That is why building a Business Framework used to be so difficult.

Another important characteristic of the Business Framework is persistence. Businesses need to store large quantities of data, and the Business Framework defines the data to be stored and its structure. DeKlarit will store the data in a Relational DBMS: Microsoft SQL Server, SQL Server 2005 Mobile Edition, MySQL, Access, Oracle, or IBM DB/2.

The advantages of building three-tier applications are well-known in the software industry: the basic idea of three-tier applications is to isolate the business logic and data access code (business and data access tier) from the user interface code (presentation tier). The components that constitute this middle tier are responsible for validating the data and updating the database with the appropriate data-access technology.

This architecture has multiple advantages:

- It allows different kinds of user interfaces to reuse the same business logic (Browser clients, Windows clients, Cell phones, PDAs, etc).
- Database updates are centralized on server-side components that secure the data.
- Changes to the business rules only require you to update the server code, and not each client.
- Middle-tier components can run on an application server, which provides runtime services such as connection pooling, etc., to maximize the throughput and scalability of the applications.

We can say that a good Business Framework is a must for building successful three-tier architecture Applications. Microsoft .NET platform designers have addressed this issue and the .NET Framework provides the functionality to make Business Frameworks development and deployment easier.

Going back to the first question, a DeKlarit Business Framework is a business model composed of:

- A Database Schema
- A set of DataAdapters that will encapsulate the business logic and data access layers
- A set of DataSets to be used by presentation layer programmers

DeKlarit: a RAD tool for .NET Business Frameworks

DeKlarit can be easily described as an add-in that converts the Visual Studio .NET environment into a RAD (Rapid Application Development) tool for designing and building Business Frameworks.

With DeKlarit you define the Business Framework at the requirements level and it automatically generates the Business Layer, Data Access Layer, and Database schema.

You can define a DeKlarit Business Framework just by defining two types of components: Business Components and Data Providers.

Business Components

A Business Component is a set of information that needs to be stored in the Database. It usually —but not always— maps to a real-world entity such as a Customer, Invoice, etc. Business Components are designed to support transactional operations and their objective is to insert, update and delete their specific instances.

You can Insert, Update or Delete an instance of a Business Component, unless forbidden by its rules. A Business Component is defined by its Structure and Rules.

The Structure is composed by attributes. In the following example a Customer's Structure has two attributes: CustomerID and CustomerName.

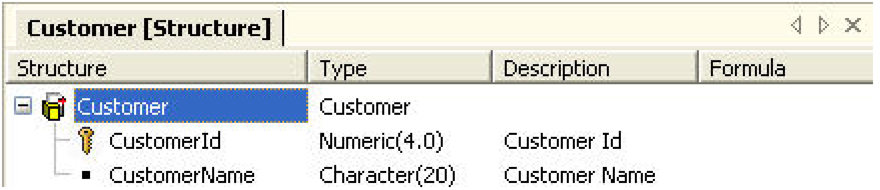


Figure 1

The before CustomerID means that there are no two Customers with the same CustomerID. In other words, CustomerID is the Primary Key of Customer.

DeKlarit will use this information to automatically design the correct Database Schema, in Third Normal Form.

A Business Component Structure can be more sophisticated, such as an Invoice that needs a two-level Structure, as shown in Figure 2:

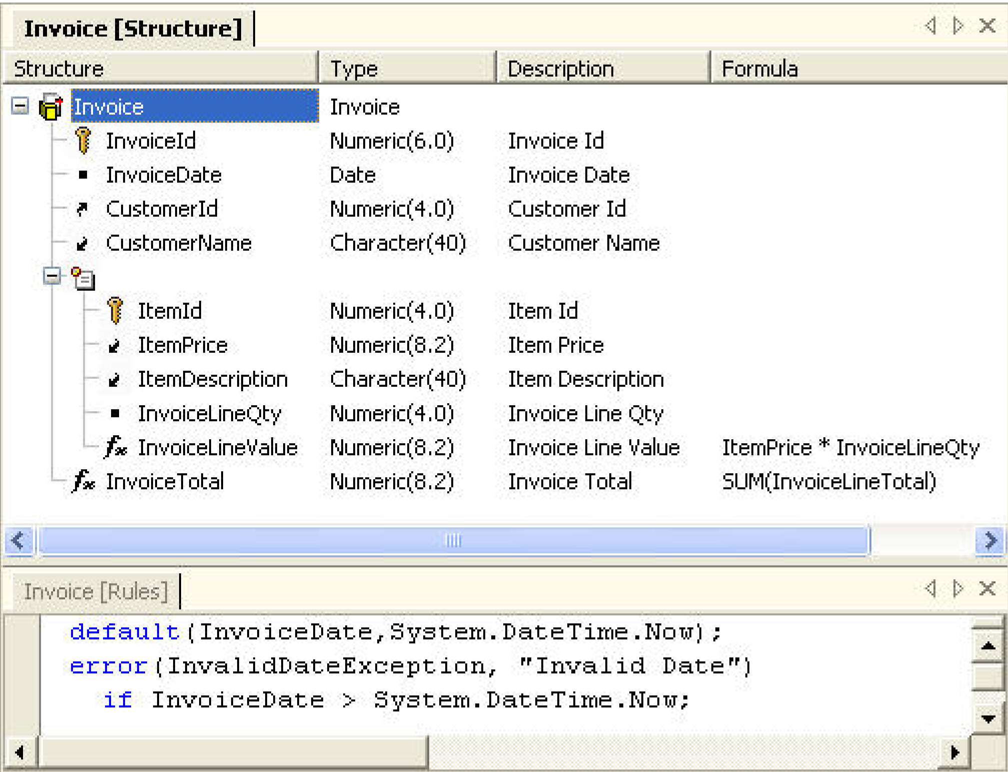


Figure 2

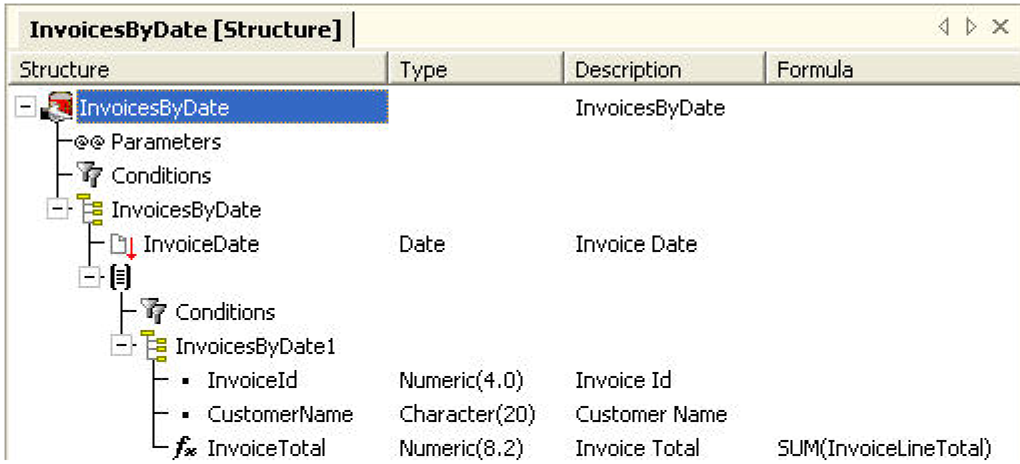
The ***fx*** symbol means the attribute is a Formula. For example, InvoiceTotal is the sum of InvoiceLineValue.

Business Component rules are a declarative way to define its behavior. For example, in the Invoice defined above there is a rule that forbids inserting an invoice with a future date.

When it comes to retrieving data, a different view is needed. Usually the data retrieval process can involve attributes from many Business Components with specific orders and conditions. DeKlarit allows you to retrieve data by using Data Providers, thus simplifying the data retrieval process.

Data Providers

When an end user requests data, this process usually involves retrieving attributes from different tables in a specific order and with certain conditions. The problem is that the end user ignores which tables contain the needed attributes or how to join those tables. However, by using the **DeKlarit Data Providers**, the request can be defined in a very user-friendly way.
You can do this by selecting the attributes needed as shown in the example of Figure 3.


InvoicesByDate [Structure]

Structure	Type	Description	Formula
InvoicesByDate		InvoicesByDate	
@@ Parameters			
Conditions			
InvoicesByDate			
InvoiceDate	Date	Invoice Date	
Conditions			
InvoicesByDate1			
InvoiceId	Numeric(4.0)	Invoice Id	
CustomerName	Character(20)	Customer Name	
InvoiceTotal	Numeric(8.2)	Invoice Total	SUM(InvoiceLineTotal)

Figure 3

In this example, the Data Provider will return a DataSet with the Invoice ID, Customer Name and Invoice Total grouped by InvoiceDate. DeKlarit will generate the DataSet with the structure and the DataAdapter to execute the SQL statements. Also, it will evaluate the appropriate formulas and load the resulting data into the DataSet.

As the Data Provider is specified without any reference to a specific database table, programs that use the generated .NET components do not need to be changed when the underlying database schema changes.

Designing a Business Framework: Online Broker

To introduce the new concepts around DeKlarit, we will develop a simplified version of a Business Framework for an Online Broker.

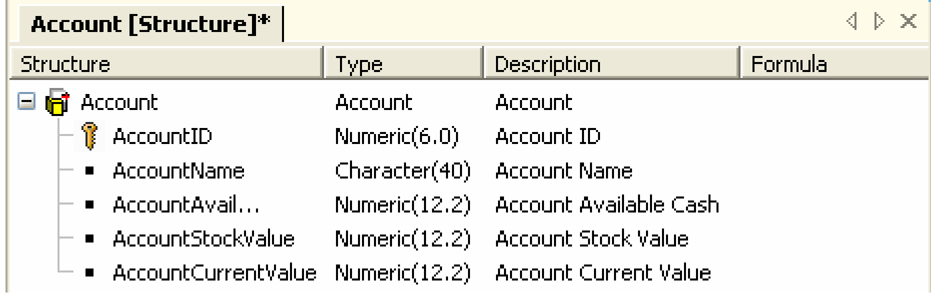
It is very important to start the Business Framework's design by defining its objective. It is surprising how different a Business Framework can be if you change its objective. In this case, the Business Framework's objective is to allow users to trade stock online.

Another element you should consider is the Essentiality Principle, an old design principle that in this case can be worded as follows:

A Business Framework should have only the essential features, but it should have all of them.

Online Broker's Business Components

The design process starts by defining Business Components. Here it is started by the most obvious one: Account. Its structure is as follows:


Account [Structure]*

Structure	Type	Description	Formula
Account	Account	Account	
AccountID	Numeric(6.0)	Account ID	
AccountName	Character(40)	Account Name	
AccountAvail...	Numeric(12.2)	Account Available Cash	
AccountStockValue	Numeric(12.2)	Account Stock Value	
AccountCurrentValue	Numeric(12.2)	Account Current Value	

Figure 4

The Account has five attributes:

- AccountID (since it has a , it identifies an Account. There are no two Accounts with the same AccountID) - A Numeric value with a maximum of 6 digits. We want AccountID to be numbered by the system, so we set its AutoNumber property to True.
- AccountName
- AccountAvailableCash - Amount of cash the Account has to buy stock.
- AccountStockValue - Current value of all the stock the Account has on hand.
- AccountCurrentValue - The sum of AccountAvailableCash and AccountStockValue.

In DeKlarit, AccountCurrentValue is a Formula and its expression is defined in the attribute's properties:

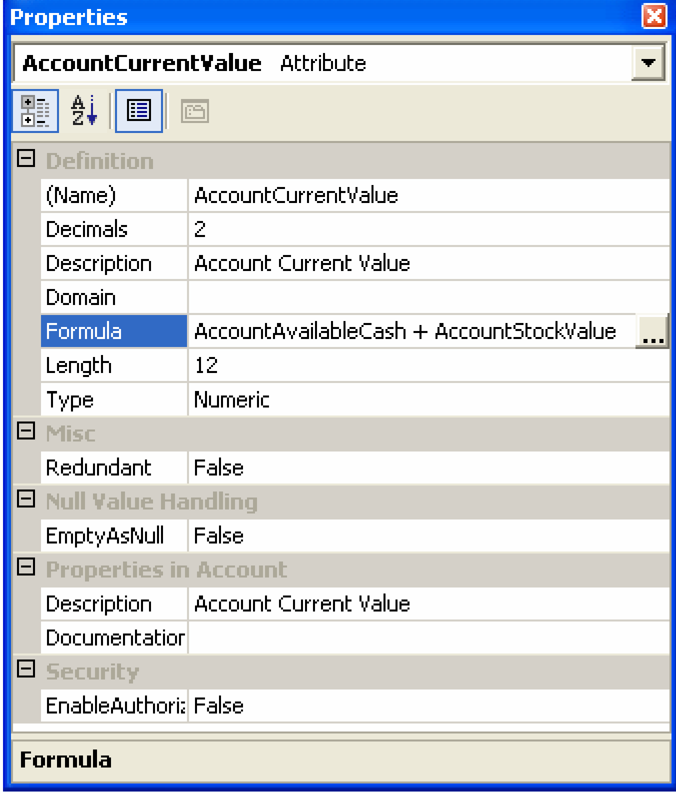


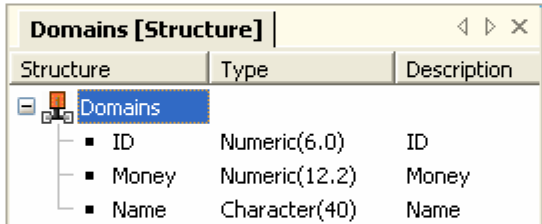
Figure 5

Another aspect to take into account is that AccountAvailableCash, AccountStockValue, and AccountCurrentValue all have the same type definition, N(12.2), which means a Numeric value of 12 and 2 decimals. If you want these attributes to always have the same definition, you need to use Domains.

A Domain is used to group common definitions for many attributes. For example, a new domain called Money is defined as N(12.2), and later the Domain property of AccountAvailableCash, AccountStockValue and AccountCurrentValue is set to Money. If the Domain is changed later, for example from N(12.2) to N(14.4), all the attribute definitions with the Money domain will change.

Although it is not necessary to use Domains in a Business Framework, its use is highly recommended. Some of the Domains used in this example are:

- ID - For autonumbered identification attributes.
- Name - For names, defined as C(40).
- Money - N(12.2).

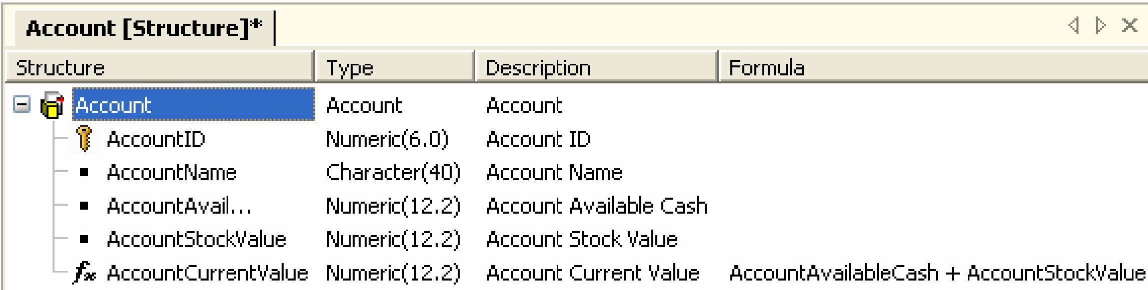


Domains [Structure]

Structure	Type	Description
Domains		
ID	Numeric(6.0)	ID
Money	Numeric(12.2)	Money
Name	Character(40)	Name

Figure 6

After these changes take place, the Account's structure will be as follows:



Account [Structure]*

Structure	Type	Description	Formula
Account	Account	Account	
AccountID	Numeric(6.0)	Account ID	
AccountName	Character(40)	Account Name	
AccountAvail...	Numeric(12.2)	Account Available Cash	
AccountStockValue	Numeric(12.2)	Account Stock Value	
AccountCurrentValue	Numeric(12.2)	Account Current Value	AccountAvailableCash + AccountStockValue

Figure 7

Identification Attributes

Every Business Component must have a set of identification attributes. Choosing the right ID attribute is a crucial step in the Business Component design, so you should be very careful about this area.

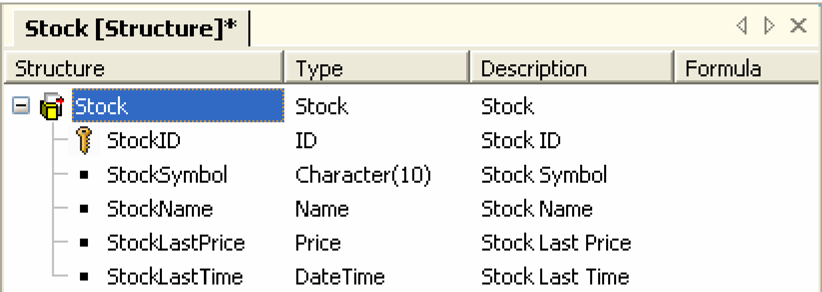
Identification attributes must have the following properties:

- They uniquely identify a Business Component instance.
- They cannot be changed.
- They have a value from the creation of the Business Component instance.
- They are the minimum set that complies with the other 3 properties. For example, AccountID identifies an Account, so AccountID and AccountName also identify an Account but they are not a minimum set.

We will discuss how to choose the ID attribute by creating a new Business Component called Stock. It stores information about the companies that can be traded through the system. The minimum information is the Stock Symbol (for example, MSFT for Microsoft), company name and pricing.

Which is the best identification for Stock? At first sight it seems that the Stock Symbol is a good one, as there are no two companies with the same symbol. But identification attributes have another restriction: they must remain unchanged over time. It is unusual for a company's Stock Symbol to change, but it may happen. For instance, E*Trade's symbol was changed from EGRP to ET.

If you cannot find a good set of identification attributes, the best idea is to create a new attribute and set its Autonumber property to true. In this case the new attribute is StockID.



Stock [Structure]*

Structure	Type	Description	Formula
Stock	Stock	Stock	
StockID	ID	Stock ID	
StockSymbol	Character(10)	Stock Symbol	
StockName	Name	Stock Name	
StockLastPrice	Price	Stock Last Price	
StockLastTime	DateTime	Stock Last Time	

Figure 8

Note that:

- A new Domain was created, Price –N(9.4)–, for stock pricing.
- StockLastTime is of DateTime type.

Sometimes one attribute is not enough to uniquely identify a Business Component instance. In this case set the to as many attributes as needed to identify an instance. Some examples of this case will be presented in the section on Data Modeling.

Relating Business Components

One of DeKlarit's key concepts is that you only need to focus on what the end user 'knows' about the Business Component. For example, an end user does not necessarily understand data relations or more abstract concepts, but he/she knows that a Trade must have an AccountID and AccountName in order to correctly identify who is doing the trade.

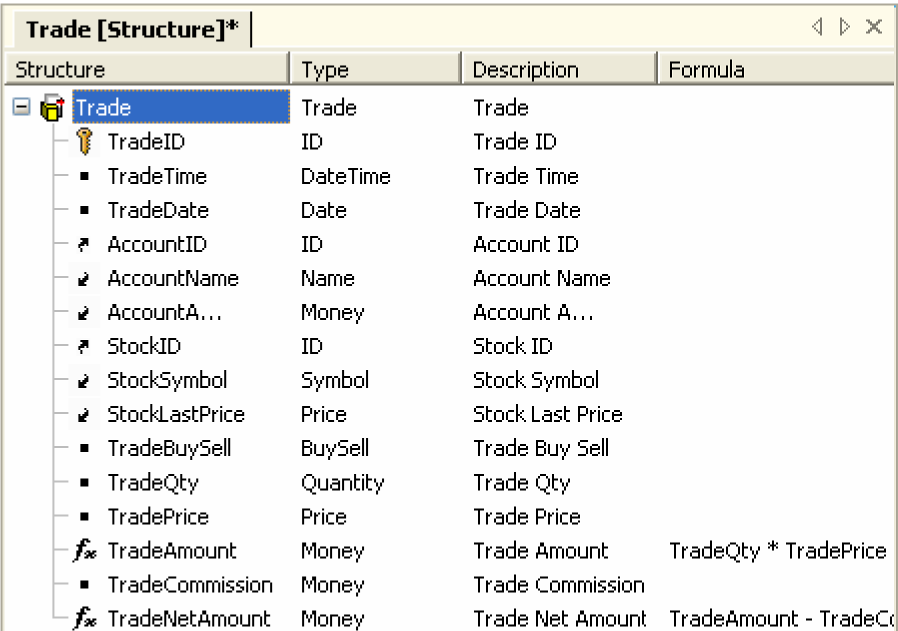
Trade is a more sophisticated Business Component. A Trade stores the information about an Account buying or selling certain Stock. For example, Account 123 bought 100 MSFT shares at 62.13 yesterday.

TradeID, TradeQuantity, and TradeTime will be needed. But how can a Trade relate to an Account? Here comes the magic: it is not necessary!

Tip: when designing a Business Component structure, always focus on the data the end users will interact with.

To insert one Business Component's attributes into another just drag it to the target component. The drag and drop operation copies the attributes of the first Business Component to the second one in the editor, so all attributes can be changed, removed, etc.

A Trade also has a Stock traded, so the Stock Business Component was dragged to Trade. The complete Trade structure is as follows:


Trade [Structure]*

Structure	Type	Description	Formula
Trade	Trade	Trade	
TradeID	ID	Trade ID	
TradeTime	DateTime	Trade Time	
TradeDate	Date	Trade Date	
AccountID	ID	Account ID	
AccountName	Name	Account Name	
AccountA...	Money	Account A...	
StockID	ID	Stock ID	
StockSymbol	Symbol	Stock Symbol	
StockLastPrice	Price	Stock Last Price	
TradeBuySell	BuySell	Trade Buy Sell	
TradeQty	Quantity	Trade Qty	
TradePrice	Price	Trade Price	
TradeAmount	Money	Trade Amount	TradeQty * TradePrice
TradeCommission	Money	Trade Commission	
TradeNetAmount	Money	Trade Net Amount	TradeAmount - TradeC

Figure 9

Now take a look at TradeNetAmount, which is a more sophisticated kind of formula, as shown in Figure 10:

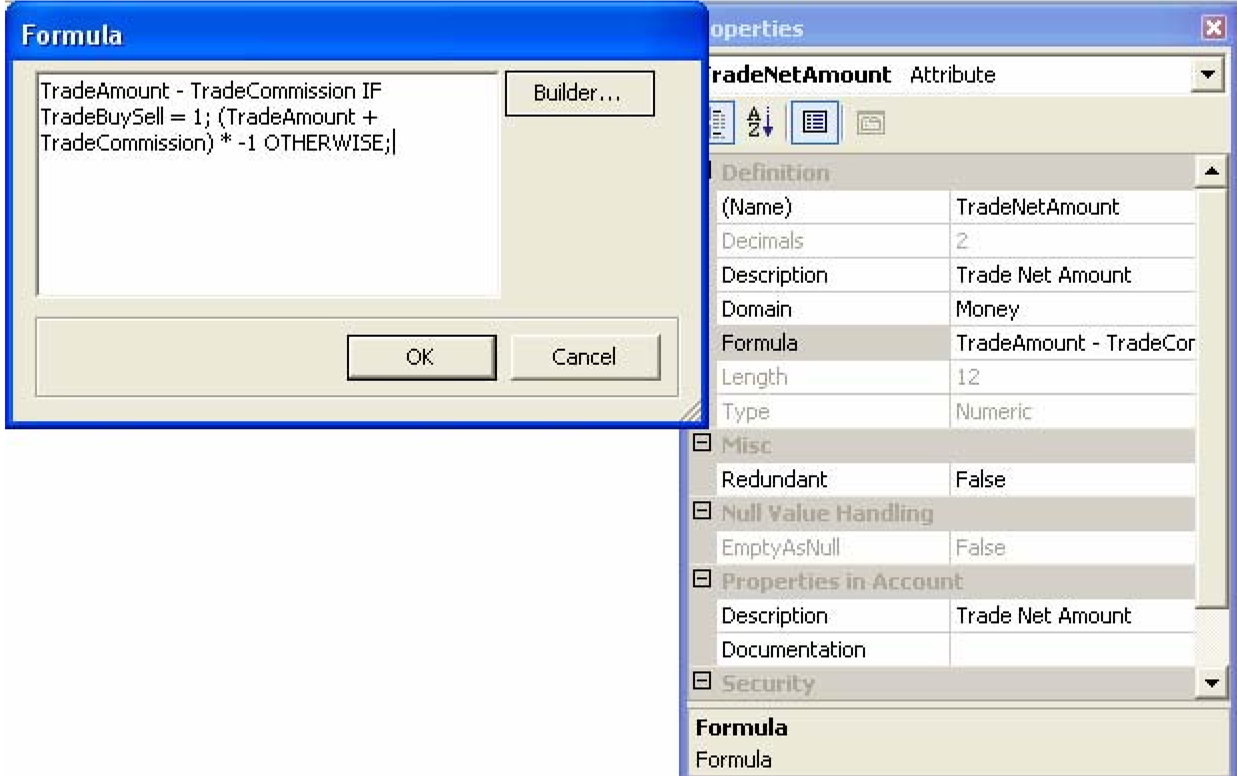


Figure 10

- It is a formula of a formula (TradeAmount is also a formula).
- It is a 'conditional' formula. It has many expressions, each one surrounded by a conditional expression (IF), so the value will correspond to the first expression whose conditional expression is true. The use of the OTHERWISE clause is recommended in case the other conditions are false.

Data Modeling

While the Business Components are being created, DeKlarit does all the Relational Data Modeling work, including Data Normalization to the Third Normal Form.

Given the Business Components defined for the Online Broker, the Relational Database Schema necessary to persist them (following the Relational Theory guidelines) is as shown in Figure 11:

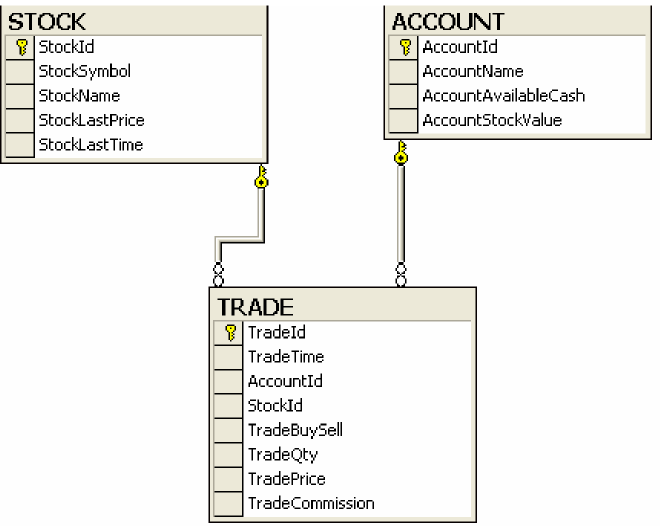


Figure 11

Some interesting points:

- DeKlarit automatically designs these tables.
- All the Primary Keys and Foreign Keys are also automatically defined.
- All the Referential Integrity Constraints will be **automatically** enforced. For example, it is not possible to insert a Trade whose AccountID is not present in the Account table. Also, it is not possible to delete an Account which has related Trades.
- The table's structure is different from its corresponding Business Component. For example AccountName is present in the Trade Business Component but not in the Trade table, due to data normalization.
- Formulas are calculated on the fly, not stored in the tables by default. See the Advanced Concepts chapter on Formulas to learn how to persist them (although this is not needed it can yield better performance).
- DeKlarit uses the Universal Relation Assumption, which states that one attribute should have the same name wherever it is used. For example, the account name is defined as AccountName in the Account Business Component **and** in the Trade Business Component.
- Database modeling takes place as Business Components are designed.

Business Rules

While the Business Component Structure defines which attributes will be stored, the Business Component Rules define its behavior. Surprisingly, only a few types of Rules are necessary to set the Business Component behavior.

One of the most commonly used rules is Error, which defines the condition under which the values are invalid. For example, a Trade is invalid if there is not enough money in the account to buy the stock. The DeKlarit rule is:

```
Error(NotEnoughStockException, "Not enough cash available to buy the stock") if
AccountAvailableCash < 0 and TradeBuySell = -1;
```

TradeBuySell was defined as having two possible values, -1 for Buy and 1 for Sell because this is very useful for formulas, as you can see in the section about Formulas.

Another common use of the Error rule is to forbid certain actions. By default you can Insert, Update or Delete an instance of the Business Component, but if for some reason you want to forbid some of them (such as updating or deleting a Trade) you can use:

```
Error( UpdateOrDeleteDisabled, "Update or Delete not allowed") if update or delete;
```

The Add rule is also very useful. Every time a sale trade takes place you want to increase the account's available cash, and decrease it in case of a buy trade:

```
Add(TradeNetAmount, AccountAvailableCash);
```

In this rule the action is taken into account. When inserting, it adds the value of the first attribute to the second; when deleting, it subtracts the value; and when updating, it adds the difference between the old and the new values.

Remember that to be used in a Rule, an attribute must be present in the Business Component's structure.

Rules are declarative, which means that the order they are written in is not important. Instead, the order is defined by the data dependencies. In the following example, the first rule will be triggered after the third one because the Add rule changes the value of an attribute used in the Error rule.

Trade [Structure] | Trade [Rules]

```
Error(NotCashException, "Not enough available cash to buy the stock")
    if AccountAvailableCash < 0 and TradeBuySell = -1;
Error(UpdateDeleteException, "Update or Delete not allowed")
    if insert or delete;
Add(TradeNetAmount, AccountAvailableCash);
```

Figure 12

Referential Integrity Constraint Rules do not have to be explicitly defined because they are inferred by DeKlarit.

Vertical Formulas

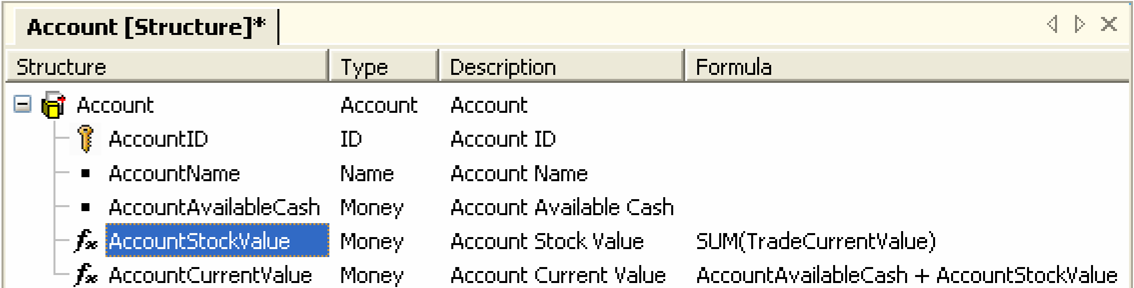
To add a new feature to the Business Framework —the current stock value of each Account— it is necessary to define a new formula in the Trade Business Component:

```
TradeCurrentValue = TradeQty * StockLastPrice * TradeBuySell * -1;
```

Note that TradeCurrentValue is positive if the Trade is a purchase, and negative if it is a sale. That is why coding TradeBuySell as -1 or 1 is practical, because otherwise conditions should be used:

```
TradeCurrentValue = TradeQty * StockLastPrice if TradeBuySell = -1;
TradeCurrentValue = TradeQty * StockLastPrice * -1 if TradeBuySell = -1;
0 otherwise;
```

The Account current stock value is the sum of TradeCurrentValue for all the Account's trades. In DeKlarit, this is expressed by defining it in the Account Business Component:


Account [Structure]*

Structure	Type	Description	Formula
Account	Account	Account	
AccountID	ID	Account ID	
AccountName	Name	Account Name	
AccountAvailableCash	Money	Account Available Cash	
AccountStockValue	Money	Account Stock Value	SUM(TradeCurrentValue)
AccountCurrentValue	Money	Account Current Value	AccountAvailableCash + AccountStockValue

Figure 13

AccountAvailableCash is a Vertical Formula because it aggregates the values of an attribute from another table, in this case TradeCurrentValue from the Trade table.

DeKlarit already knows the relationship between the Account and Trade tables, so it is not necessary to explicitly say that AccountStockValue is the sum of only the Account's trades and not all the trades.

The most common Vertical Formulas are SUM and COUNT. See the Formulas section for more on this.

Multi-level Business Components

One more feature to be added to the Broker Business Framework is a record of daily Stock prices. You can do this by defining a new Business Component called StockQuote.

StockQuote should include StockID, StockSymbol, StockQuoteDate, StockQuoteVolume StockQuoteLastPrice, StockQuoteAskPrice, StockQuoteBidPrice, StockQuoteHighPrice, StockQuoteLowPrice, etc. This is a valid, but somewhat strange, Structure because there are multiple prices and one attribute for each price type. A better way to express this is by using a multi-level Business Component:

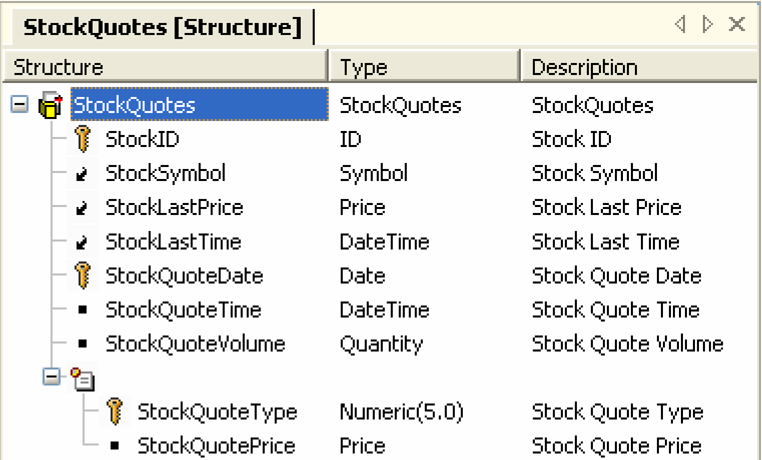


Figure 14

This means that a StockQuote instance only has one StockID, StockSymbol, StockLastPrice, StockLastTime, StockQuoteDate, and StockQuoteVolume, but multiple StockQuoteType; each one has a StockQuotePrice.

This structure is more flexible because you will be able to add new price types, such as MidDayPrice, without changing the Business Component structure, and therefore, the Database Schema.

The first level needs a Primary Key, and so do the rest of the levels. Given an upper level instance, there are no two instances of that level with the same Primary Key attribute values. In this case, in a StockQuote instance there is only one line with the StockQuoteType value.

Another interesting thing about this structure is that the first level has two Primary Key attributes (StockID and StockQuoteDate), meaning that there are no two instances of StockQuote with the same values of StockID AND StockQuoteDate. Or, in Business Framework-oriented words, for each Stock we store only one set of prices per day.

What should you change in this structure so as to be able to store more than one daily value?

Although the multi-level Business Component is a powerful concept, it sometimes misleads designers into making one of the most common design errors: mixing two Business Components in one just because they are related.

For example, instead of creating the Account and the Trade Business Components an inexperienced designer could define only one, as shown in Figure 15.

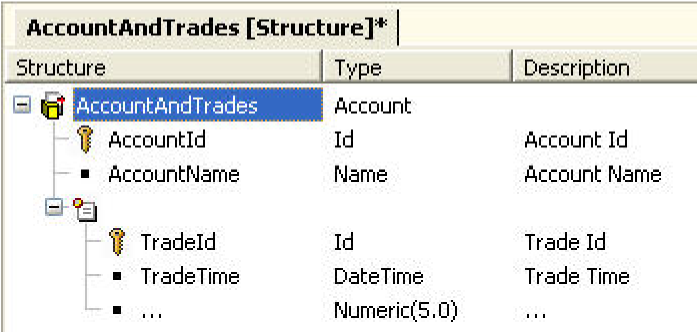


Figure 15

This structure is wrong because it does not reflect the Broker's reality. Remember that a Business Component is a chunk of information that can be Inserted, Updated or Deleted **together**. In this case, the structure says an Account and its Trades are Inserted/Updated/Deleted at the same time, which is clearly inexact. Account and Trades are related, but they are created at different times.

So, to store more than one daily value you should change the StockQuoteDate type from Date to DateTime.

Data Modeling – Business Component Levels

DeKlarit defines a Table for each Business Component level. The Primary Key for the subordinate table is formed by the upper level Primary Key plus the level Primary Key.

For example, the Database Schema for StockQuote is:

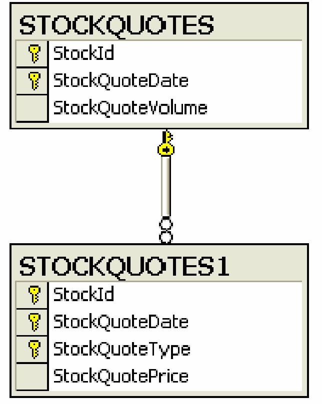


Figure 16

The complete Database Schema is shown in Figure 17.

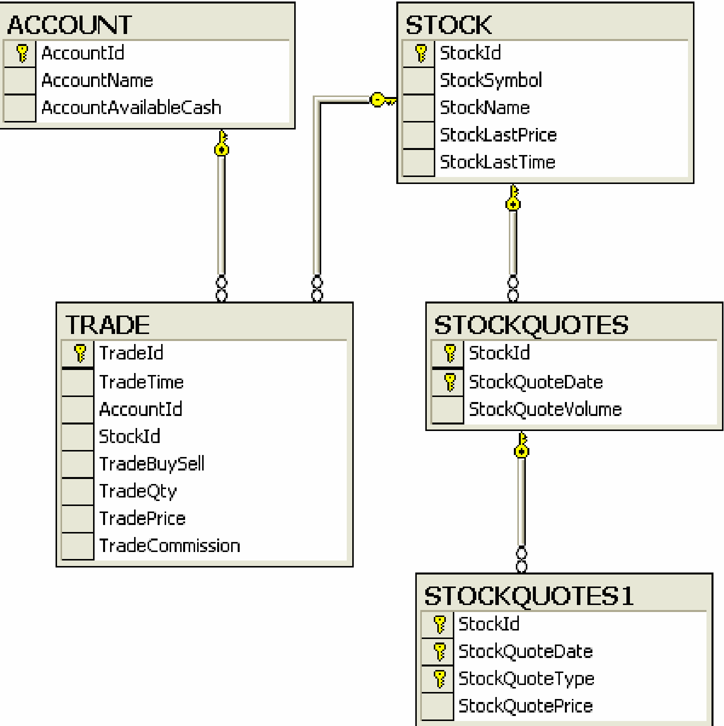


Figure 17

Note: An attribute with the Autonumber property set to true will be autonumbered only in the table where the attribute is the full Primary Key, as DeKlarit guarantees there is only one such table. For example, even though StockID is present in four tables —it is part of the Primary Key in three of them— it will be autonumbered only in the Stock Table.

Nested & Parallel Levels

Even though structures with more than two levels are not common, DeKlarit supports nested and parallel levels.

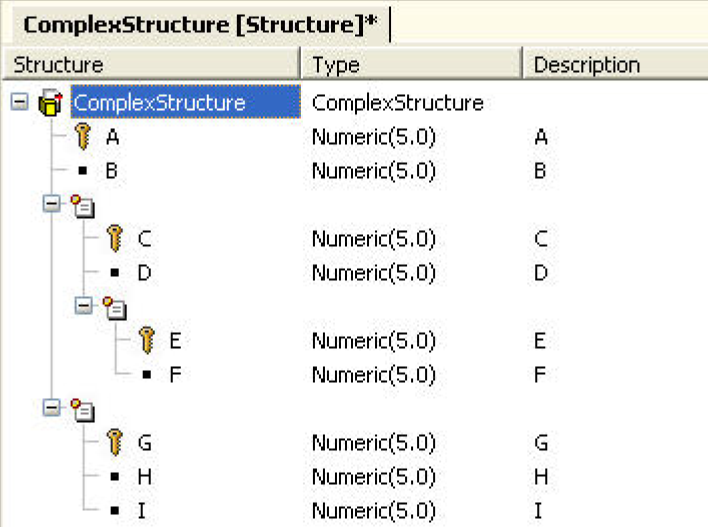


Figure 18

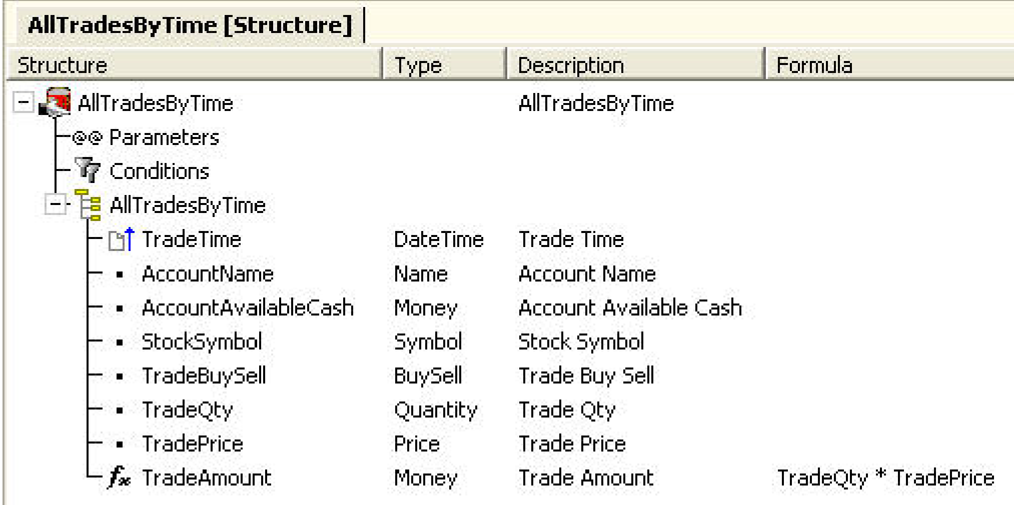
Remember that the structure should have only one first level.

Online Broker's Data Providers

The concept of Data Providers is important when designing a Business Framework with DeKlarit. Data Providers are read-only elements used to define which information can be extracted from the Business Framework.

They are used exclusively to query the Database, and the only way to update the data is through Business Components.

Like Business Components, Data Providers also have a Structure, only theirs defines the attributes to be retrieved. In the following example there is a Data Provider to list all Trades ordered by TradeTime:



AllTradesByTime [Structure]

Structure	Type	Description	Formula
AllTradesByTime		AllTradesByTime	
Parameters			
Conditions			
AllTradesByTime			
TradeTime	DateTime	Trade Time	
AccountName	Name	Account Name	
AccountAvailableCash	Money	Account Available Cash	
StockSymbol	Symbol	Stock Symbol	
TradeBuySell	BuySell	Trade Buy Sell	
TradeQty	Quantity	Trade Qty	
TradePrice	Price	Trade Price	
TradeAmount	Money	Trade Amount	TradeQty * TradePrice

Figure 19

- This Structure Editor is quite similar to the Business Component Structure Editor, but in this case instead of using the to define uniqueness, arrows are used to define the order in which the data will be presented: the up arrow ( ) for ascending order and the down arrow ( ) for descending order. If more than one attribute has an arrow, precedence is determined by the order of these attributes in the Structure.
- Data Providers can only use already defined attributes; they cannot be used to create new ones. The only way to define new attributes is through Business Components.
- Formulas can be used as any other attribute and it is not necessary to define the formula expression again.
- Drag and Drop is allowed, so it is very practical to drag a Business Component to the Data Provider, and select the attributes needed. In this example the Trade Business Component was used.

You do not need to worry about how to obtain the data, as this is DeKlarit's task. For example, in this Data Provider three tables should be joined to obtain the data: Account (because of AccountName), Stock (StockSymbol) and Trade. DeKlarit knows how to join the tables, and if there is no way to relate a set of attributes it will show an error message. In order to be related, all the level's attributes must be present in what is called the Extended Table.
Data can be filtered using Conditions. Suppose this is a Data Provider for today's Trades:

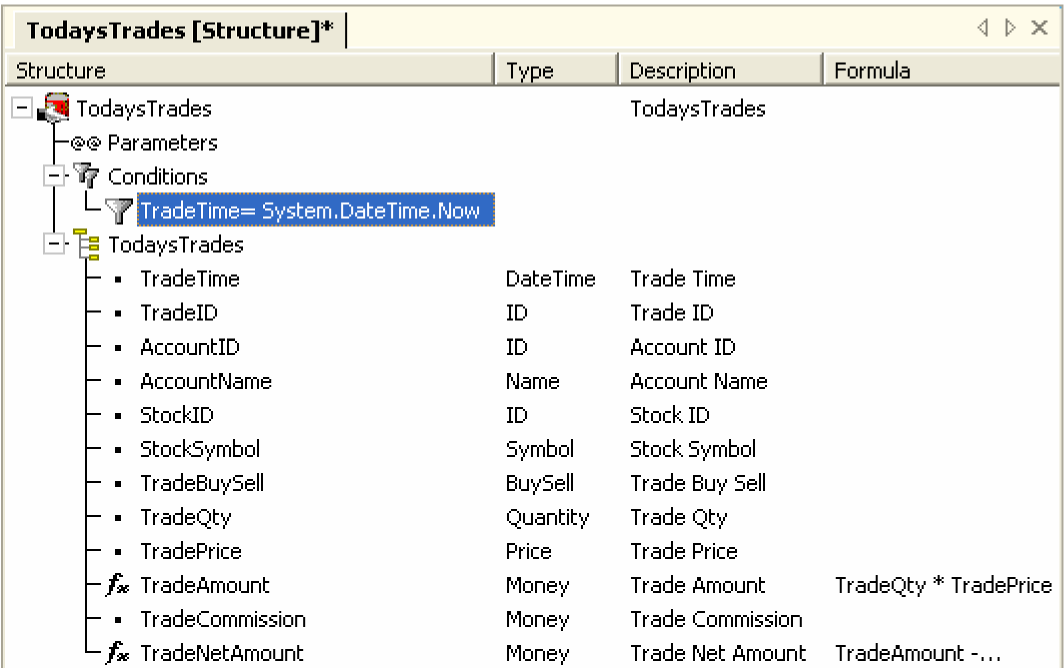


Figure 20

A more flexible Data Provider can allow the date used as a filter to be passed as a Parameter.

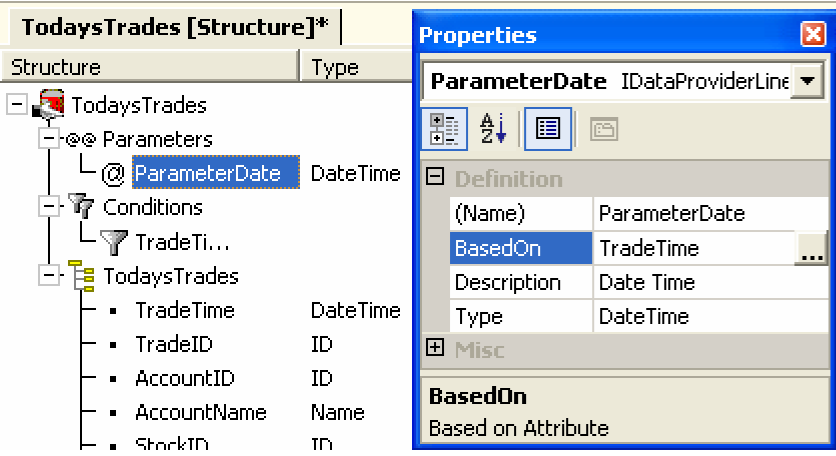


Figure 21

The type of ParameterDate can be defined as usual, but it is better to define it using the **Based On** property. In this case ParameterDate is based on TradeTime, so if you later change TradeTime's type, DeKlarit will change ParameterDate's type automatically. To differentiate a Parameter from an attribute, parameter names are prefixed by @.

The corresponding condition is shown in Figure 22.

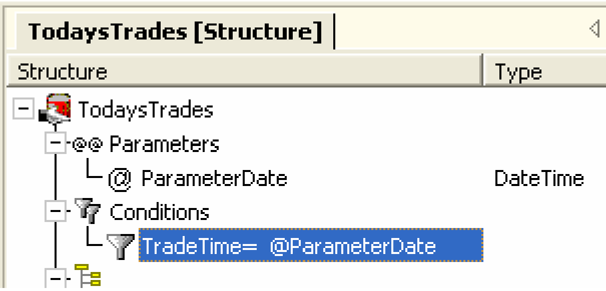


Figure 22

The Conditions syntax is quite simple:

- Logical operators: AND, OR and NOT
- Relational operators: =, <, <=, >, >=, <> and LIKE
- Operands: attributes, parameters and methods.

The LIKE operator is useful for searching simple patterns in character strings, for example, AccountName LIKE @SearchString. The % character can be used as a wildcard:

- AccountName LIKE "AA%" will match any occurrence of AccountName starting with AA.
- AccountName LIKE "%AA%" will match any occurrence of AccountName with AA anywhere.
- AccountName LIKE "%AA" will match any occurrence of AccountName ending with AA.

Multi-level Data Providers

Data Providers can have nested and parallel levels like Business Components. This is a very convenient way to list the Trades grouping them by date:

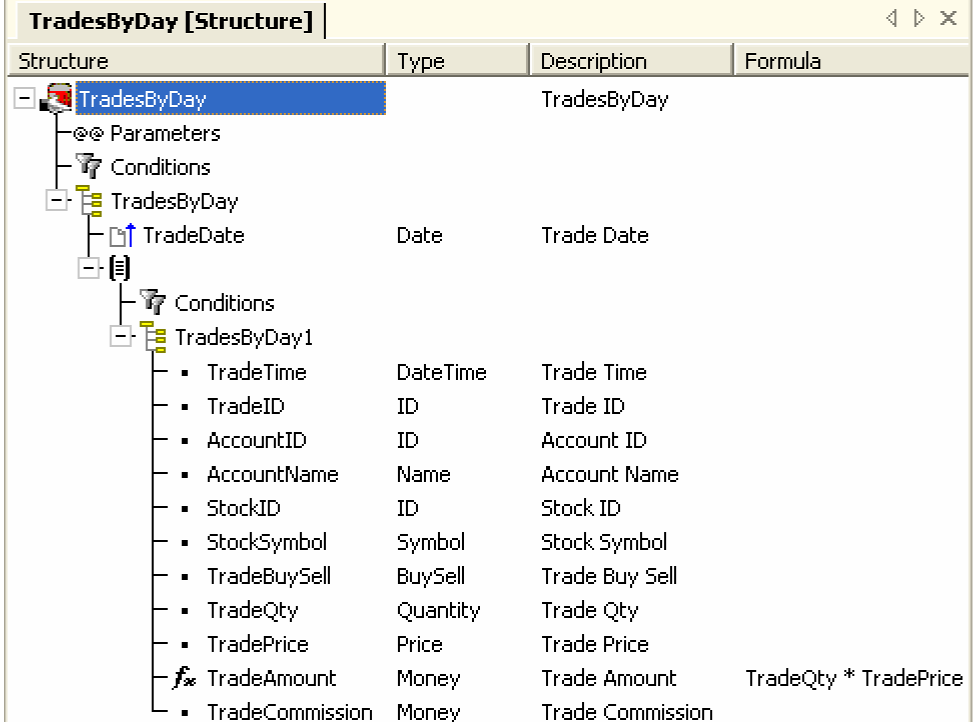


Figure 23

If you add two parameters like @InitialTradeDate and @EndTradeDate, and define the following condition:

TradeDate >= @InitialTradeDate and TradeDate <= @EndTradeDate

The first two Data Providers will not be necessary and, according to the Essentiality Principle, you should delete them.

Each level has its own Condition; for example, a Condition can be defined in the second level to list only the Trades greater than 10,000 dollars:

Figure 24

Data Providers can have nested and parallel levels:

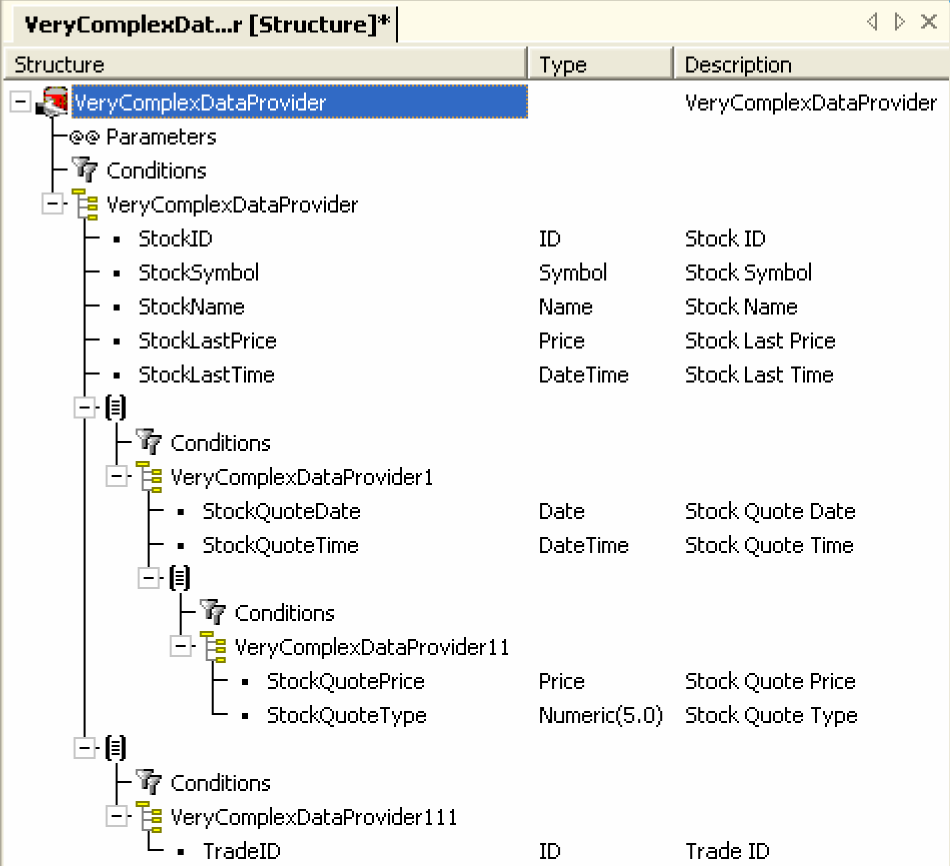


Figure 25

Note that even though this Data Provider seems to be impractical, it is functional because it will list each Stock's daily prices and also all its trades.

Organizing the Business Framework

A Business Framework can have a large number of Business Components and Data Providers. DeKlarit provides a Folder feature to organize them.

In this case a Stock folder was created to group all the Stock related components:

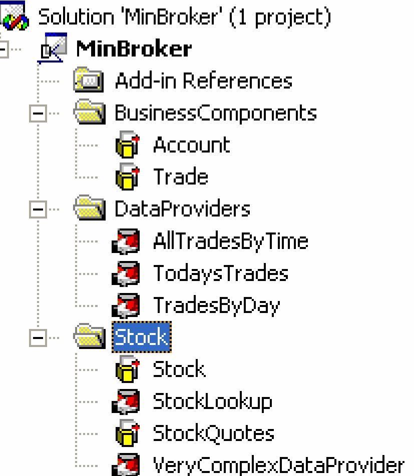
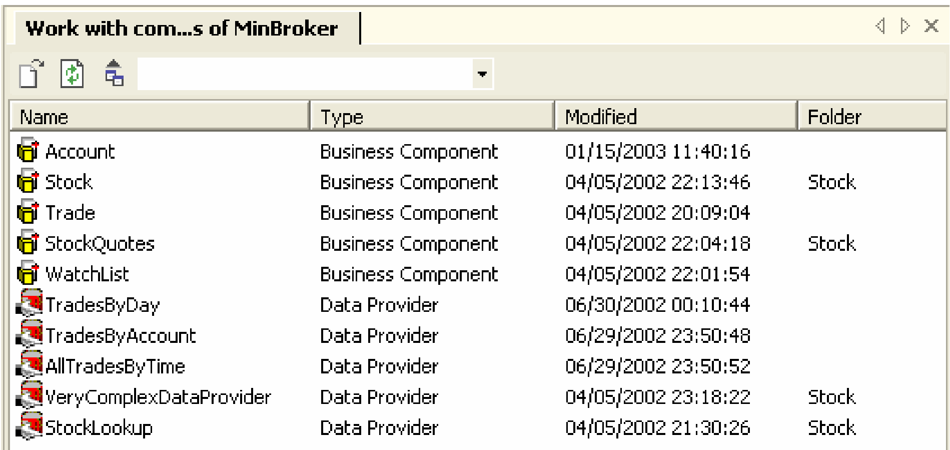


Figure 26

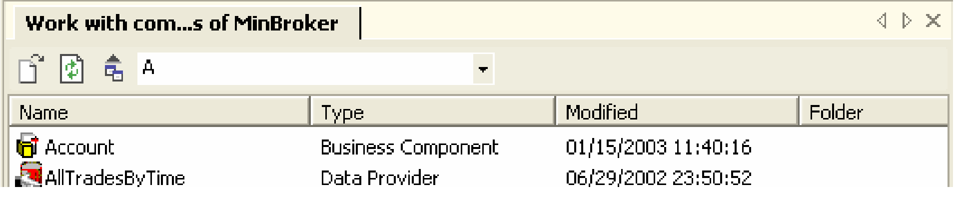
A Work With Components is also provided:



Name	Type	Modified	Folder
Account	Business Component	01/15/2003 11:40:16	
Stock	Business Component	04/05/2002 22:13:46	Stock
Trade	Business Component	04/05/2002 20:09:04	
StockQuotes	Business Component	04/05/2002 22:04:18	Stock
WatchList	Business Component	04/05/2002 22:01:54	
TradesByDay	Data Provider	06/30/2002 00:10:44	
TradesByAccount	Data Provider	06/29/2002 23:50:48	
AllTradesByTime	Data Provider	06/29/2002 23:50:52	
VeryComplexDataProvider	Data Provider	04/05/2002 23:18:22	Stock
StockLookup	Data Provider	04/05/2002 21:30:26	Stock

Figure 27

Here you can filter by name or order by any column, etc.



Name	Type	Modified	Folder
Account	Business Component	01/15/2003 11:40:16	
AllTradesByTime	Data Provider	06/29/2002 23:50:52	

Figure 28

Generating and Using the Business Framework

Once you have created the Business Components and Data Providers, as well as their rules, you should put DeKlarit to work.

DeKlarit's job is to:

- Generate and run the script to physically create tables in the selected database.
- Create a new Visual C# or Visual Basic project containing all the DeKlarit generated code, called Business Framework Project.
- Generate DataSet and DataAdapter classes to support the Framework.

Then you can:

- Use the DeKlarit Business Framework Prototyper to interact with your Business Framework.
- Use DeKlarit Add-Ins to generate an ASP.NET or Windows Forms application to work with the Business Framework.

The following sections explain each step in more detail.

Impact Analysis & Database Creation

Before creating the Database, the Solution only has a DeKlarit project. To generate and run the scripts needed to create the database, right-click the Business Framework item (MinBroker in this case) and choose **Create Database Tables**.

DeKlarit will first ask for the database to be used. Once the database is set, DeKlarit generates the script and executes it. Then it displays the following dialog box:

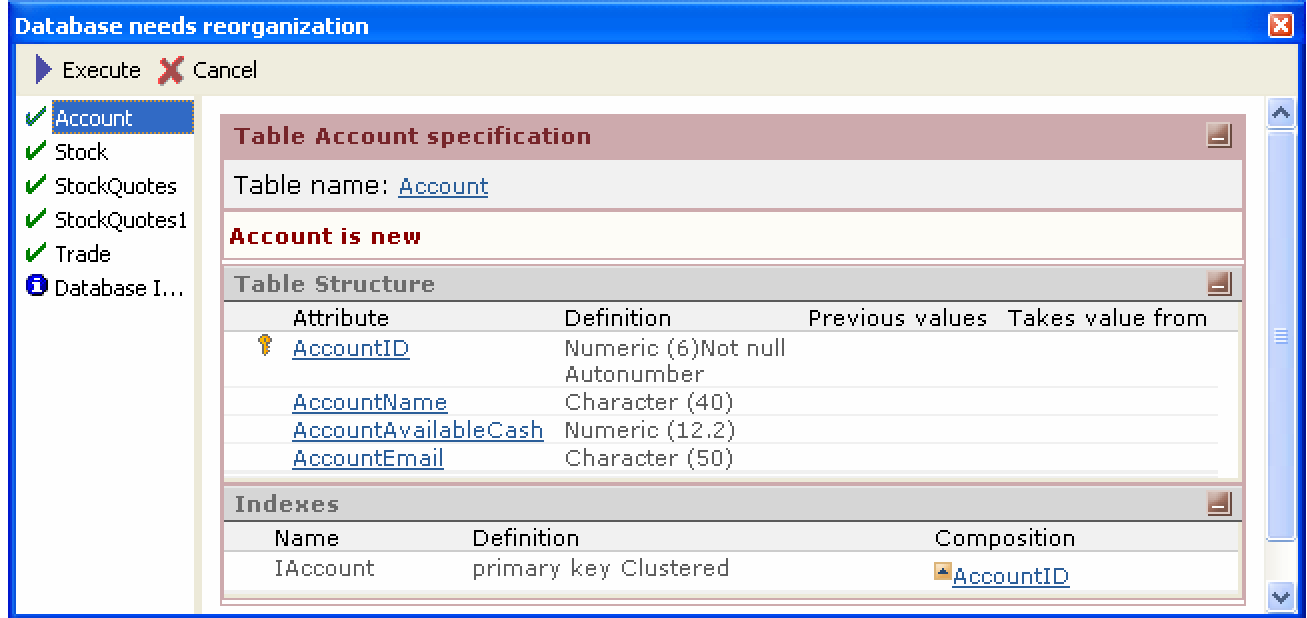


Figure 29

After you press the 'Execute' button, the script is run. The output window looks as shown in Figure 30.

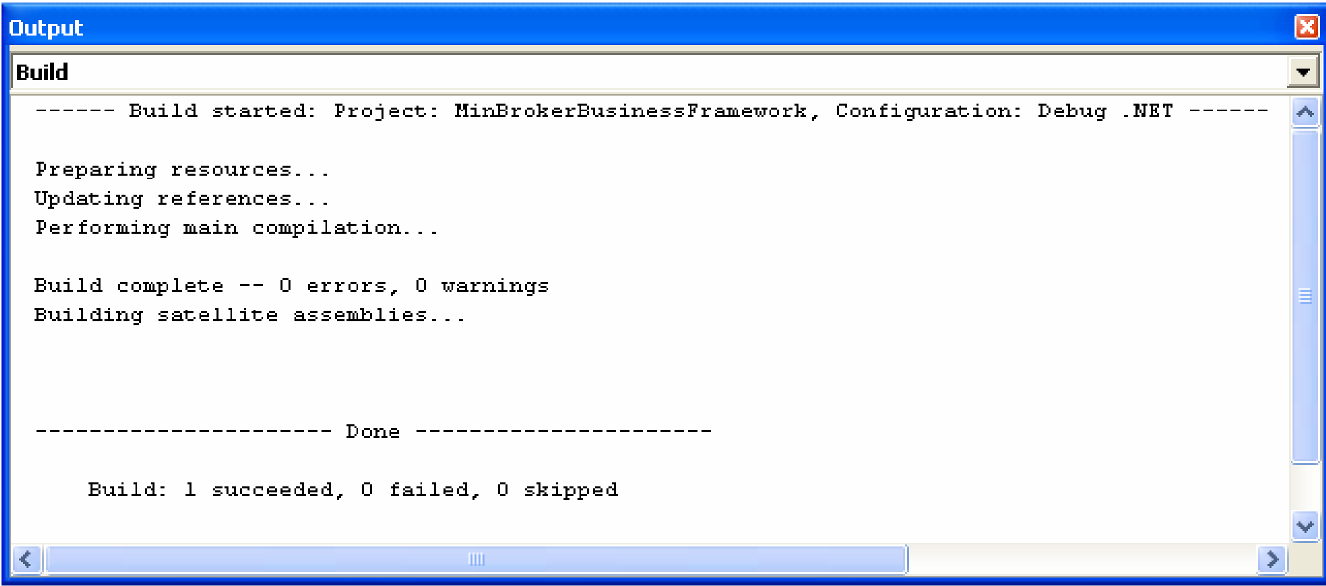


Figure 30

At this point all the tables are physically created in the corresponding SQL Server Database.

DeKlarit also shows an Impact Analysis Report with more detailed information about each table:

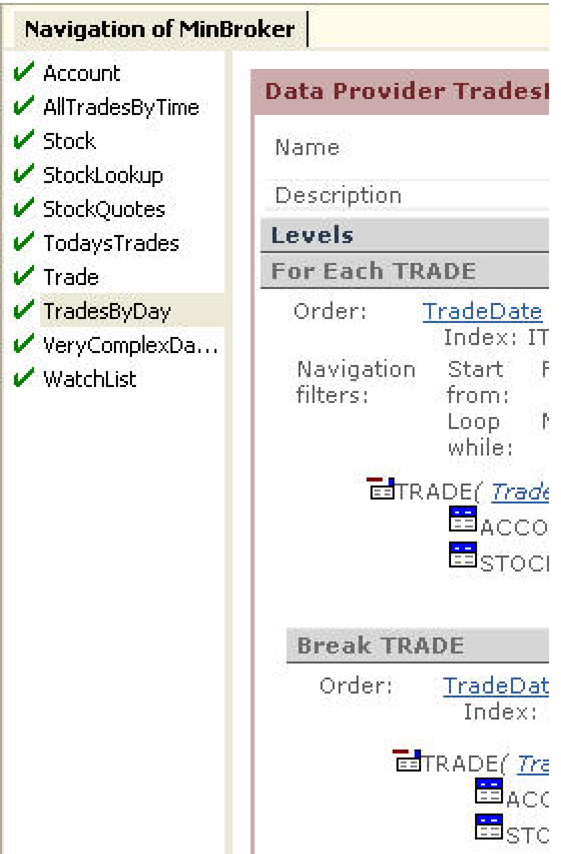


Figure 31

Code Generation

After creating the Database, the first time you Build the project DeKlarit generates a new project named <DeKlarit Project Name>BusinessFramework (in this case, MinBrokerBusinessFramework). Then it generates a DataSet and DataAdapter class for each Business Component or Data Provider defined in this new project.

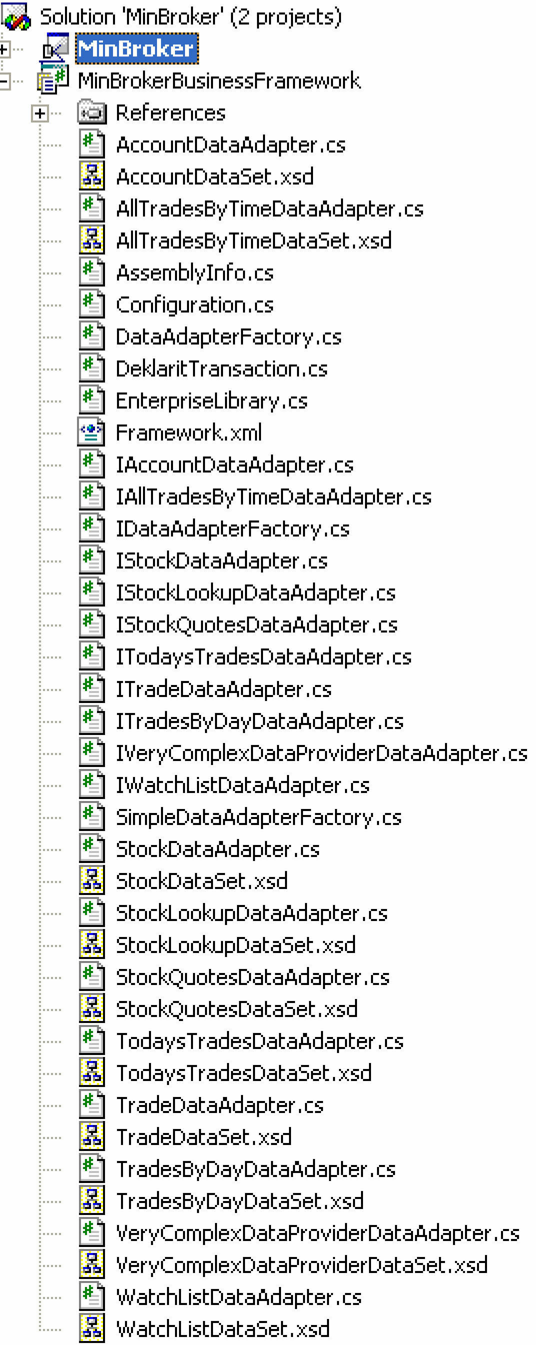


Figure 32

It will also provide a report with information about each program generated, as shown in Figure 33.

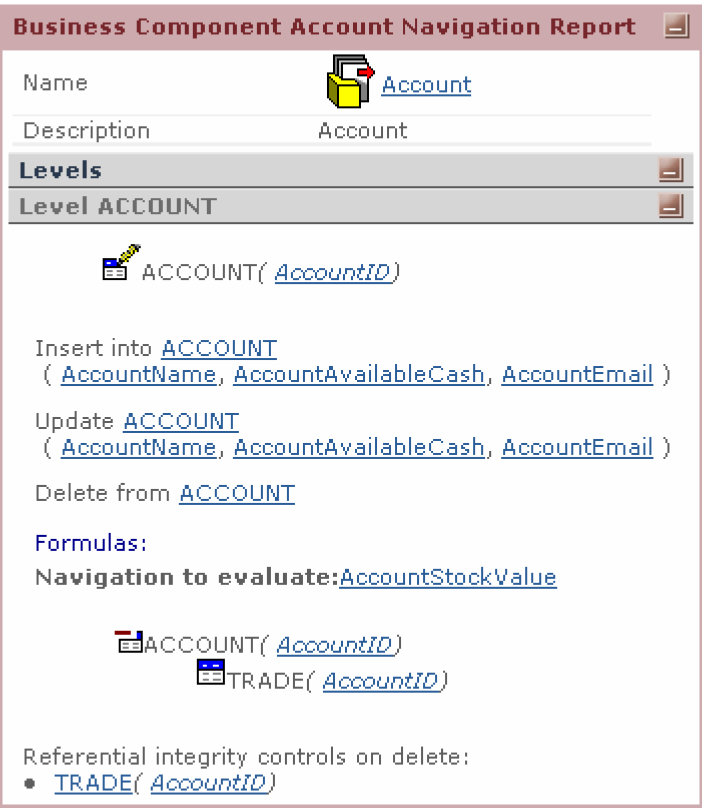


Figure 33

The objective of this report is to give you a detailed list of what the generated program will do. This is part of the information shown in a Business Component navigation report for each level:

- List of tables being used. The first table is the level's base table (see Extended Table for more details). The others are the tables used in the referential integrity constraints and to obtain the necessary attributes.
- List of attributes being inserted/updated/deleted for each table.
- Vertical formulas. It shows the navigation needed to evaluate these formulas.
- Referential integrity constraints on delete.
- See the Navigation Report for a multi-level Business Component (StockQuotes) in Figure 34.

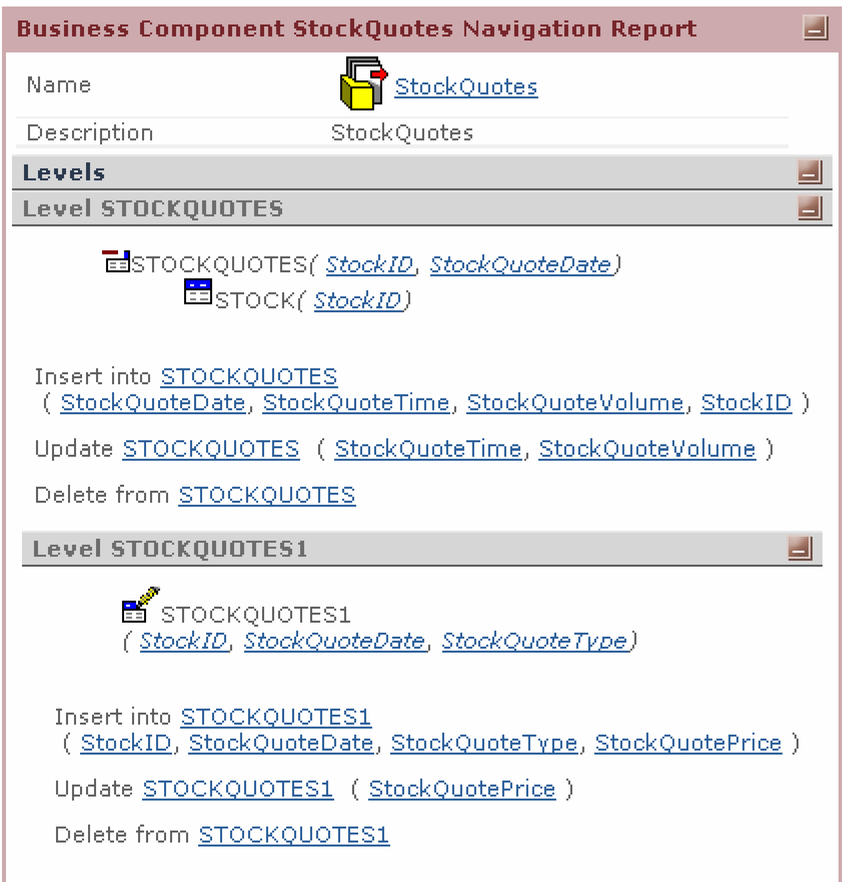


Figure 34

The Data Providers Navigation Report is similar, only that the focus is on the navigation of each level (tables to join, conditions, vertical formulas, etc.) and the relationship between levels.

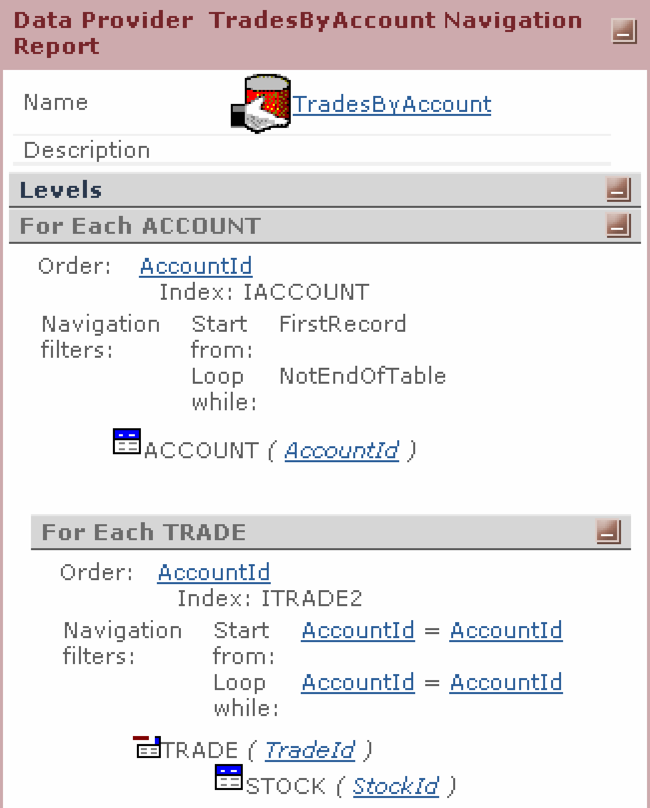


Figure 35

So as to let you focus on the Business Framework itself, DeKlarit has a powerful inference engine. These inferences are displayed in the Navigation report. In the above example some of them are:

- The relationship between the first and second level is through AccountID
- A join between Trade and Stock is needed in the second level.

At this point, the framework is ready to use.

From the programmers' point of view, a DeKlarit Business Framework is just a set of DataSets and DataAdapters that can be manipulated as standard ADO.NET objects. Therefore, you do not need to know anything new to use it.

Do not forget to Build the Business Framework project before using it in other projects.

The Build option checks all the components and generates only those needed. Use the Rebuild option to regenerate the whole Business Framework from scratch.

Using the DeKlarit Business Framework Prototyper

DeKlarit provides a tool that lets you prototype your Business Framework inside Visual Studio. To execute it, select the DeKlarit project in the Solution Explorer and press F5 on your keyboard, or select the **Start** option in the **Debug** menu:

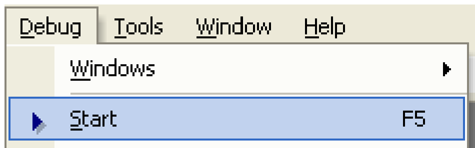


Figure 36

The DeKlarit Business Framework Prototyper will appear.

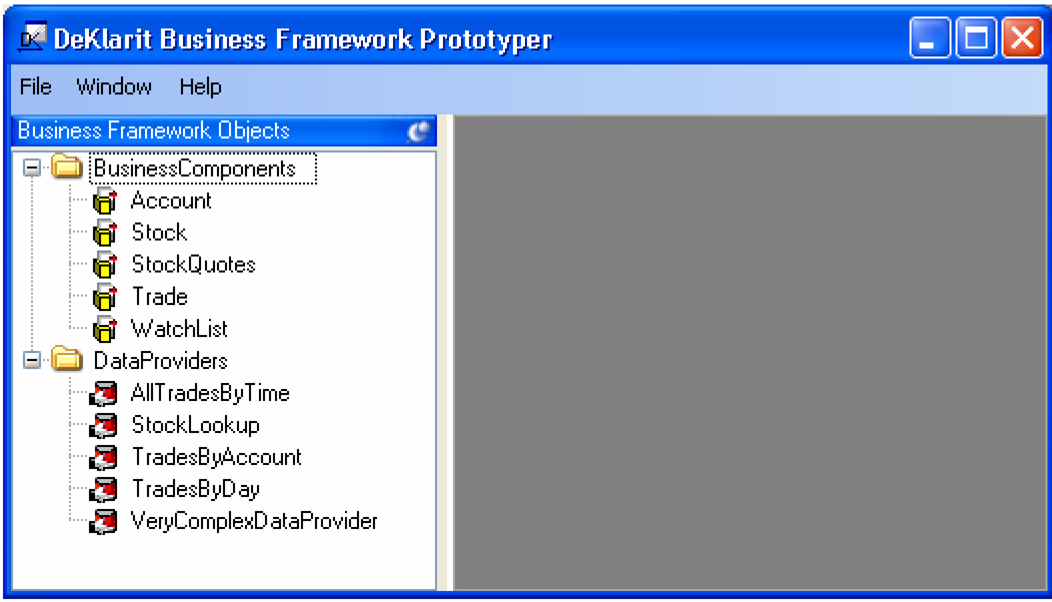


Figure 37

You will see a list of all the Business Components and Data Providers. If you double-click the 'Account' Business Component, the **Work With Account** form will be displayed. The Work With form is the implementation of a very common pattern used in data-based applications. It is composed of two forms, the first one lists the objects you want to work with (Account in this case) and a set of actions you can apply to each instance (Insert, Update and Delete are the default ones). The second form is for working with a specific Business Component instance, and lets you insert a new one, update it or delete it.

Below is the Work With pattern with the Account Business Component:

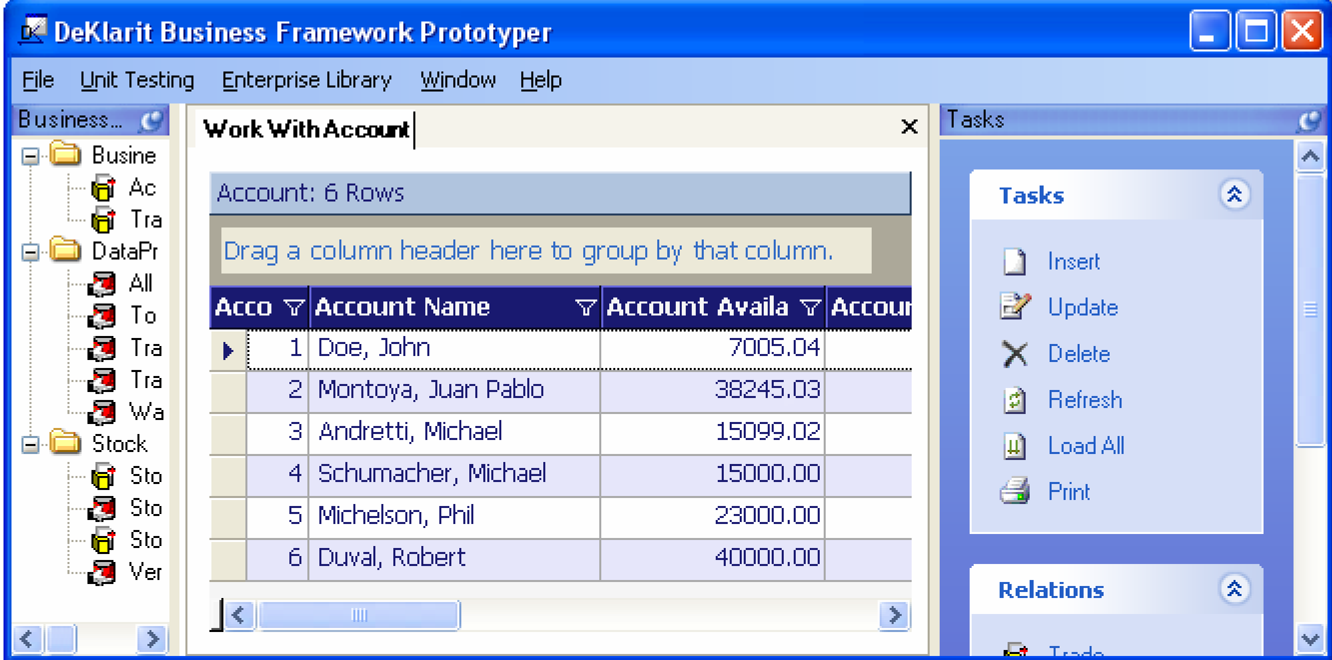


Figure 38

And if you choose the **Update** button for 'Doe, John' ...

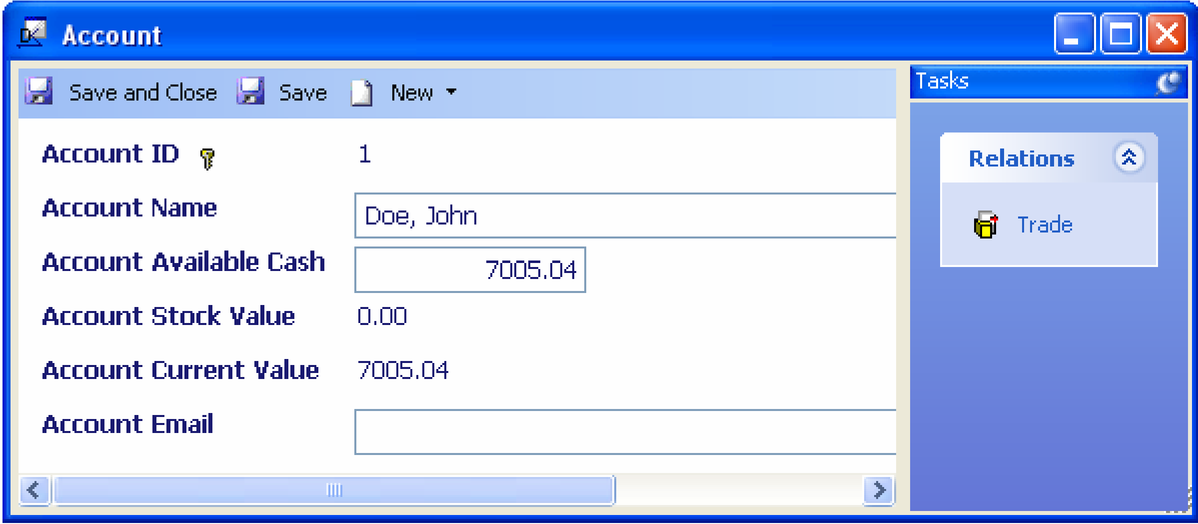


Figure 39

... you can update an instance of the Account Business Component. Remember that all the Business Component Rules are being applied. In this case, the formulas Account Current Value and Account Stock Value are not enabled because they are automatically calculated by the Business Framework.

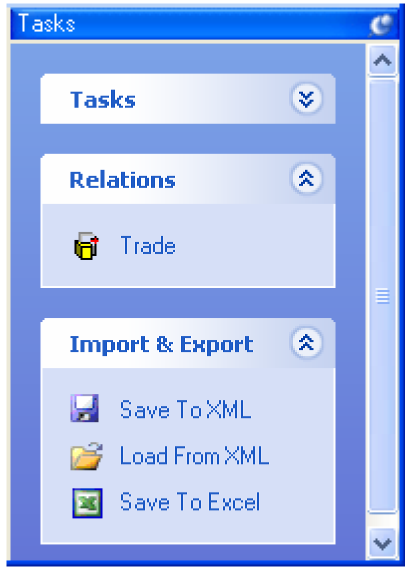


Figure 40

The other option available in the Business Framework Prototyper for Business Components is the 'Relations' option.

As DeKlarit knows all the relationships between Business Components, the Business Framework Prototyper uses that information and builds forms that let you to navigate through all the Business Components subordinates, including the first one. For example, here you can see all the Trades related to the first Account, as shown in Figure 41.

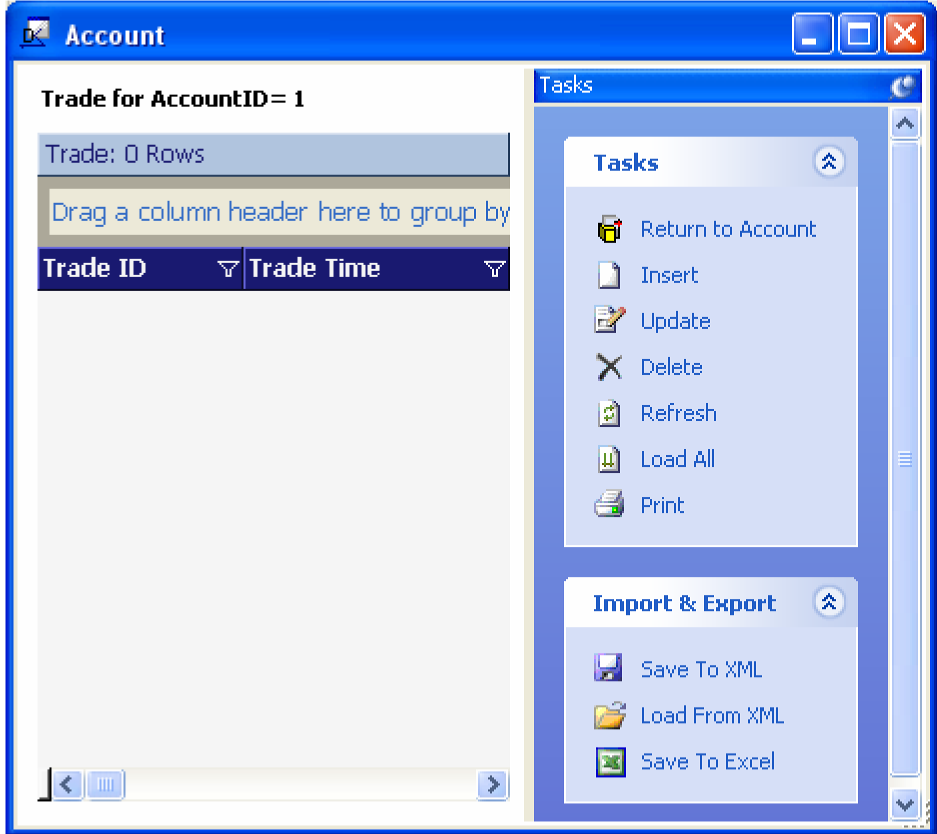


Figure 41

As the Business Framework Prototyper exposes all the Business Framework functionality, it is an invaluable tool for testing the Business Framework at development time, and its use is strongly recommended.

This tool uses the metadata that DeKlarit generates for the .NET Components. If you want to know more about it, including how it was built, and how you can build your own, see the Business Framework Prototyper documentation, and the DeKlarit Metadata section. The source code for this tool is distributed with DeKlarit.

DeKlarit Add-Ins

DeKlarit's set of add-ins take advantage of the generated metadata to create .NET applications that use the Business Framework. To be able to use an add-in, you need to add a reference to it. To add a reference, right-click the **Add-in References** folder in the DeKlarit project and select **Add Add-in Reference**.

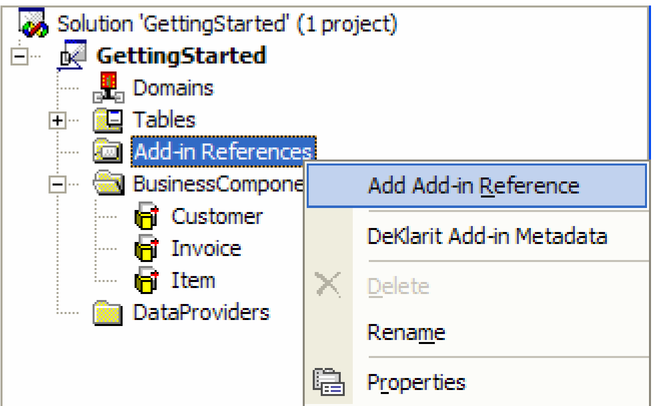


Figure 42

The add-in reference dialog will be displayed for you to select an Add-in:

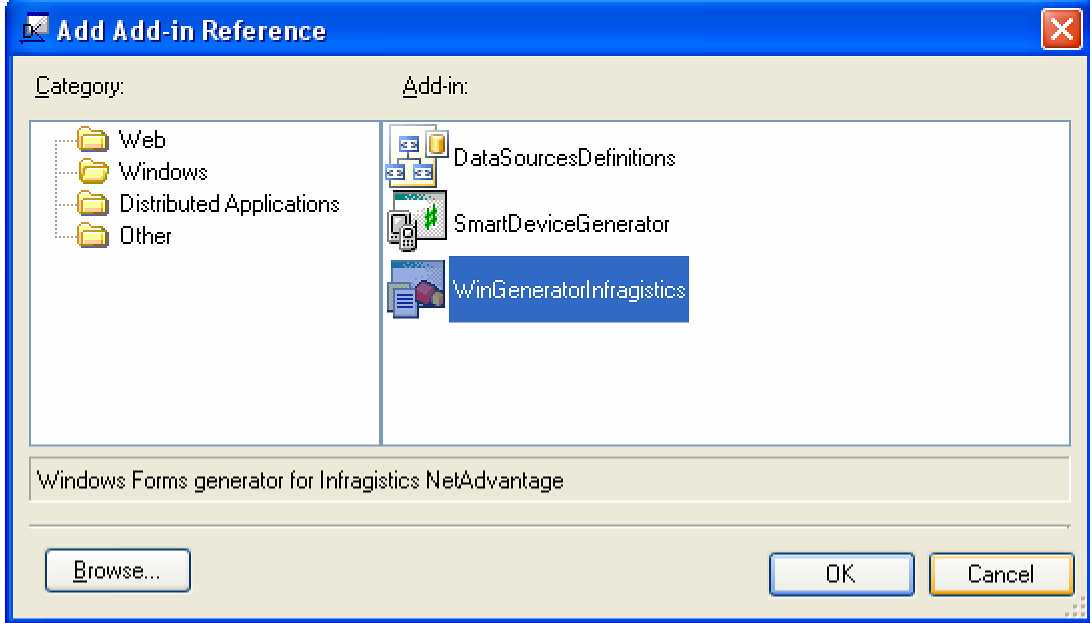


Figure 43

For more information about these tools, see Creating the Presentation Layer in the Help documentation provided with the software.

Incremental Development

Suppose that you create an application and when you show it to your users, they want to make some changes to it. Some of these changes could be easily implemented while others could be more difficult. Those that require modifications to the Database Schema are the most troublesome ones, so developers usually try to avoid them.

DeKlarit implements a set of features that truly makes incremental development methodology a reality. One of these features is the Impact Analysis Report, which allows you to check the differences between the modified Database Schema and the old one.

This feature is complemented by the DeKlarit Reorganization utility, which converts the data from the old Database to the new one.

See how it works in this example. Suppose it is necessary to store the email address of each account. You just have to open the Account Business Component and add the new attribute, as shown in Figure 44.

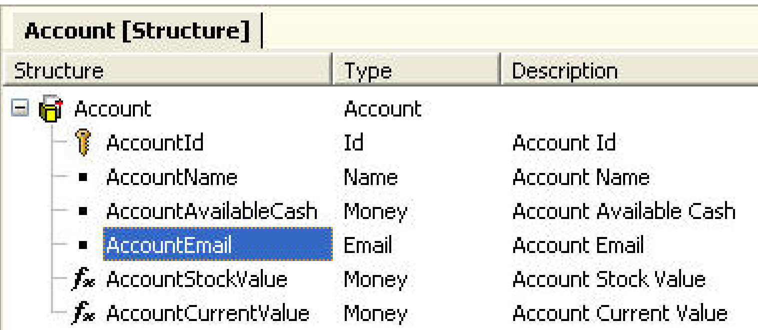


Figure 44

If you run the Impact Analysis Report,

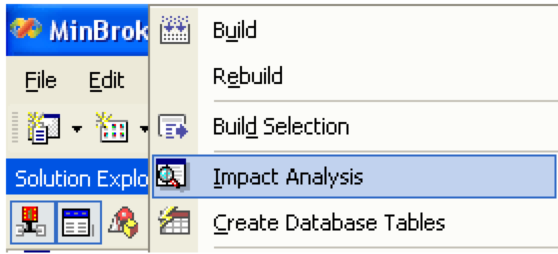


Figure 45

DeKlarit will detect the change in the Account table and tell you that once you run the Reorganization, the AccountEmail attribute will be created with a Null value. If you want another value, change the attribute's Initial Value property.

The Impact Analysis is just a report, and it does not update the Database. To run the Reorganization utility, choose Build and DeKlarit will detect a change in the Database Schema. It will automatically generate and run the Reorganization utility before regenerating the Business Framework.

It is very convenient to incrementally develop the Business Framework. Start with a minimum set of Business Components and Data Providers, and then add more attributes or components as necessary. Remember that an intensive use of prototypes allows you to deliver the Business Framework to your users in the shortest time possible.

In this example the modification was not so difficult to achieve, as only one new attribute was added to a Business Component, but DeKlarit is capable of doing far more sophisticated reorganizations, like moving an attribute from one table to another or defining redundancies. For more details, read the white paper An Introduction to DeKlarit.

Rules

One of DeKlarit's most surprising features is that the order in which the Rules are written is not important. The reason for this approach is to free the developer from HOW so as to focus on WHAT during the Business Framework development.

More on Rules

Formally, DeKlarit's Rule language is a simple declarative programming language derived from Production Rules. It is declarative as opposed to procedural or imperative, meaning that the rules order is not important. For example, in a procedural programming language like Visual Basic, the following code:

```
A = 1
B = A + 3
```

Is different from:

```
B = A + 3
A = 1
```

But the following DeKlarit Rules:

```
A = 1;
B = A + 3;
```

Are exactly the same as:

```
B = A + 3;
A = 1;
```

DeKlarit generates Visual C#, which is a procedural language. So, how can it generate the Rules? It is very simple because it generates them in the 'correct order,' using data dependencies. In the previous example, B depends on the value of A and A depends on a constant (1), so regardless of the order in which you write it, the correct order is:

```
A = 1;
B = A + 3;
```

What happens when there are circular dependencies?

```
A = B;
B = A + 1;
```

In this case, DeKlarit uses the order in which they were defined. In the real world these circular dependencies are very uncommon.

The general form for Rules is:

```
<Action> [IF <Condition>];
```

Which reads to: execute <Action> if <Condition> is true. If <Condition> is absent, always execute <Action>. Any attribute that is defined in the Structure can be used in the <Action> or in the <Condition>.

<Condition> is a logical expression. The reserved words Insert, Update, and Delete can be used. As their name imply, these variables are Boolean, and are true depending on whether you are inserting, updating or deleting a Business Component instance.

Error

This is one of the most common Rules, and it prevents you from entering invalid data in the Database.

```
error( ExceptionName [, <StringExpression>] ) if <Condition> ;
```

They can be very simple:

```
error("Name is required and not filled") if AccountName.IsEmpty();
error( EmptyAccountNameException) if AccountName.IsEmpty();
```

Or more complex, like:

```
error( NoCashException, "Not enough available cash to buy the stock")
if AccountAvailableCash < 0 and TradeBuySell = -1;
```

Every time an Error rule becomes true at execution time, the exception is thrown and its message is <StringExpression>, if provided. See how the following code handles the exception thrown by the Business Framework:

```
{
        accountDataAdapter1.Update(accountDataSet1);
}
catch (NoCashException e)
{
        Label1.Text = e.Message;
}
```

Assign

Assign is another important Rule. It is used to assign an attribute value programmatically.

```
<Attribute> = <Expression> [ if <Condition> ] ;
```

For example:

```
TradeCommission = 9.99 if insert;
```

It is valid to have many rules with the same assignment:

```
TradeCommission = 9.99 if insert and TradeAmount < 10000;
TradeCommission = 7.99 if insert and TradeAmount < 100000;
TradeCommission = 5.99 if insert and TradeAmount >= 100000;
```

In this case, the first rule whose condition evaluates to true will be the only one executed. For example if the TradeAmount is 20,000 the TradeCommission will be 7.99.

Note: it is a bad design choice to define commission amounts in the Business Component Rules, as this is just an example of multiple assignment rules.

It is important to know the difference between the Assign Rule and a Formula because the syntax is exactly the same, but their meaning is different. Just like the other rules, the Assign rule is local to the Business Component which contains it. So the Assign rule is only executed when inserting, deleting or updating a Business Component instance, and the attribute assigned by the rule is stored in the Database. On the other hand, a Formula is global and can be used everywhere. Also, it is not stored in the Database by default.

Which is better to use, an Assign rule or a Formula? Because this question does not have a right answer, DeKlarit uses both.

From the performance point of view the Assign rule is better, because Formulas are recalculated as needed. But it is not always possible to substitute a Formula by an Assign rule. Consider AccountStockValue:

```
AccountStockValue = sum(TradeCurrentValue);
TradeCurrentValue = TradeQty * StockLastPrice * TradeBuySell * -1;
```

The value of AccountStockValue changes every time StockLastPrice changes, so it is better to calculate the AccountStockValue as needed instead of updating all the Accounts every time a StockQuote is added.

From the Data Modeling point of view, an Assign rule is a kind of redundancy.

Default

This Rule lets you define default values in the Business Component itself.

```
Default( <Attribute>, <Expression>);
```

The default value will apply only if the attribute IsNull() methods return true.

Example:

Default(TradeCommission, 9.99);

means that every time a Trade is inserted, and the TradeCommission is null, it is stored with a value of 9.99.

The Default rule is equivalent to:

<Attribute> = <Expression> if insert and <Attribute>.IsNull();

Static Method Call

Another <Action> is calling a Class Static Method.

```
<StaticMethodCall> if <Condition> ;
```

This rule is used for some special tasks inside the Business Framework.

Example:

MyNamespace.MyClass.MyStaticMethod(...) if insert;

Add and Subtract

The Add and Subtract rules are shortcuts to help you write less code. They are very useful to maintain balances like AccountAvailableCash in this example.

```
add( <Attribute1>, <Attribute2> );
subtract( <Attribute1>, <Attribute2> );
```

The Add rule is equivalent to:

<Attribute2> = <Attribute2> + <Attribute1> if insert;

<Attribute2> = <Attribute2> - <Attribute1> if delete;

<Attribute2> = <Attribute2> + <Attribute1> - <Attribute1>.OldValue() if update;

In other words, this rule adds the value of the first attribute to the second on Insert, subtracts the value on Delete, and adds the difference on Update.

The subtract rule is quite similar except that the signs are opposite, as it subtracts instead of adding.

Serial

The Serial rule is usually used to automatically number some values of a Primary Key depending on other values.

```
serial( <AttributeToBeNumbered>, <AttributeWhichHasTheLastValue>, <Increment> );
```

Consider an Invoice with an Invoice header and many lines. How can each Invoice line be identified? Each line has an Item; if in the real world an Invoice cannot have two lines for the same Item, the Primary Key for the lines is ItemID, as shown in Figure 46.

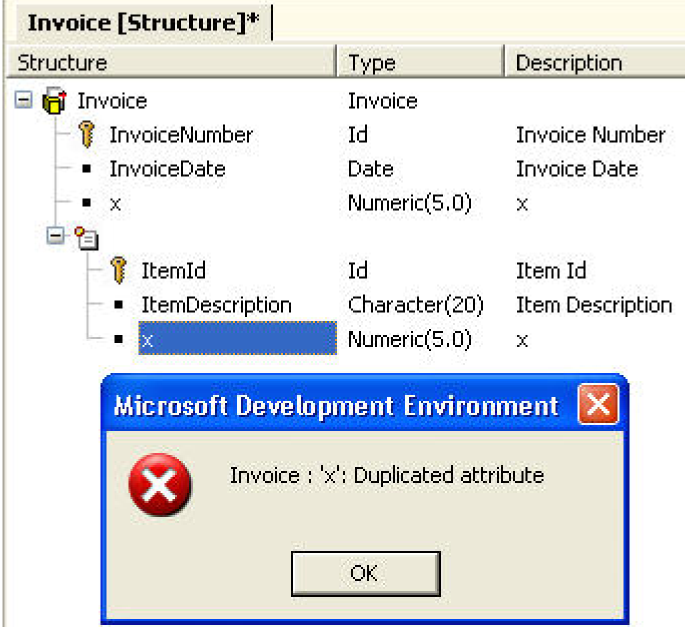


Figure 46

But, what if this is not true? You have to uniquely number the lines and make this number part of the Primary Key.

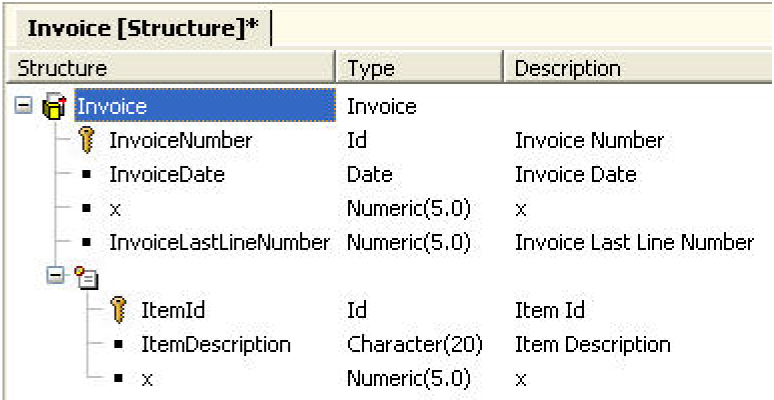


Figure 47

The Serial Rule provides a convenient way to autonumber the lines:

```
serial(InvoiceLineNumber, InvoiceLastLineNumber, 1);
```

With this rule, each Invoice line will be numbered and the last value stored in InvoiceLastLineNumber.

The Serial rule is equivalent to:

```
InvoiceLastLineNumber = InvoiceLastLineNumber + 1 if insert;

InvoiceLine = InvoiceLastLineNumber if insert;
```

Note that in this case the Autonumber property cannot be used because it autonumbers all the rows of a table. Here, the idea is to start a new numbering sequence in each Invoice.

Extended Table

The Extended Table concept is a generalization of a Relational Table. Take a look at the Database Schema for an Online Broker shown in Figure 48.

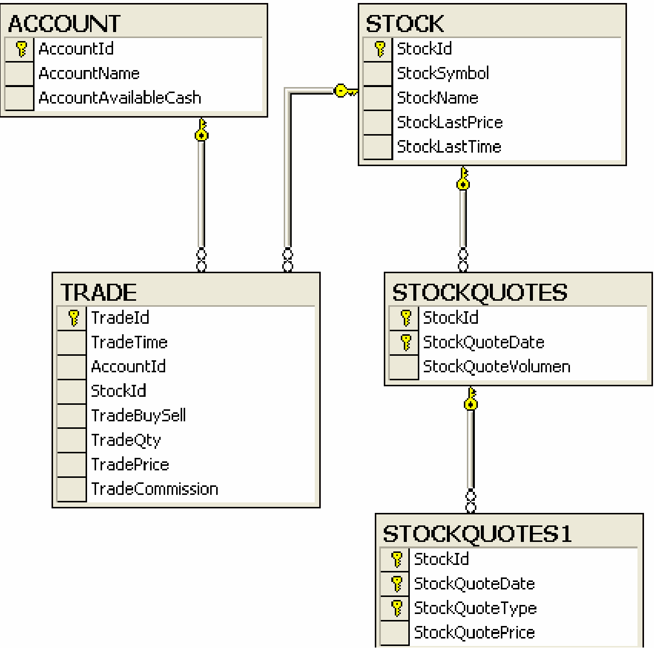


Figure 48

The Trade table is composed by its Primary Key TradeID, and its attributes are TradeID, TradeTime, AccountID, StockID, TradeBuySell, TradeQty, TradePrice, and TradeCommission. From the Relational Theory point of view this is the same as saying:

"There are no two Trades with the same TradeID value, and given a TradeID value, there is only one value of TradeTime, AccountID, StockID, TradeBuySell, TradeQty, TradePrice, and TradeCommission."

This relationship among attributes is known as Functional Dependencies, and it can be said that Data Normalization is the process of defining a minimum, non-redundant set of functional dependencies.

But, is the relationship between TradeID and AccountName a functional dependency? The answer is yes, because:

for each TradeID there is only one AccountID

for each AccountID there is only one AccountName

So

for each TradeID there is only one AccountName.

In a normalized Database Schema (as DeKlarit models are) a Table contains the set of the **minimum** functional dependencies related to its Primary Key.

On the other hand, the Extended Table of a Table is the set of **all** the functional dependencies related to its Primary Key. More precisely, it is the transitive closure of the table's functional dependencies (this table is generally known as the Base Table of the Extended Table).

This rather technical definition may not clearly express how easy it is to 'see' a certain table's Extended Table. In fact, it is the set of table attributes, and all the attributes present in the N-to-1 related tables.

For example, what is the Trade's Extended Table?

TradeID, TradeTime, AccountID, StockID, TradeBuySell, TradeQty, TradePrice, TradeCommission (from Trade table)

Plus:

AccountName, AccountAvailableCash (from Account)

And:

StockSymbol, StockName, StockLastPrice, StockLastTime

Visually, it is just a matter of following the arrows from the N side () to the 1 () side, but not the other way around.

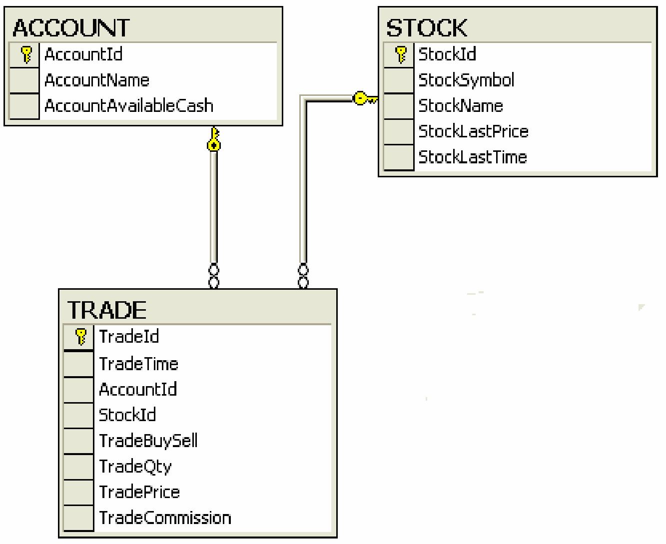


Figure 49

StockQuotes' Extended Table is:

> StockID, StockQuoteDate, StockQuoteVolume (from StockQuotes itself), StockSymbol, StockName, StockLastPrice, StockLastTime (from the N-to-1 table Stock).

Sometimes the Extended Table is exactly the same as its base table. For example the Account's Extended Table is AccountID, AccountName, AccountAvailableCash, the same as its base table (Account). This happens when the base table does not have any N-to-1 related tables.

After years of research and real-world designs, the Extended Table concept was found to be very practical for Business Framework design. Therefore, it is used in many areas of DeKlarit.

Extended Table and Data Providers

The Extended Table Concept is frequently used in Data Providers. Taking the example of an Online Broker, suppose that a single-level Data Provider has been defined as follows:

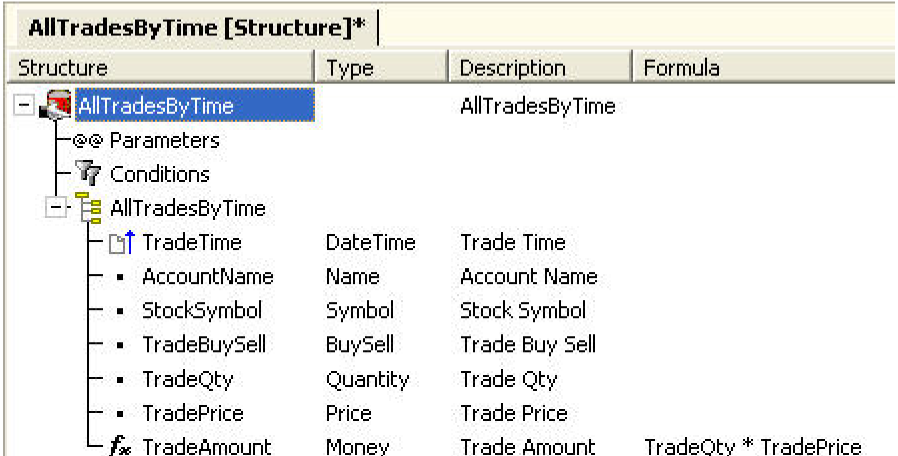


Figure 50

There is no single table that has all these attributes. However, DeKlarit is capable of finding the relationship among them because all these attributes belong to Trade's Extended Table. The Navigation Report is shown in Figure 51.

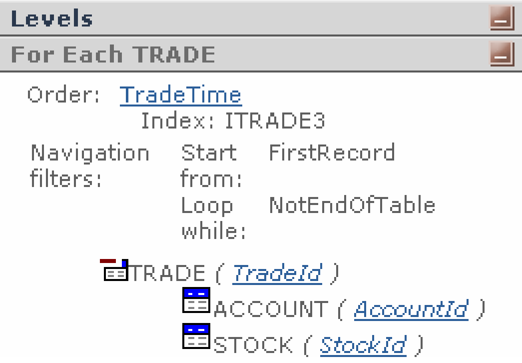


Figure 51

This means that DeKlarit will automatically generate a join between the Trade, Account and Stock tables. Note that the join between Trade and Account uses AccountID, but it is not necessary to define AccountID in the Structure.

The attributes of a Data Provider level do not have to belong to a single Table; however, they must belong to an existing Extended Table. You do not need to explicitly define which one because DeKlarit will find it.

One of the advantages of this approach is that Data Providers are more independent from the underlying Data Model. For example, if a modification in the Data Model does not alter the Extended Table, the Data Provider remains the same.

The Extended Table Concept is also used in multi-level Data Providers to find their relationships. For example, if the attributes of a parent level belong to the Extended Table of the nested one, a constraint relating both levels is automatically defined. For example:

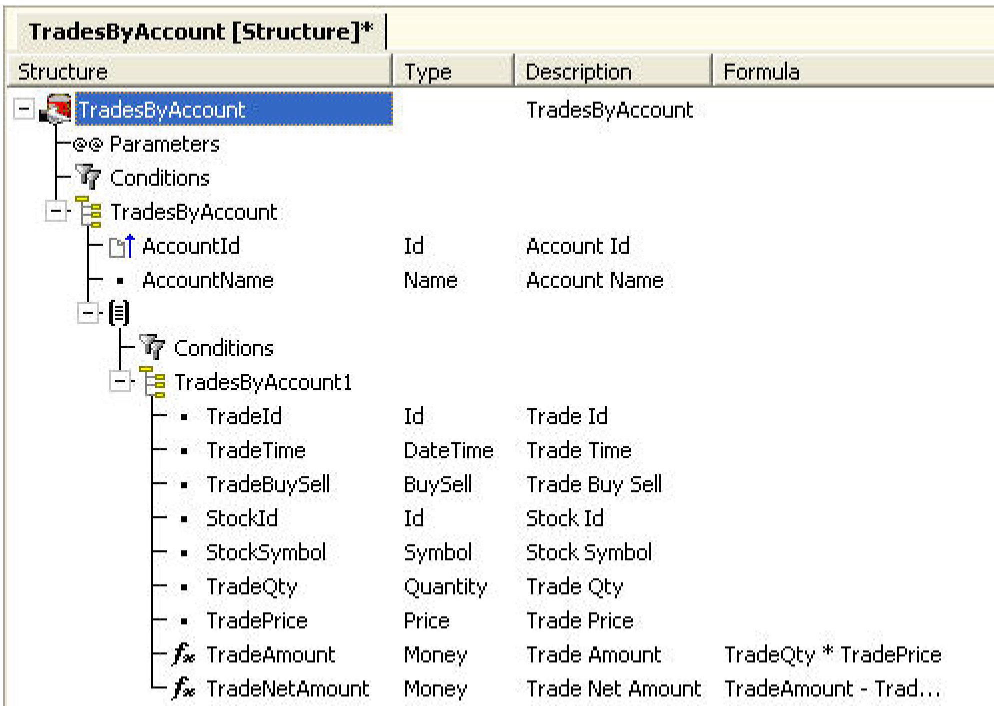


Figure 52

For each Account, you want to list all its Trades, not all the trades in the Trade table. This is correctly inferred by DeKlarit, as you can see in the Navigation Report shown in Figure 53.

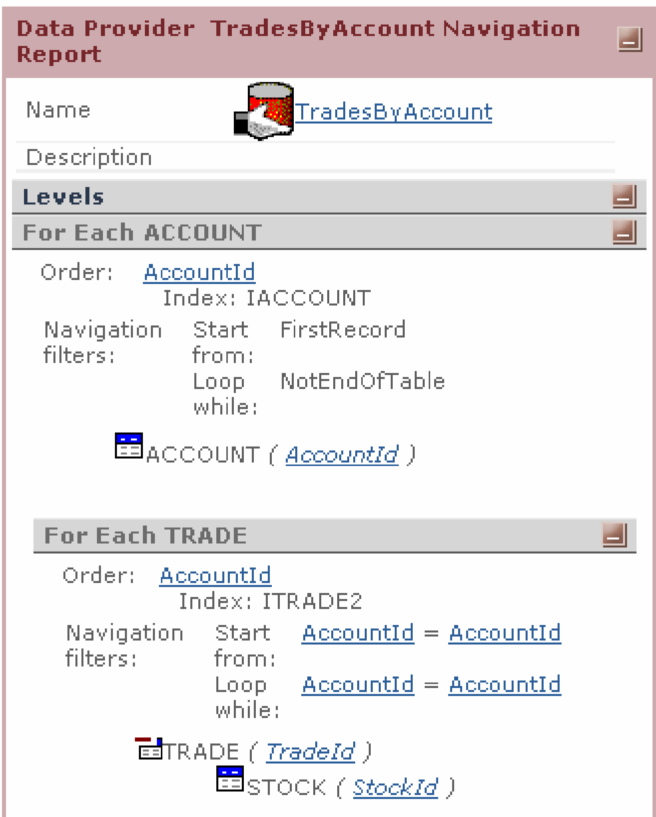


Figure 53

There is a constraint in the second level –AccountID- inferred by DeKlarit.

When one level cannot be related to its parent, a 'Cartesian product' is used. In other words, for each parent level instance, all the instances of the nested level are retrieved without any constraints. Usually, this is not what developers want so it is advisable to look at the Navigation Reports to see the constraints for each nested level.

A case where the parent level's Extended Table belongs to the nested level's Extended Table has already been discussed. But, what happens if they are exactly the same? Consider the defined TradesByDay Data Provider shown in Figure 54.

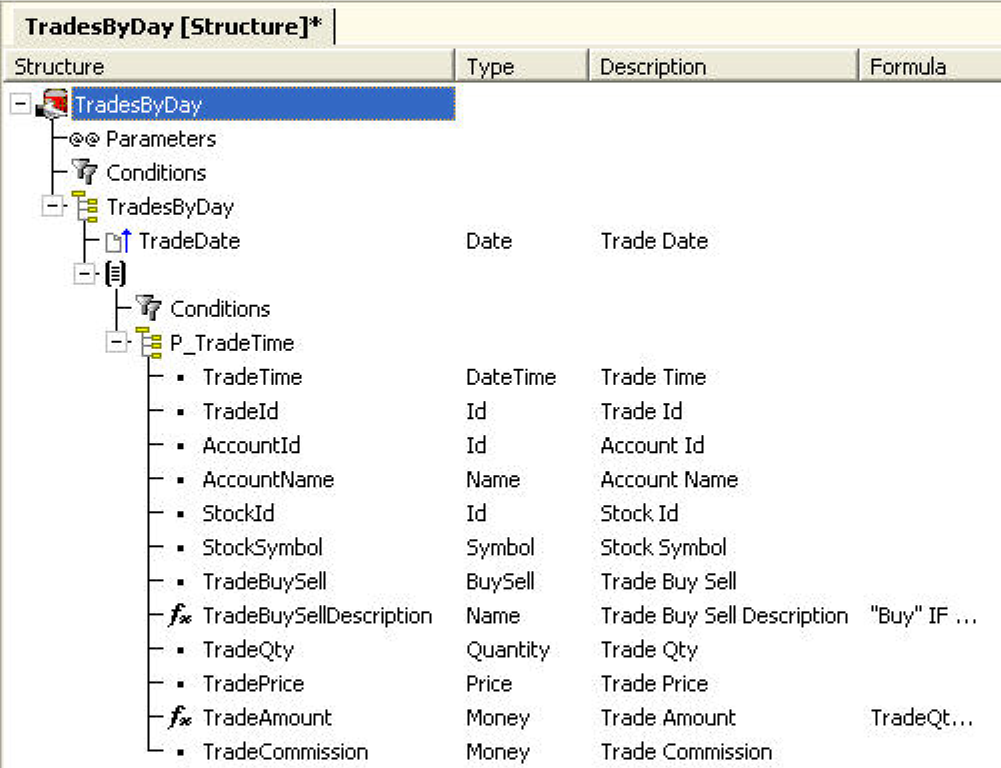


Figure 54

And look at the Navigation Report:

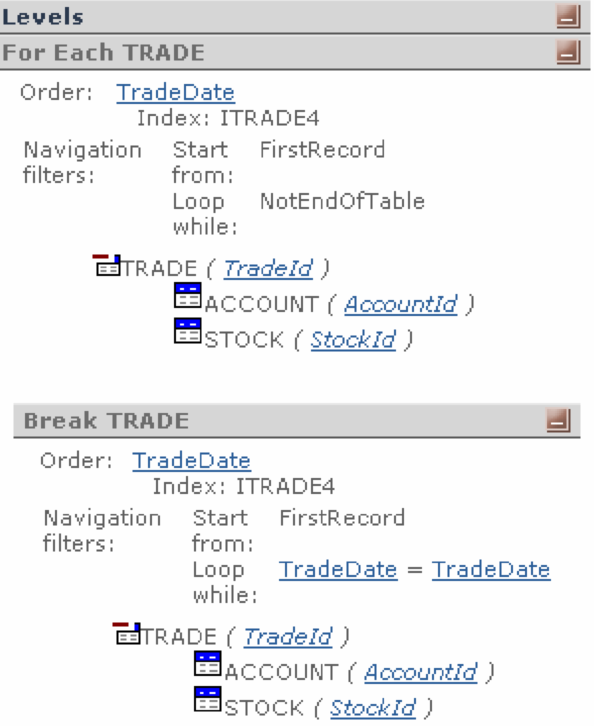


Figure 55

The Extended Table of both levels is Trade. Instead of adding a For Each to both of them, DeKlarit adds a Break in the second one, meaning that the second level is a Control Break of the first one.

DeKlarit inferred the TradeDate constraint in the second level (TradeDate) from the first level Order. To define Breaks correctly, it is important to define the order of the parent level.

More on Formulas

An attribute is a Formula when its value can be calculated from other attribute values. Or, more technically, it can be defined by intention instead of by extension. Unless the Formula is defined as Redundant, its value is calculated every time it is needed, and it is not stored in the Database. As you can see in the Broker Business Framework, Formulas are very common in a Business Framework.

An important property of Formulas is that they are global. A Formula is defined in a Business Component, but once defined, it can be used everywhere as any other attribute.

Horizontal Formulas

There are many types of Formulas. The most common one is known as Horizontal Formula. Its general expression is:

```
Formula = f( Att1, ..., Attn);
```

Where f(...) is an expression whose type must be the same as the Formula.

It can have just one expression like:

AccountCurrentValue = AccountAvailableCash + AccountStockValue;

Or it can have multiple conditional expressions:

TradeNetAmount = TradeAmount – TradeCommission if TradeBuySell = 1;

(TradeAmount + TradeCommission) * -1 Otherwise;

In this case the formula will have the value of the first expression whose condition is true. It is a good design technique to always define the Otherwise condition in this kind of formulas.

DeKlarit's design objective is to let you define the formula in the most intuitive way possible. Just define WHAT the formula is without any reference to HOW to calculate it, as it is DeKlarit's responsibility to find the way to do it. Once again, the Extended Table Concept is helpful.

For example, look at a formula used in the Broker Business Framework:

TradeCurrentValue = TradeQty * StockLastPrice * TradeBuySell * -1;

The WHAT of this formula is very easy to understand, but the HOW is more complicated. There is no single Table that has all the attributes (TradeCurrentValue, TradeQty, StockLastPrice, and TradeBuySell), but there is an Extended Table (Trade) that has all the attributes, as shown in Figure 56.

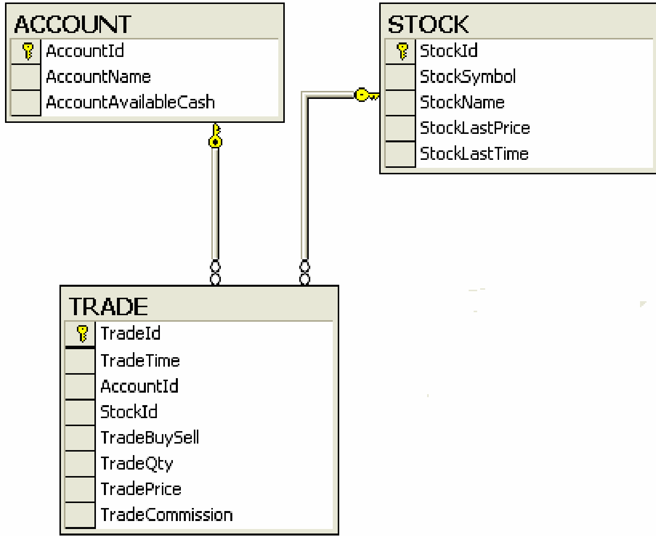


Figure 56

NOTE: Given a Horizontal Formula (Formula = f(Att1… Attn);), Att1, … Attn should belong to the Extended Table of the Table where Formula is defined.

A Horizontal Formula can depend on other Formulas (of any type).

The expression can have arithmetic operators like +, -, *, /, and even static methods defined outside the Business Framework like System.DateTime.Now. It is also possible to use static methods in conditions.

Vertical Formulas

As opposed to Horizontal Formulas, a Vertical Formula aggregates the values of another attribute; therefore, this attribute is not in the Formula's Extended Table.

The most common type of Vertical Formula is SUM.

For example:

AccountStockValue = SUM(TradeCurrentValue);.

In this case, AccountStockValue is in the Account table –logically, not physically because it is a formula- and TradeCurrentValue is in Trade. Account has a 1-to-N relationship with Trade, so for each AccountStockValue DeKlarit will sum only the Trades of that specific Account.

Note that since it is not necessary to define the join between both tables, the Business Framework is more independent from the Data Model and easier to define.

The navigation to calculate the Vertical Formula is detailed in the Navigation Report:

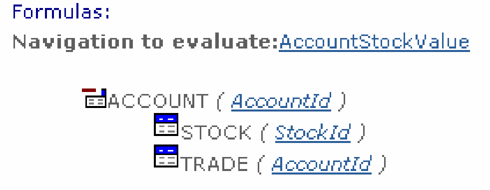


Figure 57

If DeKlarit cannot find a relationship between the attributes a 'Cartesian product' is applied.

Sometimes it is necessary to SUM not all the possible values, but just a few that comply with a certain condition. In this case the Where clause is used:

AccountTodayTradedAmount = SUM(TradeAmount) Where TradeDate = System.DateTime.Now();

The syntax for Vertical Formulas is:

```
<VerticalFormula> ::= <VerticalFunction> [Where <Condition>];
<VerticalFunction> ::= SUM( <SummedAttribute>)
<VerticalFunction> ::= COUNT( <CountedAttribute>)
<VerticalFunction> ::= MAX( <AttributeToMaximize> [, <ReturnedAttribute> ])
<VerticalFunction> ::= MIN( <AttributeToMinimize> [, <ReturnedAttribute> ])
<VerticalFunction> ::= FIND( <AttributeToFind> )
```

COUNT counts the number of times the attribute occurs. For example:

AccountTrades = COUNT(TradePrice);

NOTE: the value of the counted attribute (TradePrice in this case) is not used; it just counts the rows where TradePrice is present.

MAX and MIN are quite similar but they have an additional, optional parameter (<ReturnedAttribute>). Suppose you need to know which Account traded a Stock at the highest price:

StockHighestTradedPriceAccount = MAX(TradePrice);

The previous formula is not what you need because MAX will return the TradePrice, but not who traded it. The correct expression is:

StockHighestTradedPriceAccount = MAX(TradePrice, AccountID);

FIND just returns a value of <AttributeToFind>. Do not assume the rows will be inspected in a certain order. Example:

AccountIsActiveToday = 1 if not AccountTradedToday.IsEmpty();
= 0 Otherwise;

AccountTradedToday = FIND(TradeQty) Where TradeDate = System.DateTime.Now();

If no matching value is found when using MAX, MIN or FIND, the formula will have an empty value, which can be checked with the IsEmpty() method.

Built-in Methods

DeKlarit has a set of built-in methods to be used in Rules and Formulas, and most of them are associated to the attribute's type. An intellitip feature is provided in the Rules Editor, so every time you enter . after an attribute, the list of possible built-in methods will appear.

A brief list of these methods is:

All data types

- IsNull()
- SetNull()
- IsEmpty()True if the attribute has the 'empty' value: 0 for Numeric, "" for Character, etc.
- SetEmpty()
- OldValue()

Date/DateTime

- Day()
- Month()

- Year()
- Hour()
- Minute()
- Second()
- DayOfWeek()
- Age([<Date>|<DateTime>])
- AddDays(<Numeric>
- AddMonths(<Numeric>)
- AddYears(<Numeric>)
- AddHours(<Numeric>)
- AddMinutes(<Numeric>)
- AddSeconds(<Numeric>)
- EndOfMonth()
- ToString([<format>])
- Set(<Y>, <M>, <D> [, H [, M [, [S]]])

Character

- SubString(<numeric>, <numeric>)
- Length()
- Trim()
- TrimStart()
- TrimEnd()
- PadLeft()
- PadRight()
- ToUpper()
- ToLower()
- IndexOf(<char> ...)
- LastIndexOf (<char> ...)
- Replace(...)
- ToNumeric()

Data Types

DeKlarit's data types are DBMS oriented instead of memory oriented. That is why there are no data types such as Integer, String, etc. Supported data types:

- Characters

1. Char(length). Fixed length character string.
2. VarChar(length). Variable length character string. Length is the maximum length.
3. Text.

- Numeric(length [, decimals])
- DateTime
- Date

- Blob
- Boolean
- GUID

For more on this, read the Data Types and Methods section of this documentation.

Subtypes

DeKlarit uses the Universal Relation Assumption (URA), which says that one thing has the same name everywhere. For example, the Account's number is AccountID in the Account Business Component **and** in Trade Business Component. However, sometimes it is not possible to follow the URA due to two kinds of situations, such as the existence of more than one relationship between Business Components, and the need to avoid unnecessary Referential Integrity Constraints.

Multiple Relationships between Business Components

Consider a Business Framework for an Airline that needs to record its flights. Once again, this is a very simplified model. The Airport Business Component is easy:

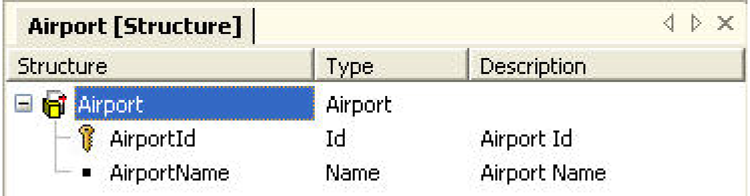


Figure 58

But the Flight Business Component has some complications. A Flight occurs between two Airports, but of course, it is impossible to put AirportID twice in the Structure. This has been simplified because in the real world a Flight has many Flight Legs, and each Leg occurs between two Airports.

Figure 59

One of these Airports is the Departure Airport and the other is the Arrival Airport. The use of subtypes is needed to solve the problem: every Departure Airport is an Airport (but not necessarily the opposite). DeKlarit defines subtypes on an attribute and group basis. For example, the first group is Departure:

Figure 60

The second group is Arrival, as shown in Figure 61:

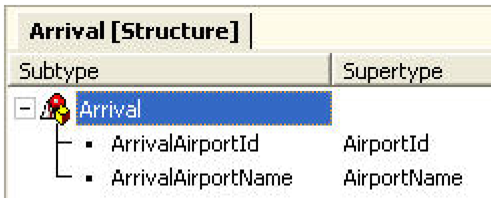


Figure 61

The Flight Business Component Structure is as follows:

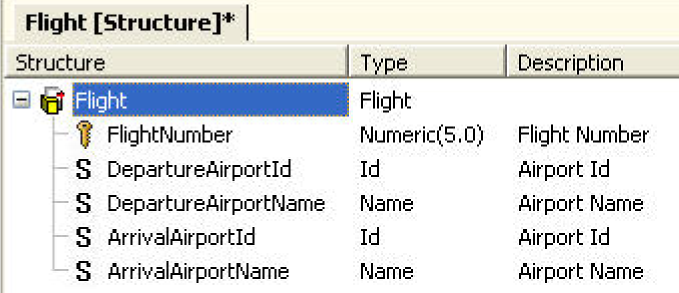


Figure 62

Note how DeKlarit adds an **S** to indicate the attribute is a subtype.

The Database Schema is:

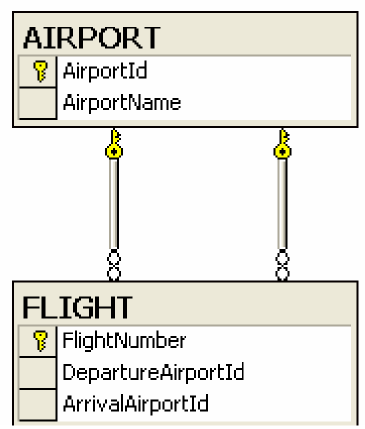


Figure 63

Self-References

Another common use of subtypes is when you need to reference one Business Component to itself. For example, suppose that you want to define an 'Employee' Business Component in which there is a Manager for each Employee, but this Manager should also be an Employee.

First, start by defining the Employee Business Component as shown in Figure 64.

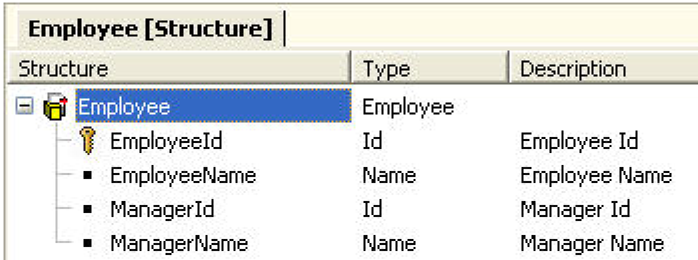


Figure 64

As you want Managers to be Employees too, define a Subtype that will tell DeKlarit that Managers are a subtype of Employees:

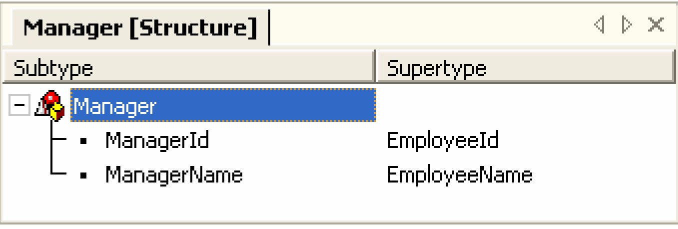


Figure 65

Now the Employee Business Component will show which attributes are subtypes and which ones are inferred:

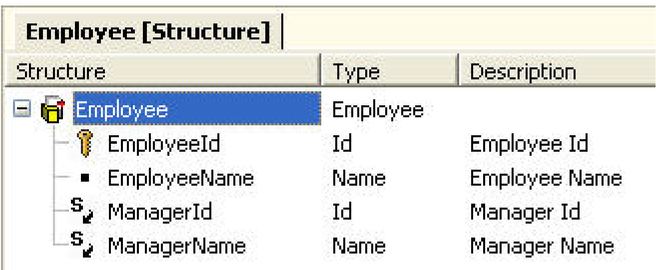


Figure 66

To complete the definition of the Business Component you need to take into account a border case. When inserting the first Employee, there will not be any Managers that you can assign to it, so you need to allow null values for the ManagerID field by setting the **AllowNull** property to **Yes**:

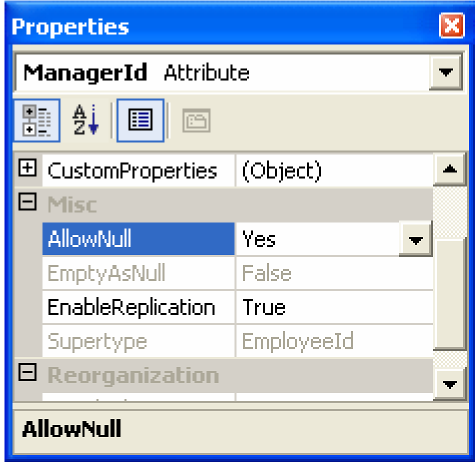


Figure 67

The table defined by DeKlarit will look as shown in Figure 68.

Figure 68

Inferred Attributes

If you look at the previous Database Schema, DepartureAirportID is stored in the Flight table, but DepartureAirportName is not present in any table. DepartureAirportName is an inferred attribute because DeKlarit knows how to infer its value from DepartureAirportID; it only needs to go to the Airport table with DepartureAirportID and pick AirportName. The same applies to ArrivalAirportName.

Once an attribute is defined as a subtype, DeKlarit takes control of the attribute type. For example, if the AirportID type is changed, DeKlarit changes the type of all its subtypes (DepartureAirportID and ArrivalAirportID). In other words, you do not need to define a subtype attribute type, as it is always the same as its supertype's.

Someone can argue that it would be enough with just one group of subtypes:

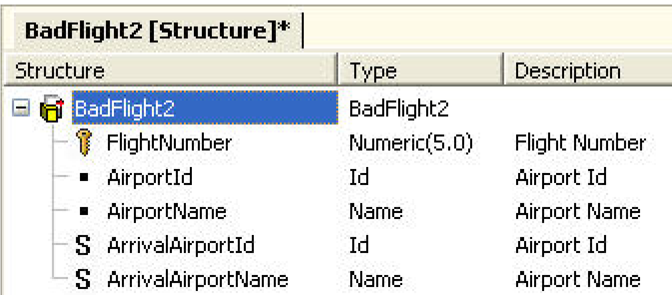


Figure 69

However, mixing the subtype with its supertype is a bad design choice, because the relationship between AirportID and ArrivalAirportID is unclear.

Avoiding Unnecessary Referential Integrity Constraints

Going back to the Online Broker example, suppose it is necessary to implement a Watch List feature.

Each Account can have a Watch List containing the Stocks the Account owners want to have in their 'radars'. It is important to note that it is not necessary to actually own the Stock, it is only a list of stocks to watch.

An inexperienced developer would define a Business Component with this structure:

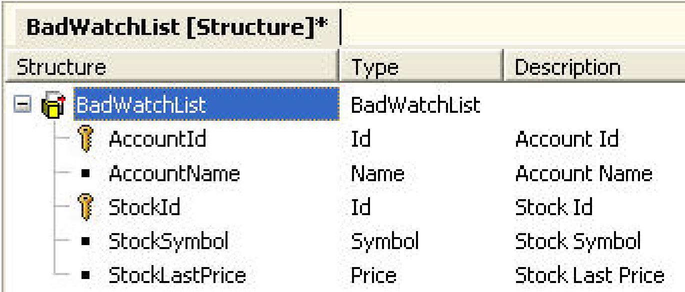


Figure 70

This Business Component's problem is that there is already a relationship between Account and Stock: Trades.

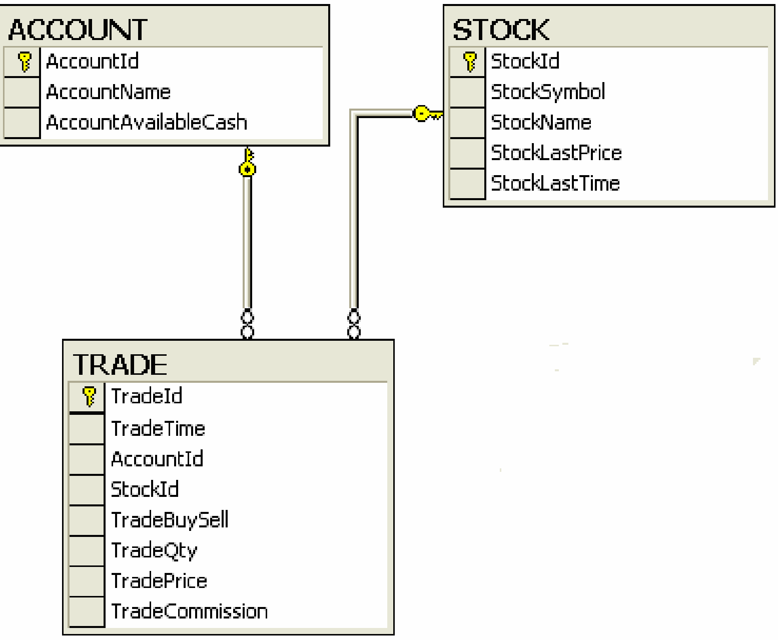
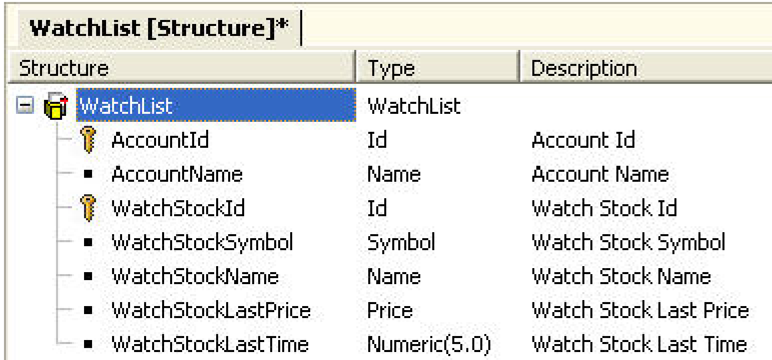


Figure 71

DeKlarit infers a Referential Integrity Constraint between BadWatchList and Trades. In other words, to successfully insert a Trade, the Stock must be in the Account's watch list. To avoid this, you should define that the Stock in the Watch List should not be related to the Stock in Trade. The correct WatchList Structure is as follows:



WatchList [Structure]*

Structure	Type	Description
WatchList	WatchList	
AccountId	Id	Account Id
AccountName	Name	Account Name
WatchStockId	Id	Watch Stock Id
WatchStockSymbol	Symbol	Watch Stock Symbol
WatchStockName	Name	Watch Stock Name
WatchStockLastPrice	Price	Watch Stock Last Price
WatchStockLastTime	Numeric(5.0)	Watch Stock Last Time

Figure 72

StockID is replaced by WatchStockID, StockSymbol by WatchStockSymbol, and so on. WatchStockID is somehow related to StockID, and more precisely, WatchStockID is a **subtype** of StockID. Every WatchStockID is a StockID, but not every StockID is a WatchStockID. The corresponding subtype group is shown in Figure 73.

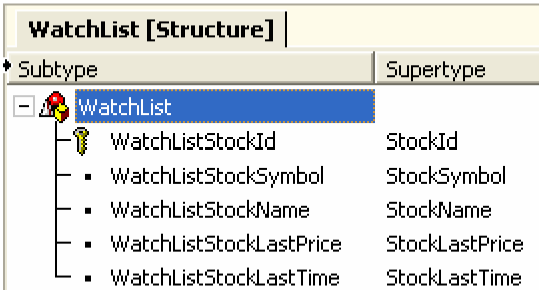


Figure 73

After defining the WatchList subtypes, the WatchList Business Component Structure is:

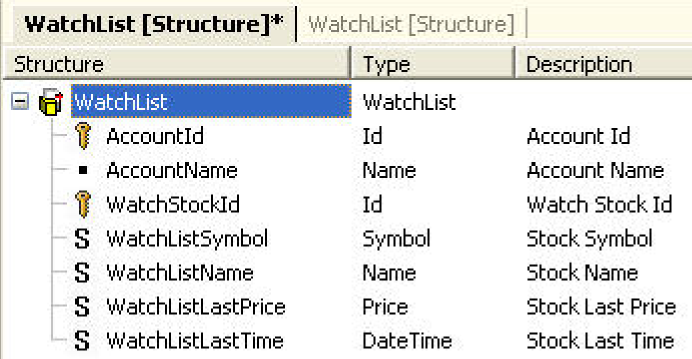


Figure 74

The corresponding Database Schema is:

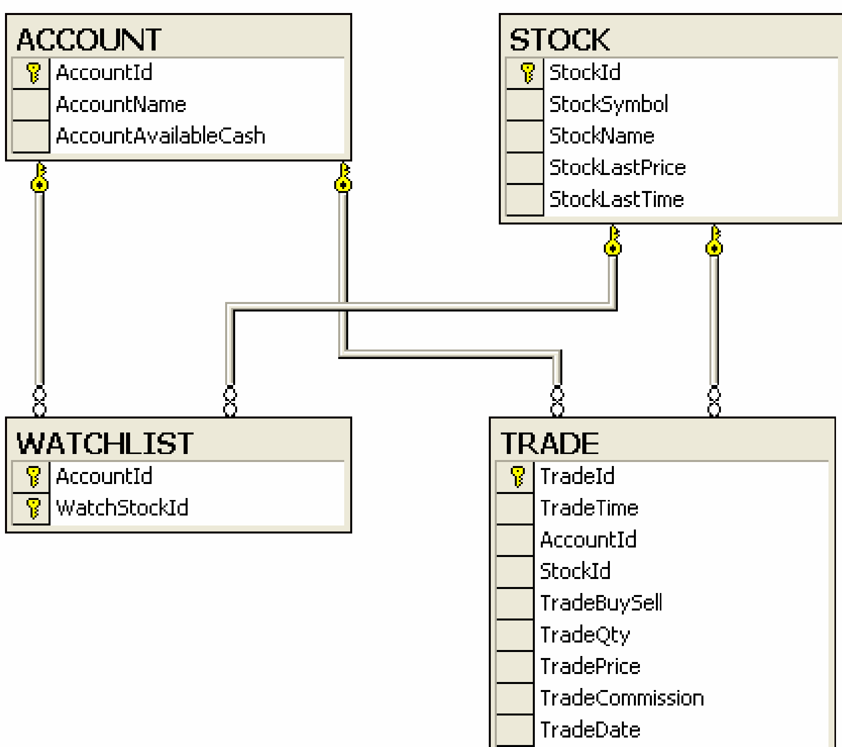


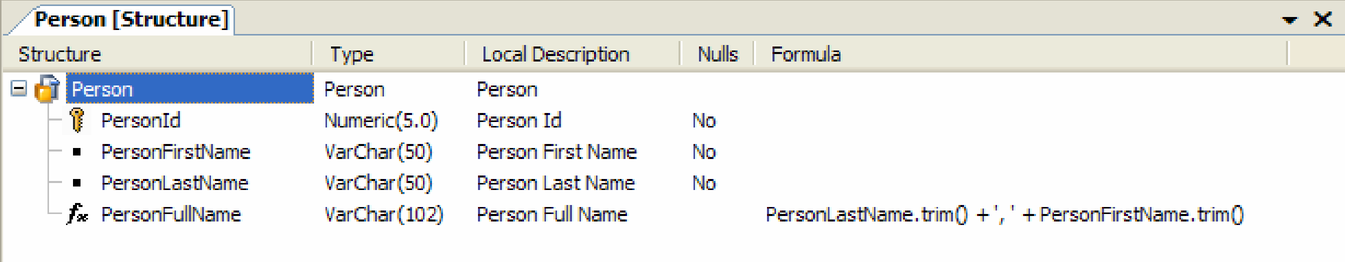
Figure 75

Note that there aren't any Referential Integrity Constraints between the Trade and WatchList tables.

Note: the Referential Integrity Constraints are listed in the Navigation Report.

Inheritance with Subtypes

Subtypes can be seen as an implementation of Inheritance in a DeKlarit model. Look at the classical example of Person, Student, and Employee data model. This is the Person Business Component:

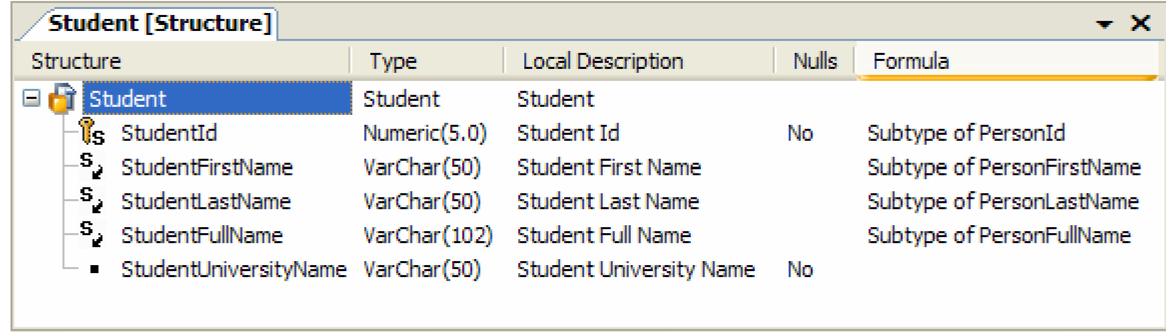


Person [Structure]

Structure	Type	Local Description	Nulls	Formula
Person	Person	Person		
PersonId	Numeric(5.0)	Person Id	No	
PersonFirstName	VarChar(50)	Person First Name	No	
PersonLastName	VarChar(50)	Person Last Name	No	
PersonFullName	VarChar(102)	Person Full Name		PersonLastName.trim() + ', ' + PersonFirstName.trim()

Figure 76

And this is the Student Business Component:

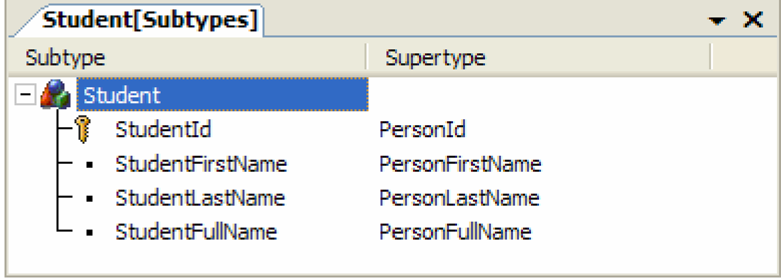


Student [Structure]

Structure	Type	Local Description	Nulls	Formula
Student	Student	Student		
StudentId	Numeric(5.0)	Student Id	No	Subtype of PersonId
StudentFirstName	VarChar(50)	Student First Name		Subtype of PersonFirstName
StudentLastName	VarChar(50)	Student Last Name		Subtype of PersonLastName
StudentFullName	VarChar(102)	Student Full Name		Subtype of PersonFullName
StudentUniversityName	VarChar(50)	Student University Name	No	

Figure 77

Note that StudentName is a subtype because the Student subtype group has been defined:

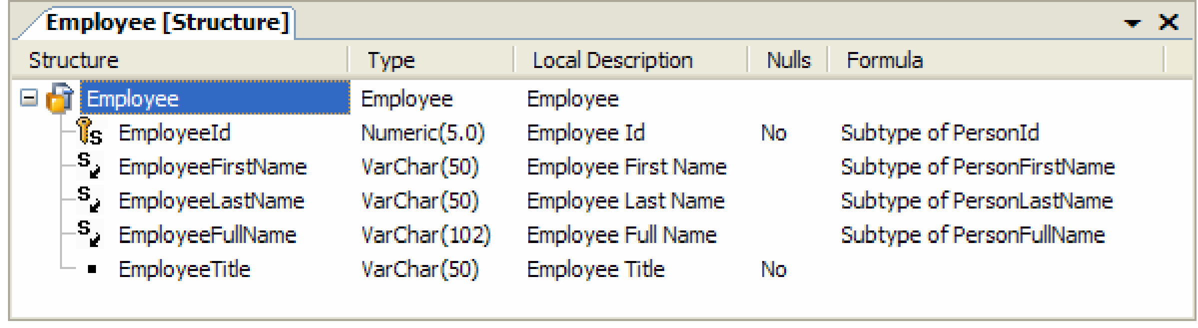


Student[Subtypes]

Subtype	Supertype
Student	
StudentId	PersonId
StudentFirstName	PersonFirstName
StudentLastName	PersonLastName
StudentFullName	PersonFullName

Figure 78

Finally, this is the Employee Business Component:



Employee [Structure]

Structure	Type	Local Description	Nulls	Formula
Employee	Employee	Employee		
EmployeeId	Numeric(5.0)	Employee Id	No	Subtype of PersonId
EmployeeFirstName	VarChar(50)	Employee First Name		Subtype of PersonFirstName
EmployeeLastName	VarChar(50)	Employee Last Name		Subtype of PersonLastName
EmployeeFullName	VarChar(102)	Employee Full Name		Subtype of PersonFullName
EmployeeTitle	VarChar(50)	Employee Title	No	

Figure 79

With its subtype:

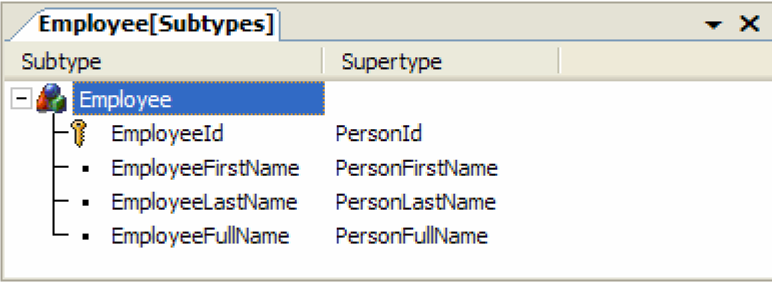


Figure 80

The Database Schema is as follows:

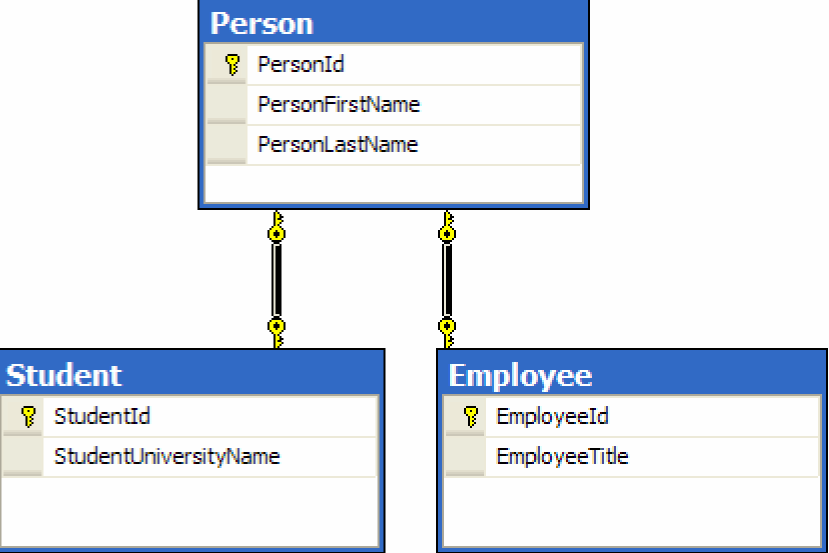


Figure 81

From a design best practices standpoint, there is controversy over whether Composition is better than Inheritance. In this example, if Composition is used instead of Inheritance, the Person Business Component remains the same. However, Student looks as follows:

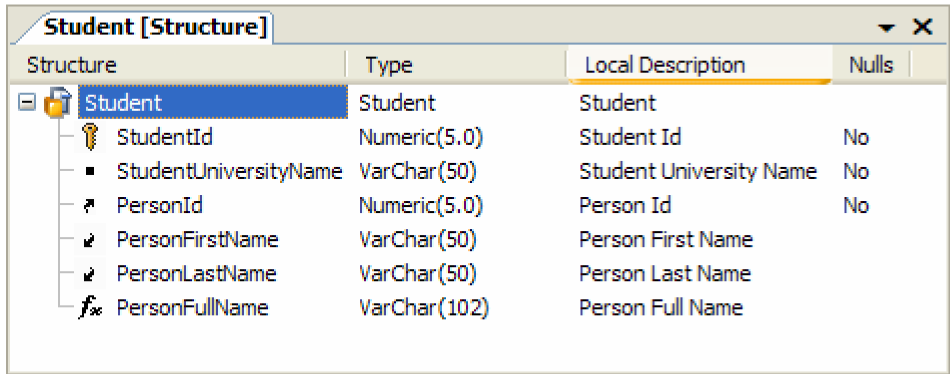


Figure 82

© Copyright 2002-2006 ARTech www.deklarit.com

This is Employee:

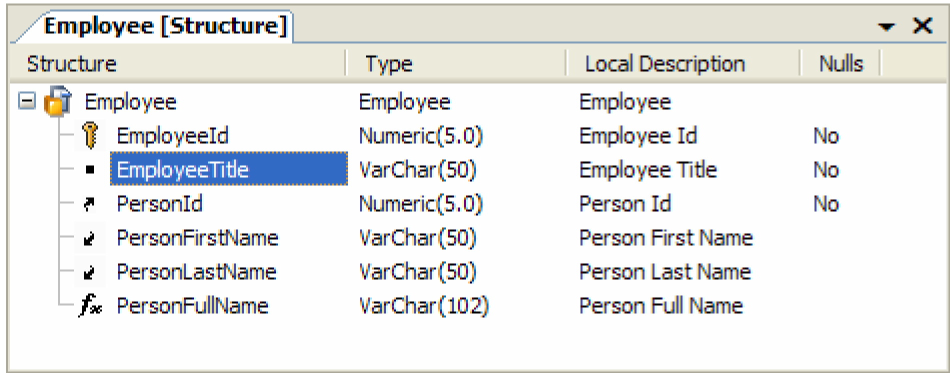


Figure 83

And the new Database Schema is slightly different:

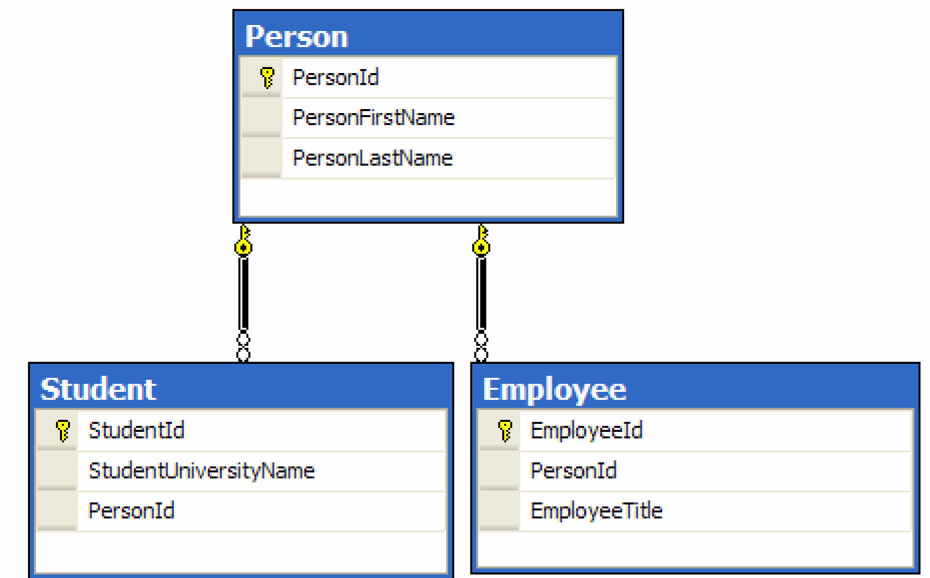


Figure 84

Composition seems to be more flexible (it is possible to change the Employee's PersonID) and it is more widely used than Inheritance.

As a rule of thumb, Subtypes make the Business Framework more complex; thus, they should be used only when necessary.

Implementing Inheritance

The previous section explains how to accomplish inheritance using subtypes, but if you want to automatically create the Person record when you add a new Student or Employee, you can easily achieve it using rules. Below is a sample for the Student Business Component.

First, you need to add this rule to the Student Business Component:

InheritanceSample.Extensions.NewPersonRow(GetCurrentDataRow()) if insert;

The 'NewPersonRow' static method will add the person row using the Person Business Component; it should look as follows:

```
public static void NewPersonRow(StudentDataSet.StudentRow stdRow)
{
   PersonDataSet personDS = new PersonDataSet();
```

```
   PersonDataAdapter personDA = new PersonDataAdapter();
   PersonDataSet.PersonRow perRow = personDS.Person.NewPersonRow();
   if ( ! stdRow.IsStudentFirstNameNull() )
   {
          perRow.PersonFirstName = stdRow.StudentFirstName;
   }
   if ( ! stdRow.IsStudentLastNameNull() )
   {
          perRow.PersonLastName = stdRow.StudentLastName;
   }
   personDS.Person.AddPersonRow(perRow);
   personDA.Update(personDS);
   stdRow.StudentId = perRow.PersonId;
}
Public Shared Function NewPersonRow(ByVal stdRow As StudentDataSet.StudentRow)
   Dim personDS As New PersonDataSet
   Dim personDA As New PersonDataAdapter
   Dim perRow As PersonDataSet.PersonRow = personDS.Person.NewPersonRow()
   If Not stdRow.IsStudentFirstNameNull() Then
          perRow.PersonFirstName = stdRow.StudentFirstName
   End If
   If Not stdRow.IsStudentLastNameNull() Then
          perRow.PersonLastName = stdRow.StudentLastName
   End If

   personDS.Person.AddPersonRow(perRow)
   personDA.Update(personDS)
   stdRow.StudentId = perRow.PersonId
End Function
```

To create a Person, you need to write one static method for each Business Component (Student, Employee, etc). You can have a more generic method for all of them but one disadvantage is that it will not support null values as the previous one does:

```
StudentId = InheritanceSample.Extensions.NewPersonRow( StudentFirstName, StudentLastName ) if
insert on BeforeValidate;
public static int NewPersonRow(string personFirstName, string personLastName)
{
   PersonDataSet personDS = new PersonDataSet();
   PersonDataAdapter personDA = new PersonDataAdapter();
   PersonDataSet.PersonRow perRow = personDS.Person.NewPersonRow();
   perRow.PersonFirstName = personFirstName;
   perRow.PersonLastName = personLastName;

   personDS.Person.AddPersonRow(perRow);
   personDA.Update(personDS);
   return perRow.PersonId;
}
Public Shared Function NewPersonRow(ByVal personFirstName As String, ByVal personLastName As
String)
   Dim personDS As New PersonDataSet
   Dim personDA As New PersonDataAdapter
   Dim perRow As PersonDataSet.PersonRow = personDS.Person.NewPersonRow()
   perRow.PersonFirstName = personFirstName
   perRow.PersonLastName = personLastName
   personDS.Person.AddPersonRow(perRow)
   personDA.Update(personDS)
   return perRow.PersonId
End Function
```

Add this Update rule to the Student Business Component:

Update(StudentName);

This update rule is not necessary unless you want to update the Person attributes when updating the subtypes in Student. If you do not define the Update rule, the Business Framework Prototyper application and the add-in-generated applications will treat those subtype secondary attributes as read only in the Student Form; also, the updates to those attributes in the 'StudentDataSet' will be discarded.

To delete the parent record when deleting the child, you should add a rule to the Student Business Component:

```
InheritanceSample.Extensions.DeletePersonRow( StudentId ) on AfterDelete;
public static void DeletePersonRow(int personId)
{
    PersonDataSet personDS = new PersonDataSet();
    PersonDataAdapter personDA = new PersonDataAdapter();
    personDA.Fill(personDS, personId);
    personDS.Person[0].Delete();
    personDA.Update(personDS);
}
Public Shared Function DeletePersonRow (ByVal personId As Integer)
    Dim personDS As New PersonDataSet
    Dim personDA As New PersonDataAdapter
    personDA.Fill(personDS, personId)
    personDS.Person(0).Delete()
    personDA.Update(personDS)
End Function
```

Physical Database Design

Given a set of Business Components, DeKlarit automatically defines the Database Schema necessary to support them. In general, you do not need to work with the Physical Schema, but sometimes it is necessary to modify it. DeKlarit lets you change certain things that will not modify the model's consistency.

When changing the Physical Model, once you have generated the Business Framework DeKlarit automatically creates the script necessary to update the Database Schema.

Table Name

The first thing you can change is Table names. DeKlarit takes the default Table name from the Business Component name. In the Broker example:

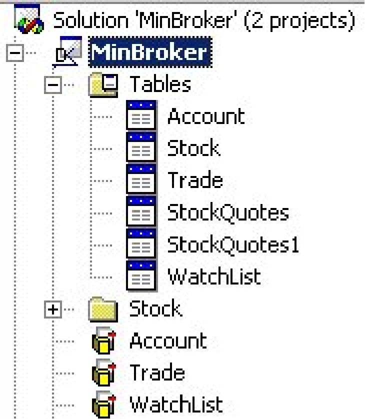


Figure 85

Note that the StockQuotes Business Component has two levels, so the table name corresponding to the second level is StockQuotes**1**.

To rename a Table, right-click it and choose the Rename option.

Indexes

DeKlarit automatically defines all the indexes needed to efficiently handle Referential Integrity Constraints. For example, if you open the Trade Table you see the following predefined indexes:

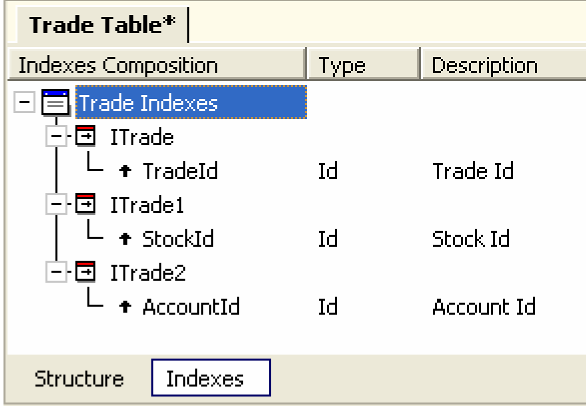


Figure 86

The first one supports the Primary Key, and the second and third ones are for the Referential Integrity with Stock and Account, respectively.

Sometimes, other indexes could improve the Business Framework's performance. For example, take the TradesByDay Data Provider:

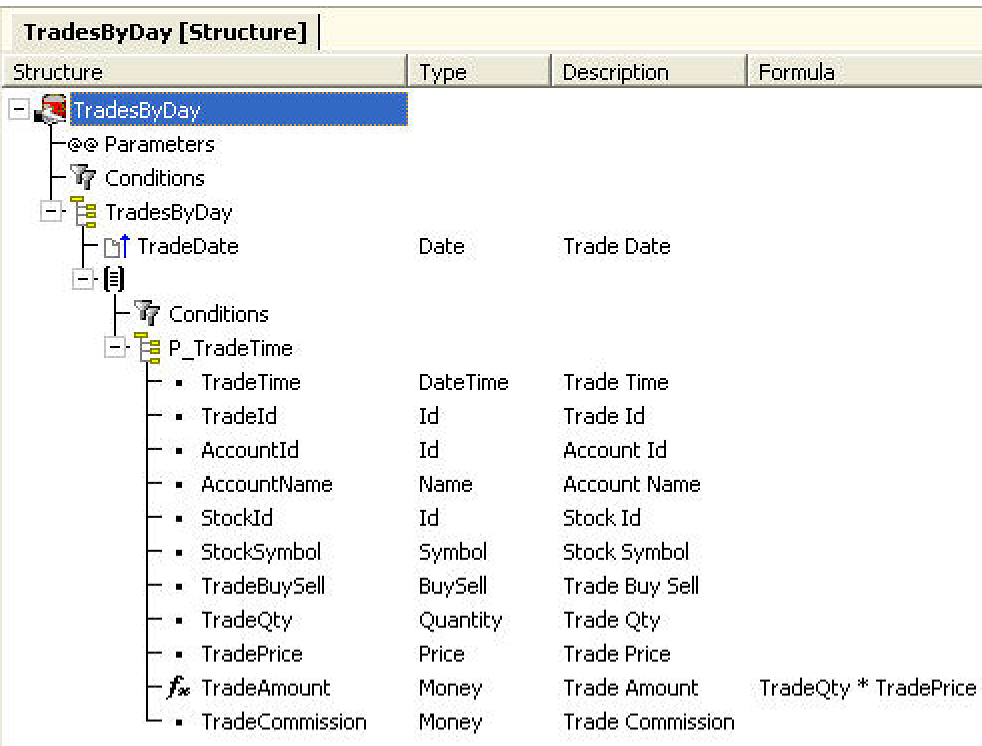


Figure 87

Its Navigation Report has a Warning, as shown in Figure 88.

Figure 88

This warning indicates that since there is no index for TradeDate in Trade, the server is forced to sort the data every time the Data Provider is executed, which can cause performance problems.

It can be solved by creating the required Index:

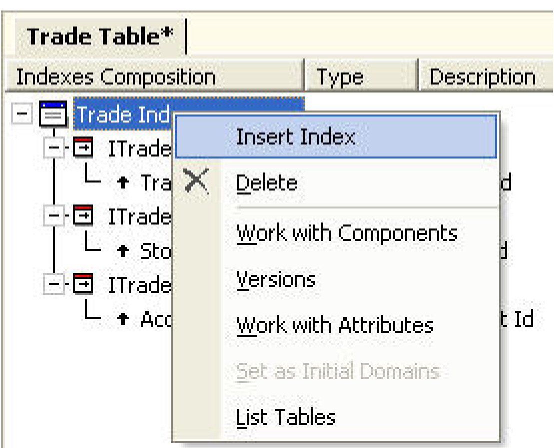


Figure 89

The next time you generate the Data Provider the Warning will not be present.

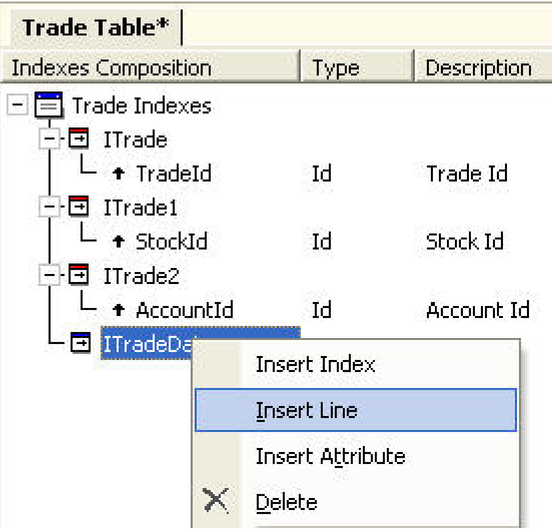


Figure 90

© Copyright 2002-2006 ARTech www.deklarit.com

Figure 91

Do not create an Index every time you get this kind of Warning. Whether it is better to create the Index or let the server sort the data depends on many parameters such as quantity of rows, frequency, etc.

It is a good idea to use DeKlarit's Incremental Development features, and to create the Index only if you experience performance problems. In other words, it is very easy to add the Index later, so do not rush.

After renaming a Table or creating an Index, when you regenerate the Business Framework the program executes a Database reorganization.